Journal of Computer Security 26 (2018) 459–508
DOI 10.3233/JCS-15769
IOS Press

Abstractions for security protocol verification

Binh Thanh Nguyen [a,*,**], Christoph Sprenger [b] and Cas Cremers [c]
[a] *Institute of Networks and Security, Johannes Kepler University, Linz, Austria*
[b] *Institute of Information Security, Department of Computer Science, ETH Zurich, Switzerland*
[c] *Department of Computer Science, University of Oxford, United Kingdom*

Abstract. We present a large class of security protocol abstractions with the aim of improving the scope and efficiency of verification tools. We propose abstractions that transform a term's structure based on its type as well as abstractions that remove atomic messages, variables, and redundant terms. Our theory improves on previous work by supporting rewrite theories with the finite-variant property, user-defined types, and untyped variables to cover type flaw attacks. We prove soundness results for an expressive property language that includes secrecy and authentication. Applying our abstractions to realistic IETF protocol models, we achieve dramatic speedups and extend the scope of several modern security protocol analyzers.

Keywords: Security protocols, formal verification, abstraction technique

1. Introduction

Security protocols play a central role in today's networked applications. Past experience has amply shown that informal arguments justifying the security of such protocols are insufficient. This makes security protocols prime candidates for formal verification. In the last two decades, research in formal security protocol verification has made enormous progress, which is reflected in many state-of-the-art tools including AVANTSSAR [5], ProVerif [9], Maude-NPA [21], Scyther [14], and Tamarin [34]. These tools can verify small to medium-sized protocols in a few seconds or less, sometimes for an unbounded number of sessions. Despite this success, they can still be challenged when verifying real-world protocols such as those defined in standards and deployed on the internet (e.g., TLS, IKE, and ISO/IEC 9798). Such protocols typically have messages with numerous fields, support many alternatives (e.g., cryptographic setups), and may be composed from more basic protocols (e.g., IKEv2-EAP).

Abstraction [10] is a standard technique to over-approximate complex systems by simpler ones to make verification more efficient or feasible. Sound abstractions preserve counterexamples (or attacks in security terms) from concrete to abstracted systems. In the context of security protocols, abstractions are extensively used. Here, we only mention a few examples. First, the Dolev–Yao model [19] is a standard (but not necessarily sound) abstraction of cryptography. Second, many tools encode the verification problem in the formalism of an efficient solver or reasoner. These encodings often involve abstraction as well. Therefore, we call these *back-end* abstractions. For example, ProVerif [9] translates models in the applied pi calculus to a set of Horn clauses, SATMC [6] reduces protocol verification to SAT solving, and Paulson [40] models protocols as inductively defined trace sets. Finally, some abstractions aim at

*Corresponding author. E-mail: binh@ins.jku.at.
**Most of this work was done while this author was working at ETH Zurich, Switzerland.

speeding up automated analysis by simplifying protocols within a given protocol model before feeding them to verifiers [28,37]. Our work belongs to this class of *front-end* abstractions.

Extending Hui and Lowe's work [28], we proposed in [37] a rich class of protocol abstractions and proved its soundness for a wide range of security properties. We used a type system to uniformly transform all terms of a given type (e.g., a pattern in a protocol role and its instances during execution) whereas [28] only covers ground terms. Our work [37] exhibits several limitations: (1) the theory is limited to the free algebra over a fixed signature; (2) all variables have strict (possibly structured) types, hence we cannot precisely model ticket forwarding or Diffie–Hellman exchanges. While the type system enables fine-grained control over abstractions (e.g., by discerning different nonces), it may eliminate realistic attacks such as type flaw attacks; (3) some soundness conditions involving quantifiers are hard to check in practice; and (4) it only presents experimental results for a single tool (SATMC) using abstractions that are crafted manually.

In this work, we address all the limitations above. First, we work with rewrite theories with the finite-variant property modulo a set of axioms to model cryptographic operations. Second, we support untyped variables, user-defined types, and subtyping. User-defined types enable the grouping of similar atomic types (e.g., session keys) and adjusting the granularity of matching in message abstraction. Furthermore, we have separated the removal of variables, atomic messages, and redundancies, from the transformation of the message structure. This separation simplifies the specifications and soundness proof of the abstractions that transform the message structure. Third, we provide effectively checkable syntactic criteria for the conditions of the soundness theorems. Finally, we extended Scyther [14] with fully automated support for our abstraction methodology. The resulting tool is available online [36]. We validated our approach on an extensive set of realistic case studies drawn from the IKEv1, IKEv2, ISO/IEC 9798, and PANA-AKA standard proposals. Our abstractions result in very substantial performance gains. We have also obtained positive results for several other state-of-the-art verifiers (ProVerif, CL-Atse, OFMC, and SATMC) with manually produced abstractions.

This article is based on the conference paper [38] from which it differs mainly as follows. On the theoretical side, we have generalized the class of supported rewrite systems from a subclass of shallow subterm-convergent ones to all those with the finite-variant property. Using the finite-variant property, we have significantly simplified the condition needed for equality preservation (Theorem 4.23). On the practical side, we provide additional details of the abstraction heuristics and the implementation. We have also extended the Scyther implementation with a check for spurious attacks. Moreover, we have performed several additional case studies.

Due to space constraints, most proofs are moved to the full version of the paper [39]. Table 1 gives an overview of the rest of the paper.

Table 1
Structure of paper

Topic	Main description
Motivating example: IKE	Section 2
Modeling security protocols	Section 3
Abstraction theory	Section 4
Abstraction generation algorithm	Section 5
Algorithm implementation in Scyther	Section 6.1
Experimental results	Section 6.2

2. Motivating example: An IKE protocol

The Internet Key Exchange (IKE) family of protocols is part of the IPsec protocol suite for securing Internet Protocol (IP) communication. IKE establishes a shared key, which is later used for securing IP packets, realizes mutual authentication, and offers identity protection as an option. Its first version (IKEv1) dates back to 1998 [27]. The second version (IKEv2) [30] significantly simplifies the first one. However, the protocols in this family are still complex and contain a large number of fields.

Concrete protocol. As our running example, we present a member of the IKEv2 family, called IKEv2-mac (or $\mathsf{IKE_m}$ for short), which sets up a session key using a Diffie–Hellman (DH) key exchange, provides mutual authentication based on MACs, and also offers identity protection. We use Cremers' models of IKE [15] as a basis for our presentation and experiments (see Section 6.2). Our starting point is the following concrete $\mathsf{IKE_m}$ protocol between an initiator A and a responder B, where we write $\{\!|m|\!\}_k$ to denote the symmetric encryption of m with key k.

$$\mathsf{IKE_m}(1).\ A \to B : SPIa, o, sA1, g^x, Na$$

$$\mathsf{IKE_m}(2).\ B \to A : SPIa, SPIb, sA1, g^y, Nb$$

$$\mathsf{IKE_m}(3).\ A \to B : SPIa, SPIb, \{\!|A, B, AUTHa, sA2, tSa, tSb|\!\}_{SK}$$

$$\mathsf{IKE_m}(4).\ B \to A : SPIa, SPIb, \{\!|B, AUTHb, sA2, tSa, tSb|\!\}_{SK}$$

Here, *SPIa* and *SPIb* denote the *Security Parameter Indices* (two unique values that together identify a connection), o is a constant number, *sA1* and *sA2* are *Security Associations* (a group of security parameters that the parties will agree on, such as the used cryptographic algorithms), g is the DH group generator, x and y are secret DH exponents, *Na* and *Nb* are nonces, and *tSa* and *tSb* denote *Traffic Selectors* specifying certain IP parameters. *AUTHa* and *AUTHb* denote the authenticators of A and B and *SK* the session key derived from the DH key g^{xy}. These are defined as follows.

$$SK = \mathsf{kdf}(Na, Nb, g^{xy}, SPIa, SPIb)$$

$$AUTHa = \mathsf{mac}(\mathsf{sh}(A, B), SPIa, o, sA1, g^x, Na, Nb, \mathsf{prf}(SK, A))$$

$$AUTHb = \mathsf{mac}(\mathsf{sh}(B, A), SPIa, SPIb, sA1, g^y, Nb, Na, \mathsf{prf}(SK, B))$$

We model the functions mac, kdf, and prf as hash functions and use $\mathsf{sh}(A, B)$ and $\mathsf{sh}(B, A)$ to refer to the (single) long-term symmetric key shared by A and B as part of the cryptographic setup.

We consider the following security properties:

(P1) the secrecy of the DH key g^{xy}, and
(P2) mutual non-injective agreement on the nonces *Na* and *Nb* and the DH half-keys g^x and g^y.

The DH key serves as the master secret for *SK*. We could also consider the secrecy of *SK*, but for the running example we only consider the simpler property.

Abstraction. Our theory supports the construction of abstract models by removing inessential fields and operations using a range of abstractions. Typically, we use abstractions in a first step to remove selected cryptographic operations, remove fields under hashes, and to pull fields outside other cryptographic operations like encryptions or signatures. The types enable a fine-grained selection of the messages to be abstracted. In a second step, we remove inessential top-level (i.e., unprotected) fields and redundancies.

Let us apply these two steps to the IKE_m protocol. In the first step, we remove: (i) the symmetric encryptions with the session key SK (providing identity protection), (ii) from the session key: all fields under kdf except the DH key g^{xy}, and (iii) from the authenticators: the fields $SPIa$, $SPIb$, and $sA1$ and the application of prf including the agent names underneath. Here is the resulting protocol, which we call $\mathsf{IKE}^1_\mathsf{m}$.

$$\mathsf{IKE}^1_\mathsf{m}(1).\ A \to B : SPIa, o, sA1, g^x, Na$$
$$\mathsf{IKE}^1_\mathsf{m}(2).\ B \to A : SPIa, SPIb, sA1, g^y, Nb$$
$$\mathsf{IKE}^1_\mathsf{m}(3).\ A \to B : SPIa, SPIb, A, B, AUTHa', sA2, tSa, tSb$$
$$\mathsf{IKE}^1_\mathsf{m}(4).\ B \to A : SPIa, SPIb, B, AUTHb', sA2, tSa, tSb$$

where $SK' = \mathsf{kdf}(g^{xy})$ and

$$AUTHa' = \mathsf{mac}(\mathsf{sh}(A, B), o, g^x, Na, Nb, SK')$$
$$AUTHb' = \mathsf{mac}(\mathsf{sh}(B, A), g^y, Nb, Na, SK').$$

Note that we keep the field o in $AUTHa'$ to prevent its unifiability with $AUTHb'$ and hence the potential introduction of spurious attacks. Here, the type system plays an essential role in that it allows us to distinguish $AUTHa$ (with constant o as its third field under the mac) from $AUTHb$ (where $SPIb$ is the third field under the mac, which we model as a nonce) and transform them in different ways resulting in $AUTHa'$ and $AUTHb'$.

In a second step, we use abstractions to remove the fields o, A, B, $SPIa$, $SPIb$, $sA1$, $sA2$, tSa, and tSb in unprotected positions. The resulting protocol is $\mathsf{IKE}^2_\mathsf{m}$:

$$\mathsf{IKE}^2_\mathsf{m}(1).\ A \to B : g^x, Na$$
$$\mathsf{IKE}^2_\mathsf{m}(2).\ B \to A : g^y, Nb$$
$$\mathsf{IKE}^2_\mathsf{m}(3).\ A \to B : AUTHa'$$
$$\mathsf{IKE}^2_\mathsf{m}(4).\ B \to A : AUTHb'$$

Scyther verifies the properties (P1) and (P2) in 8.7 s on the concrete and in 1.7 s on an automatically generated abstract protocol (which differs somewhat from the one presented here). Our soundness results imply that the original protocol IKE_m also enjoys these properties. We chose the protocol IKE_m as running example for its relative simplicity compared to the other protocols in our case studies. In many of our experiments (Section 6.2), our abstractions (i) result in much more substantial speedups, or (ii) enable the successful unbounded verification of a protocol where it times out or exhausts memory on the original protocol.

3. Security protocol model

We define a term algebra $\mathcal{T}_\Sigma(V)$ over a signature Σ and a set of variables V in the standard way. Let Σ^n denote the symbols of arity n. We call the elements of Σ^0 *atoms* and write $\Sigma^{\geqslant 1}$ for the set of

proper function symbols. For a fixed $\Sigma^{\geqslant 1}$, we will vary Σ^0 to generate different sets of terms, denoted by $\mathcal{T}(V, \Sigma^0)$, including terms in protocol roles, network messages, and types. We write $subterm(t)$ for the set of subterms of t. We also define $vars(t) = subterm(t) \cap V$ and $atoms(t) = subterm(t) \cap \Sigma_0$. If $vars(t) = \emptyset$ then t is called *ground*. We denote the top-level symbol of a (non-variable) term t by $topsym(t)$ and the set of its function symbols in $\Sigma^{\geqslant 1}$ by $funsym(t)$. We call a term t *composed* if $funsym(t)$ is non-empty. A *position* is a sequence of positive natural numbers denoting a path in the tree representation of a term. The *size* of a term t, denoted by $|t|$, is the cardinality of its set of positions. We denote the subterm of t at position p with $t|_p$ and write $t[u]_p$ for the term obtained by replacing $t|_p$ at position p by u. We also partition Σ into sets of public and private symbols, denoted by Σ_{pub} and Σ_{pri}. We assume Σ_{pub} includes pairing $\langle\cdot,\cdot\rangle$ which associates to the right, e.g., $\langle t, u, v\rangle = \langle t, \langle u, v\rangle\rangle$. We usually write, e.g., $\{\!|t, u, v|\!\}_k$ rather than $\{\!|\langle t, u, v\rangle|\!\}_k$. We take the liberty to lift functions on terms to functions on sets of terms T, e.g., $funsym(T) = \bigcup_{t\in T} funsym(t)$. We denote by $dom(g)$ and $ran(g)$ the domain and range of a function g. For $n \in \mathbb{N}$, $\widetilde{n}$ denotes $\{1, \ldots, n\}$.

The set of *message terms* is $\mathcal{M} = \mathcal{T}(\mathcal{V}, \mathcal{A}\cup\mathcal{F}\cup\mathcal{C})$, where $\mathcal{V}$, $\mathcal{A}$, $\mathcal{F}$, and $\mathcal{C}$ are pairwise disjoint infinite sets of variables, agents, fresh values, and constants. We use terms in $\mathcal{M}$ to model messages in *protocol definitions* which we present in Section 3.4. We partition $\mathcal{A}$ into sets of honest and compromised agents: $\mathcal{A} = \mathcal{A}_H \cup \mathcal{A}_C$. The set $fresh(t) = subterm(t) \cap \mathcal{F}$ denotes the fresh values in t. By convention, we use identifiers starting with upper-case and lower-case letters to denote variables and atoms, respectively.

3.1. Type system

We introduce a type system akin to [2] and extend it with subtyping. This type system is very fine-grained. For example, there are different types for different fresh values. We will subsequently restrict some abstractions to apply only to arguments of a specific type. Thus, the purpose of this fine-grained type system is to control when those abstractions are used. The subtyping allows us to adapt to different setups and tools by making types more coarse-grained. For example, we can define a type *nonce* as a supertype for all fresh values.

We define the set of atomic types by $\mathcal{Y}_{at} = \mathcal{Y}_0 \cup \{\alpha, msg\} \cup \{\beta_n \mid n \in \mathcal{F}\} \cup \{\gamma_c \mid c \in \mathcal{C}\}$, where α, β_n, and γ_c are the types of agents, the fresh value n, and the constant c, respectively. Moreover, msg is the type of all messages and $\mathcal{Y}_0$ is a disjoint set of user-defined types. The set of all types is then defined by $\mathcal{Y} = \mathcal{T}(\emptyset, \mathcal{Y}_{at})$.

We assume that all variables have an atomic type, i.e., $\mathcal{V} = \{\mathcal{V}_\tau\}_{\tau\in\mathcal{Y}_{at}}$ is a family of disjoint infinite sets of variables. Define $\Gamma : \mathcal{V} \to \mathcal{Y}_{at}$ by $\Gamma(X) = \tau$ if and only if $X \in \mathcal{V}_\tau$. We extend Γ to atoms by defining $\Gamma(a) = \alpha$, $\Gamma(n) = \beta_n$, and $\Gamma(c) = \gamma_c$ for $a \in \mathcal{A}$, $n \in \mathcal{F}$, and $c \in \mathcal{C}$, and then homomorphically to all terms $t \in \mathcal{M}$. Note that Γ is unique. We call $\tau = \Gamma(t)$ the *type of* t and sometimes also write $t : \tau$.

The subtyping relation $\preccurlyeq$ on types is defined by the following inference rules and by two additional rules (not shown) defining its reflexivity and transitivity.

$$\frac{\tau \in \mathcal{Y}}{\tau \preccurlyeq msg}\ \mathrm{S}(msg) \qquad \frac{\tau_1 \preccurlyeq_0 \tau_2}{\tau_1 \preccurlyeq \tau_2}\ \mathrm{S}(\preccurlyeq_0) \qquad \frac{\tau_1 \preccurlyeq \tau_1' \quad \cdots \quad \tau_n \preccurlyeq \tau_n'}{c(\tau_1, \ldots, \tau_n) \preccurlyeq c(\tau_1', \ldots, \tau_n')}\ \mathrm{S}(c \in \Sigma^n)$$

Every type is a subtype of msg by the first rule. The second rule embeds a user-defined *atomic subtyping relation* $\preccurlyeq_0 \subseteq (\mathcal{Y}_{at} \setminus \{msg\}) \times \mathcal{Y}_0$, which relates atomic types (except msg) to user-defined atomic types in $\mathcal{Y}_0$. For simplicity, we require that $\preccurlyeq_0$ is a partial function. The third rule ensures that subtyping is preserved by all symbols. The set of subtypes of τ is $\tau\downarrow = \{\tau' \in \mathcal{Y} \mid \tau' \preccurlyeq \tau\}$.

3.2. Equational theories

An *equation* over a signature Σ is an unordered pair $\{s, t\}$, written $s \simeq t$, where $s, t \in \mathcal{T}_\Sigma(\mathcal{V}_{msg})$. An *equation presentation* $\mathcal{E} = (\Sigma, E)$ consists of a signature Σ and a set E of equations over Σ. The *equational theory* induced by $\mathcal{E}$ is the smallest Σ-congruence, written $=_E$, containing all instances of equations in E. We often identify $\mathcal{E}$ with the induced equational theory.

A *rewrite rule* is an oriented pair $l \to r$, where $vars(r) \subseteq vars(l) \subseteq \mathcal{V}_{msg}$. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, Ax, R)$ where Σ is a signature, Ax a set of Σ-equations such that $vars(s) = vars(t)$ for all $s \simeq t \in Ax$, and R a set of rewrite rules. The rewriting relation $\to_{R,Ax}$ on $\mathcal{T}_\Sigma(\mathcal{V})$ is defined by $t \to_{R,Ax} t'$ iff there exists a non-variable position p in t, a rule $l \to r \in R$, and a substitution σ such that $t|_p =_{Ax} l\sigma$ and $t' = t[r\sigma]_p$. If $t \to^*_{R,Ax} t'$ and t' is irreducible under $\to_{R,Ax}$, we call t' R, Ax-*normal* and also say that t' is a *normal form* of t. A substitution σ is called R, Ax-normal if all terms in $ran(\sigma)$ are.

Provided that Ax has a finitary and complete unification algorithm and under suitable termination, confluence, and coherence conditions (see [29] for definitions), one can decompose an equational theory (Σ, E) into a rewrite theory (Σ, Ax, R) where $E = Ax \cup R$ (reading R here as a set of equations) and, for all terms $t, u \in \mathcal{T}_\Sigma(\mathcal{V})$, we have $t =_E u$ if and only if $t\downarrow_{R,Ax} =_{Ax} u\downarrow_{R,Ax}$. Here, $t\downarrow_{R,Ax}$ denotes any normal form of t. Well-formed rewrite theories, defined below, satisfy a few additional mild assumptions.

Definition 3.1. A rewrite theory (Σ, Ax, R) is *well-formed* if for all $s \simeq t \in Ax$ and all $l \to r \in R$, we have (i) $vars(s) = vars(t)$ and $vars(r) \subseteq vars(l)$, (ii) $topsym(s) = topsym(t)$, (iii) s, t, and l are composed and neither of them is a pair, and (iv) s, t, l, and r do not contain any fresh values.

The equality $vars(s) = vars(t)$ in point (i) of this definition is a standard assumption made for rewrite theories known as regularity [22]. Such rewrite theories are adequate to model many well-known cryptographic primitives as illustrated by the examples below.

Example 3.2. We model the protocols of our case studies (see Section 2 and Section 6.2) in the rewrite theory $\mathcal{R}_{cs} = (\Sigma_{cs}, Ax_{cs}, R_{cs})$ where

$$\Sigma_{cs} = \{\mathsf{sh}, \mathsf{pk}, \mathsf{pri}, \mathsf{prf}, \mathsf{kdf}, \mathsf{mac}, \mathsf{exp}, \langle\cdot,\cdot\rangle, \pi_1, \pi_2, \{\!|\cdot|\!\}_., \{\!|\cdot|\!\}_.^{-1}, \{\cdot\}_., \{\cdot\}_.^{-1}, [\cdot]_., \mathsf{ver}\} \cup \Sigma^0_{cs}$$

contains function symbols for: shared, public, and private long-term keys (where $\Sigma_{\mathsf{pri}} = \{\mathsf{sh}, \mathsf{pri}\}$); hash functions prf, kdf, and mac; exponentiation exp; pairs and projections; symmetric and asymmetric encryption and decryption; and signing and verification. The set of atoms Σ^0_{cs} is specified later. The set R_{cs} consists of rewrite rules for projections, decryption, and signature verification (with message recovery):

$$\pi_1(\langle X, Y\rangle) \to X \qquad \{\!|\{\!|X|\!\}_K|\!\}_K^{-1} \to X \qquad \mathsf{ver}([X]_{\mathsf{pri}(Y)}, \mathsf{pk}(Y)) \to X$$
$$\pi_2(\langle X, Y\rangle) \to Y \qquad \{\{X\}_{\mathsf{pk}(Y)}\}^{-1}_{\mathsf{pri}(Y)} \to Y$$

We have two equations in Ax_{cs}, namely, $\mathsf{exp}(\mathsf{exp}(g, X), Y) \simeq \mathsf{exp}(\mathsf{exp}(g, Y), X)$ to model Diffie–Hellman key exchange and $\mathsf{sh}(X, Y) \simeq \mathsf{sh}(Y, X)$. Note that the rewrite rule for signature verification models signatures with message recovery (as, e.g., for RSA signatures). In contrast, MACs do not provide message recovery, so they have to be reconstructed for verification.

Example 3.3. The theory of XOR is given by the following rewrite system. The rightmost rule is redundant but required to ensure coherence [29].

$$X \oplus Y \simeq Y \oplus X \qquad X \oplus 0 \to X \qquad X \oplus X \oplus Y \to Y$$
$$(X \oplus Y) \oplus Z \simeq X \oplus (Y \oplus Z) \qquad X \oplus X \to 0$$

We have used the AProVE termination tool [23] and Maude's Church-Rosser and coherence checker [20] to verify the termination, confluence, and coherence properties that are required for decomposing the equational theories of our case studies.

Finally, we define well-typed substitutions, which are substitutions that respect subtyping.

Definition 3.4 (Well-typed substitutions)**.** A substitution θ is *well-typed* if $\Gamma((X\theta)\downarrow_{R,Ax}) \preccurlyeq \Gamma(X)$ for all $X \in dom(\theta)$.

Since the type of any variable is atomic, this definition is independent of the representative of the Ax-equivalence class chosen for the normalized term. Hence, it is well-defined.

3.3. The finite variant property

The finite variant property simplifies equality checking and unification in equational theories. Given an equational theory $\mathcal{E} = (\Sigma, E)$ and a term t, an *$\mathcal{E}$-variant of t* is a pair (t', θ) such that $t\theta =_E t'$. A decomposition $\mathcal{R} = (\Sigma, Ax, R)$ of $\mathcal{E}$ (and hence $\mathcal{E}$) has the *finite variant property* if for all terms $t \in \mathcal{T}_\Sigma(V)$, there is a finite set $\{(t_1, \theta_1), \ldots, (t_n, \theta_n)\}$ of $\mathcal{E}$-variants of t such that t_i is R, Ax-normal and $dom(\theta_i) \subseteq vars(t)$ for all $i \in \widetilde{n}$, and for all substitutions σ, there are a substitution η and $i \in \widetilde{n}$ such that

(i) $(t\sigma)\downarrow_{R,Ax} =_{Ax} t_i\eta$,
(ii) $X\sigma\downarrow_{R,Ax} =_{Ax} (X\theta_i)\eta$ for all $X \in vars(t)$.

We also call $\mathcal{R}$ a *finite-variant decomposition* of $\mathcal{E}$. Given a such a decomposition, the algorithm in [22], based on the folding-variant narrowing strategy, computes a finite, complete, and minimal set of R, Ax-variants of a given term t, denoted by $[\![t]\!]_{R,Ax}$. This set is unique up to $=_{Ax}$-equality.

Example 3.5. Consider the XOR theory from Example 3.3 and the terms $s = X \oplus Y \oplus X$ and $t = X \oplus Y$. Then, with id denoting the identity substitution, the complete and minimal sets of R, Ax-variants of these terms are $[\![s]\!]_{R,Ax} = \{(Y, id)\}$ and

$$\begin{aligned}[\![t]\!]_{R,Ax} = \{&(X \oplus Y, id),\\ &(Z, \{X \mapsto 0, Y \mapsto Z\}),\\ &(Z, \{X \mapsto Z, Y \mapsto 0\}),\\ &(Z, \{X \mapsto Z \oplus U, Y \mapsto U\}),\\ &(Z, \{X \mapsto U, Y \mapsto Z \oplus U\}),\\ &(0, \{X \mapsto U, Y \mapsto U\}),\\ &(Z_1 \oplus Z_2, \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\})\}.\end{aligned}$$

Assumption 3.6. For our theoretical development, we consider an arbitrary but fixed equational theory $\mathcal{E} = (\Sigma, E)$ with a well-formed finite-variant decomposition $\mathcal{R} = (\Sigma, Ax, R)$. We also assume that $\mathcal{R}$ includes function symbols and rewrite rules for pairing and projections.

3.4. Protocols

We specify a security protocol as a partial function from agent variables to *roles*. A role is a sequence of events. We distinguish three types of events: *send* events, *receive* events, and *signal* events. A send event $\mathsf{send}(t)$ indicates the transmission of a message that is an instance of the term t. Likewise, a receive event $\mathsf{recv}(t)$ indicates the reception of a message that matches t. We assume a fixed set Sig of signal events disjoint from $\{\mathsf{send}, \mathsf{recv}\}$. A signal event $sig \in Sig$ marks a progressive stage of an agent playing a role, i.e., it tells how far the agent has been executing. We use signal events to specify security properties. Past research has also employed signal events to express various authentication properties [43,44].

Given a set of terms T, we define the set of events $Evt(T) = \{\mathsf{send}(t), \mathsf{recv}(t) \mid t \in T\} \cup Sig$. We also define $term(ev(t)) = t$ for event $ev \in \{\mathsf{send}, \mathsf{recv}\}$ and leave it undefined for signals. A *role* is a sequence of events from $Evt(\mathcal{M})$. We lift $term(\cdot)$ in the obvious way to sets and sequences of events.

Definition 3.7 (Protocol)**.** A *protocol* is a partial function $P : \mathcal{V}_\alpha \rightharpoonup Evt(\mathcal{M})^*$ mapping agent variables to roles. Let $\mathcal{M}_P = term(ran(P))$ be the set of *protocol terms* appearing in the roles of P, and let $\mathcal{V}_P$, $\mathcal{A}_P$, $\mathcal{F}_P$, and $\mathcal{C}_P$ denote the sets of variables, agents, fresh values, and constants in $\mathcal{M}_P$.

Example 3.8 ($\mathsf{IKE_m}$ protocol)**.** We formalize the $\mathsf{IKE_m}$ protocol from Section 2 in the rewrite theory of Example 3.2 as follows. The atoms Σ^0_{cs} are composed of constants $C = \{g, o, sA1, sA2, tSa, tSb\}$ and fresh values $F = \{na, nb, x, y, sPIa, sPIb\}$. The variables and their types are $A, B : \alpha$, $Ga, Gb : msg$, $SPIa, SPIb, Na, Nb : nonce$ where $nonce$ is a user-defined type that satisfies $\beta_n \preccurlyeq_0 nonce$ for all $n \in F$. We model mac, kdf, and prf as hash functions. We also assume a set of signal events $Sig = \{\mathsf{Running}, \mathsf{Commit}, \mathsf{Secret}\}$. We later use $\mathsf{Running}$ and Commit to specify authentication properties and Secret to specify secrecy properties (see Example 3.11). We formulate the initiator role A and the responder role B as follows.

$$\begin{aligned}
\mathsf{IKE_m}(A) = {} & \mathsf{send}(sPIa, o, sA1, \mathsf{exp}(g, x), na) \cdot \mathsf{recv}(sPIa, SPIb, sA1, Gb, Nb) \cdot \mathsf{Running} \cdot \\
& \mathsf{send}(sPIa, SPIb, \{\!|A, B, AUTHaa, sA2, tSa, tSb|\!\}_{SKa}) \cdot \\
& \mathsf{recv}(sPIa, SPIb, \{\!|B, AUTHba, sA2, tSa, tSb|\!\}_{SKa}) \cdot \mathsf{Secret} \cdot \mathsf{Commit} \\
\mathsf{IKE_m}(B) = {} & \mathsf{recv}(SPIa, o, sA1, Ga, Na) \cdot \mathsf{send}(SPIa, sPIb, sA1, \mathsf{exp}(g, y), nb) \cdot \\
& \mathsf{recv}(SPIa, sPIb, \{\!|A, B, AUTHab, sA2, tSa, tSb|\!\}_{SKb}) \cdot \mathsf{Running} \cdot \\
& \mathsf{send}(SPIa, sPIb, \{\!|B, AUTHbb, sA2, tSa, tSb|\!\}_{SKb}) \cdot \mathsf{Secret} \cdot \mathsf{Commit}
\end{aligned}$$

where the terms

$$\begin{aligned}
SKa &= \mathsf{kdf}(na, Nb, \mathsf{exp}(Gb, x), sPIa, SPIb) \\
SKb &= \mathsf{kdf}(Na, nb, \mathsf{exp}(Ga, y), SPIa, sPIb) \\
AUTHaa &= \mathsf{mac}(\mathsf{sh}(A, B), sPIa, o, sA1, \mathsf{exp}(g, x), na, Nb, \mathsf{prf}(SKa, A))
\end{aligned}$$

$$AUTHab = \mathsf{mac}(\mathsf{sh}(B, A), SPIa, o, sA1, Ga, Na, nb, \mathsf{prf}(SKb, A))$$

$$AUTHba = \mathsf{mac}(\mathsf{sh}(A, B), sPIa, SPIb, sA1, Gb, Nb, na, \mathsf{prf}(SKa, B))$$

$$AUTHbb = \mathsf{mac}(\mathsf{sh}(B, A), SPIa, sPIb, sA1, \mathsf{exp}(g, y), nb, Na, \mathsf{prf}(SKb, B))$$

respectively represent the initiator A and the responder B's view of the session key SK and of the authenticators $AUTHa$ and $AUTHb$.

3.5. Operational semantics

In this section, we introduce an operational semantics for security protocols. This semantics specifies the dynamic behaviour of the protocol roles when their events are executed. The protocol messages are sent to and received from the adversary, whom we identify with the network as usual.

We use a Dolev–Yao adversary model parametrized by an equational theory E. Its judgements are of the form $T \vdash_E t$ meaning that the intruder can derive term t from the set of terms T. The derivable judgements are defined in a standard way by the three deduction rules in Fig. 1.

When a protocol is executed, each of its roles can be executed an arbitrary number of times by possibly different agents in parallel. Such a single execution of a role is called a *thread*. We distinguish between different threads by associating each thread with a unique *thread identifier*. We index variables and fresh values with the thread identifier i to syntactically distinguish them from those of other threads. This ensures the uniqueness of fresh values.

Let TID be a countably infinite set of thread identifiers. We define the *indexing* of a term t with $i \in TID$ as the term t^i where every variable or fresh value u is replaced by u^i. Constants and agents remain unchanged. For a set of messages $M \subseteq \mathcal{M}$, we define by $M^{TID} = \{t^i \mid t \in M \wedge i \in TID\}$ the corresponding set of indexed terms. We assume that $\mathcal{V} \cap \mathcal{V}^{TID} = \emptyset$ and $\mathcal{F} \cap \mathcal{F}^{TID} = \emptyset$. For variables and fresh values u, we define $\Gamma(u^i) = \Gamma(u)$. Hence, indexing a term does not affect its type, i.e., we have $\Gamma(t^i) = \Gamma(t)$. We extend indexing to (send and receive) events by applying it to the terms they contain. We also define the set of intruder-generated fresh values as $\mathcal{F}^\bullet = \{n_k^\bullet \mid n \in \mathcal{F} \wedge k \in \mathbb{N}\}$ with types $\Gamma(n_k^\bullet) = \Gamma(n) = \beta_n$.

For example, suppose that thread i plays role A and is owned by *alice*. Hence, the agent variable A^i is bound to *alice*. Suppose thread i contains a receive event $\mathsf{recv}(\{na, Nb\}_{\mathsf{pk}(A)})$, meaning that it expects a message of the form $\{na^i, m\}_{\mathsf{pk}(alice)}$ for some message m, which is bound to the variable Nb^i. Such a message might originate from some thread j (e.g., with $m = nb^j$ a nonce generated by thread j) or from the adversary (e.g., with $m = n_0^\bullet$ a nonce generated by the adversary).

We thus define the set of *network messages* exchanged during protocol executions by

$$\mathcal{N} = \mathcal{T}(\mathcal{V}^{TID}, \mathcal{A} \cup \mathcal{C} \cup \mathcal{F}^{TID} \cup \mathcal{F}^\bullet),$$

Note that $\mathcal{M}^{TID} \subseteq \mathcal{N}$.

$$\frac{u \in T}{T \vdash_E u}\ \mathsf{Ax} \qquad \frac{T \vdash_E t' \quad t' =_E t}{T \vdash_E t}\ \mathsf{Eq} \qquad \frac{T \vdash_E t_1 \quad \cdots \quad T \vdash_E t_n}{T \vdash_E f(t_1, \ldots, t_n)}\ \mathsf{Comp}\ (f \in \Sigma_{\mathsf{pub}}^{\geqslant 1})$$

Fig. 1. Intruder deduction rules (where $\Sigma_{\mathsf{pub}}^{\geqslant 1} = \Sigma^{\geqslant 1} \cap \Sigma_{\mathsf{pub}}$).

$$\frac{th(i) = (R, \mathsf{send}(t) \cdot tl)}{(tr, th, \sigma) \to (tr \cdot (i, \mathsf{send}(t)), th[i \mapsto (R, tl)], \sigma)}\ \textit{SEND}$$

$$\frac{th(i) = (R, \mathsf{recv}(t) \cdot tl) \quad IK(tr)\sigma \cup IK_0 \vdash_E t^i\sigma}{(tr, th, \sigma) \to (tr \cdot (i, \mathsf{recv}(t)), th[i \mapsto (R, tl)], \sigma)}\ \textit{RECV}$$

$$\frac{th(i) = (R, s \cdot tl) \quad s \in Sig}{(tr, th, \sigma) \to (tr \cdot (i, s), th[i \mapsto (R, tl)], \sigma)}\ \textit{SIGNAL}$$

Fig. 2. Operational semantics.

Given a protocol P, we define a transition system with states (tr, th, σ), where

- $tr \in (TID \times Evt(\mathcal{M}_P))^*$ is a *trace* consisting of a sequence of pairs of thread identifiers and events,
- $th : TID \rightharpoonup dom(P) \times Evt(\mathcal{M}_P)^*$ are *threads*, each executing some protocol role, and
- $\sigma : \mathcal{V}^{TID} \rightharpoonup \mathcal{N}$ is a well-typed ground substitution mapping instantiated protocol variables to network messages.

The trace tr as well as the executing role are symbolic (with terms in $\mathcal{M}_P$). The substitution σ instantiates these messages to (ground) network messages as follows. The ground trace $tr\sigma \in Evt(\mathcal{N})$ associated with such a state is recursively defined by

$$\epsilon\sigma = \epsilon \quad \text{and} \quad ((i, e) \cdot tr)\sigma = (i, e^i\sigma) \cdot tr\sigma.$$

where ϵ denotes the empty sequence. The set $Init_P$ of initial states is defined by

$$Init_P = \{(\epsilon, th, \sigma) \mid \forall i \in dom(th).\ \exists R \in dom(P).\ th(i) = (R, P(R)) \wedge \mathcal{V}_P^{TID} \subseteq dom(\sigma)\}.$$

The rules in Fig. 2 define the transitions. The first premise of each rule respectively states that a send, receive, or signal event heads thread i's role. This event is removed and added together with the thread identifier i to the trace tr. The second premise of *RECV* requires that the network message $t^i\sigma$ matching the term t in the receive event is derivable from the intruder's (ground) knowledge $IK(tr)\sigma \cup IK_0$. Here, $IK(tr)$ denotes the (symbolic) intruder knowledge derived from a trace tr as the set of terms in the send events on tr, instantiated with the respective thread id, i.e.,

$$IK(tr) = \{t^i \mid (i, \mathsf{send}(t)) \in tr\}$$

and IK_0 denotes the intruder's (ground) initial knowledge. Note that the *SEND* rule thus implicitly updates the intruder knowledge. The rule *SIGNAL* expresses that the signal events' only effect is to record a signal $s \in Sig$ in the trace. Note that transitions do not change the substitution σ; it is fixed with the (non-deterministic) choice of the initial state.

Finally, we define the semantics of a protocol P with respect to the intruder's initial knowledge IK_0 as the set of states reachable from the initial states:

$$reach(P, IK_0) = \{(tr, th, \sigma) \mid \exists s_0 \in Init_P.\ s_0 \to^* (tr, th, \sigma)\}$$

where $\to^*$ is the reflexive-transitive closure of the transition relation $\to$. Note that these relations depend on IK_0 due to the rule *RECV*. Later, we will use several sets representing the intruder's initial knowledge for which we state the following global assumption.

Assumption 3.9 (Intruder's initial knowledge)**.** We assume that the intruder's initial knowledge IK_0 is a set of R, Ax-normal ground network messages that contains all constants, agents, and intruder-generated fresh values, but no fresh values generated by the protocol, i.e., $\mathcal{C} \cup \mathcal{A} \cup \mathcal{F}^{\bullet} \subseteq IK_0$ and $\mathcal{F}^{TID} \cap IK_0 = \emptyset$.

This assumption specifies the minimal requirements. The attacker usually also knows the long-term shared and private keys of the compromised agents and the public keys of all agents, i.e., the keys in $\mathsf{sh}(\mathcal{A}_C, \mathcal{A})$, $\mathsf{sh}(\mathcal{A}, \mathcal{A}_C)$, $\mathsf{pri}(\mathcal{A}_C)$, and $\mathsf{pk}(\mathcal{A})$. However, since our proofs do not rely on these keys being included in IK_0, they do not appear in our assumption.

Example 3.10 (Example trace)**.** We provide an example trace of a partial honest execution. In this trace, Alice performs a partial session with Bob, up to the point of Bob's Secret. Consider the initial state $s_0 = (\epsilon, th, \sigma)$ where th at least contains

$$th(1) = (A, P(A))$$
$$th(2) = (B, P(B))$$

and where σ meets the condition

$$\begin{aligned}\sigma \supseteq \{&A^1 \mapsto alice, A^2 \mapsto alice,\\ &B^1 \mapsto bob, B^2 \mapsto bob,\\ &Gb^1 \mapsto \mathsf{exp}(g, y^2), Ga^2 \mapsto \mathsf{exp}(g, x^1),\\ &SPIb^1 \mapsto sPIb^2, SPIa^2 \mapsto sPIa^1,\\ &Nb^1 \mapsto nb^2, Na^2 \mapsto na^1\}\end{aligned}$$

In this case, one reachable state (tr, th', σ) has the trace:

$$\begin{aligned}tr = (&1, \mathsf{send}(sPIa, o, sA1, \mathsf{exp}(g, x), na)) \cdot\\ &(2, \mathsf{recv}(SPIa, o, sA1, Ga, Na)) \cdot\\ &(2, \mathsf{send}(SPIa, sPIb, sA1, \mathsf{exp}(g, y), nb)) \cdot\\ &(1, \mathsf{recv}(sPIa, SPIb, sA1, Gb, Nb)) \cdot\\ &(1, \mathsf{Running}) \cdot\\ &(1, \mathsf{send}(sPIa, SPIb, \{\!|A, B, AUTHaa, sA2, tSa, tSb|\!\}_{SKa})) \cdot\\ &(2, \mathsf{recv}(SPIa, sPIb, \{\!|A, B, AUTHab, sA2, tSa, tSb|\!\}_{SKb})) \cdot\\ &(2, \mathsf{Running}) \cdot\\ &(2, \mathsf{send}(SPIa, sPIb, \{\!|B, AUTHbb, sA2, tSa, tSb|\!\}_{SKb})) \cdot\\ &(2, \mathsf{Secret})\end{aligned}$$

where th' denotes the threads after executing these events and *SKa*, *SKb*, *AUTHaa*, *AUTHab*, and *AUTHbb* are as defined in Example 3.8.

In this trace, the adversary does not interfere. There are also traces in which he does interfere, e.g., traces in which the adversary sends the first message. In such traces, the first event could be a responder receive, for a suitable choice of σ in the initial state.

3.6. Property language

Meier et al. [33] define a predicate-based security property language. In this language, many security properties such as those from [13,16,32] can be specified. In this section, we introduce a specification language for security properties based on [33]. Our language is similar to the languages used in [1,21,26].

Syntax. Our property specification language is an instance of first-order logic with formulas in negation normal form (i.e., only atomic formulas can be negated). Let $\mathcal{X}$ be a set of thread identifier variables disjoint from $\mathcal{V}$. The language consists of the following formulas over atomic predicates Q defined below. Explicit quantification is allowed only over thread identifier variables.

$$\phi ::= Q \mid \neg Q \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \forall \iota.\, \phi' \mid \exists \iota.\, \phi'$$

The atomic predicates and their informal meaning are as follows, where $\iota, \kappa \in \mathcal{X}$ are thread-id variables, $t, u \in \mathcal{M}$ are messages, $R \in \mathcal{V}_\alpha$ is a role name, and $e, e' \in Evt(\mathcal{M})$ are events.

$Q ::= \iota = \kappa$	thread ι and thread κ are equal
$\mid\ eq(\iota, \kappa, t, u)$	message t in thread ι's view equals message u in thread κ's view
$\mid\ secret(\iota, t)$	the intruder does not know message t as seen by thread ι
$\mid\ honest(\iota, R)$	the agent playing role R in thread ι's view is honest
$\mid\ role(\iota, R)$	thread ι executes role R
$\mid\ steps(\iota, e)$	thread ι has executed event e
$\mid\ (\iota, e) \prec (\kappa, e')$	thread ι has executed event e before thread κ has executed event e'

We use some syntactic sugar and write $t^{@\iota} = u^{@\kappa}$ for $eq(\iota, \kappa, t, u)$. An atomic predicate or negated atomic predicate is called *literal*. We say that an atomic predicate Q occurs *positively* (*negatively*) in a formula ϕ if there is a non-negated (negated) occurrence of Q in ϕ. To achieve attack preservation, we focus on the fragment of this logic where the predicate $secret(\iota, t)$ only occurs positively. We call this language $\mathcal{L}_P$. A *property* is formula of $\mathcal{L}_P$ where all thread-id variables appear in the scope of a quantifier. In examples, we freely use standard abbreviations (e.g., for implication) in formulas if there is an equivalent negation normal form in $\mathcal{L}_P$. We also write $honest(\iota, \{A_1, \ldots, A_n\})$ as an abbreviation for $\bigwedge_{k=1}^{n} honest(\iota, A_k)$.

Semantics. We define the semantics of our language $\mathcal{L}_P$. Recall that $\mathcal{A}_H$ denotes the set of honest agents. For a trace tr, we define a total ordering $\prec_{tr}$ over events occurring in tr such that $a \prec_{tr} b$ if $tr = tr_1 \cdot a \cdot tr_2 \cdot b \cdot tr_3$ for some tr_1, tr_2, and tr_3. The relation $\prec_{tr}$ is crucial to express security properties that impose strong ordering constraints between events such as *synchronization* [16] (see also Section 6.1).

Let $s = (tr, th, \sigma)$ be a state of the protocol P and let ϑ be a substitution interpreting thread-id variables from $\mathcal{X}$ as thread identifiers in $dom(th)$. Given an equational theory E, we define formula

satisfaction, $(s, \vartheta) \vDash_E \phi$, as follows:

$$
\begin{array}{ll}
(s,\vartheta) \vDash_E \iota = \kappa & \text{iff } \vartheta(\iota) = \vartheta(\kappa) \\
(s,\vartheta) \vDash_E t^{@\iota} = u^{@\kappa} & \text{iff } t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma \\
(s,\vartheta) \vDash_E \mathit{secret}(\iota, t) & \text{iff } \mathit{IK}(tr)\sigma \cup \mathit{IK}_0 \vdash_E t^{\vartheta(\iota)}\sigma \text{ is not derivable} \\
(s,\vartheta) \vDash_E \mathit{honest}(\iota, R) & \text{iff } R^{\vartheta(\iota)}\sigma \in \mathcal{A}_H \\
(s,\vartheta) \vDash_E \mathit{role}(\iota, R) & \text{iff } \pi_1(\mathit{th}(\vartheta(\iota))) = R \\
(s,\vartheta) \vDash_E \mathit{steps}(\iota, e) & \text{iff } (\vartheta(\iota), e) \in tr \\
(s,\vartheta) \vDash_E (\iota, e) \prec (\kappa, e') & \text{iff } (\vartheta(\iota), e) \prec_{tr} (\vartheta(\kappa), e') \\
(s,\vartheta) \vDash_E \neg A & \text{iff not } (s,\vartheta) \vDash_E A \\
(s,\vartheta) \vDash_E \phi_1 \wedge \phi_2 & \text{iff } (s,\vartheta) \vDash_E \phi_1 \text{ and } (s,\vartheta) \vDash_E \phi_2 \\
(s,\vartheta) \vDash_E \phi_1 \vee \phi_2 & \text{iff } (s,\vartheta) \vDash_E \phi_1 \text{ or } (s,\vartheta) \vDash_E \phi_2 \\
(s,\vartheta) \vDash_E \forall \iota.\, \phi' & \text{iff } (s,\vartheta[\iota \mapsto i]) \vDash_E \phi' \text{ for all } i \in \mathit{dom}(\mathit{th}) \\
(s,\vartheta) \vDash_E \exists \iota.\, \phi' & \text{iff } (s,\vartheta[\iota \mapsto i]) \vDash_E \phi' \text{ for some } i \in \mathit{dom}(\mathit{th})
\end{array}
$$

For properties ϕ, we write $s \vDash_E \phi$ instead of $(s, \vartheta) \vDash_E \phi$. A protocol P satisfies a property ϕ if $s \vDash_E \phi$ holds for all reachable states s of P. We write $s \nvDash_E \phi$ if $s \vDash_E \phi$ does not hold. We call a reachable state s of P an *attack on* ϕ if $s \nvDash_E \phi$.

In the following example, we present our formalizations of secrecy and authentication properties for the IKE_m protocol. Additional examples of properties are given in Section 6.1.

Example 3.11 (Properties of IKE_m)**.** We express the secrecy of the Diffie–Hellman key $\mathsf{exp}(Gb, x)$ for role A of the protocol IKE_m of Example 3.8 as follows.

$$\phi_{sec} = \forall \iota.\, (\mathit{role}(\iota, A) \wedge \mathit{honest}(\iota, \{A, B\}) \wedge \mathit{steps}(\iota, \mathsf{Secret})) \Rightarrow \mathit{secret}(\iota, \mathsf{exp}(Gb, x)).$$

Intuitively, ϕ_{sec} states that whenever an agent a playing role A completes his thread with another agent b playing role B and both a and b are honest, the key $\mathsf{exp}(Gb, x)$ is secret. In this protocol, the completion of the thread coincides with the presence of the Secret signal event in a trace.

We formalize non-injective agreement of A with B [32] on the nonces na and nb and the Diffie–Hellman half-keys $\mathsf{exp}(g, x)$ and $\mathsf{exp}(g, y)$ by

$$
\begin{aligned}
\phi_{auth} = \forall \iota.\, & (\mathit{role}(\iota, A) \wedge \mathit{honest}(\iota, \{A, B\}) \wedge \mathit{steps}(\iota, \mathsf{Commit})) \\
& \Rightarrow (\exists \kappa.\, \mathit{role}(\kappa, B) \wedge \mathit{steps}(\kappa, \mathsf{Running}) \wedge \\
& \qquad \langle A, B, na, Nb, \mathsf{exp}(g, x), Gb\rangle^{@\iota} = \langle A, B, Na, nb, Ga, \mathsf{exp}(g, y)\rangle^{@\kappa}).
\end{aligned}
$$

The formula ϕ_{auth} states that whenever an agent a playing role A completes his thread with another agent b playing role B and both agents are honest, then b has previously been running the protocol with a. Moreover, a and b agree on na and nb and the Diffie–Hellman half-keys $\mathsf{exp}(g, x)$ and $\mathsf{exp}(g, y)$. The authentication on these values is formulated by the equality in the formula, which also includes the agreement on the participating agents and their roles.

Note that our property language does not allow expressing the general notion of injective agreements as defined by Lowe [32], which amounts to counting the numbers of Commit and Running signals occurring in the trace. However, we can express a stronger version of injective agreement as an agreement where there is at most one Commit signals for a given message to be agreed on. This trivially implies

the injectiveness of the agreement. This property is suitable for protocols where the role emitting the Commit signal contributes a fresh value to the message to be agreed on, in which case the two definitions coincide. For instance, we formalize the injective agreement of role A with role B on the Diffie–Hellman half-keys $\mathsf{exp}(g, x)$ and $\mathsf{exp}(g, y)$ by

$$\begin{aligned}\phi_{iauth} = \forall \iota.\, (&\mathit{role}(\iota, A) \wedge \mathit{honest}(\iota, \{A, B\}) \wedge \mathit{steps}(\iota, \mathsf{Commit})) \\ \Rightarrow\, (&\exists \kappa.\, \mathit{role}(\kappa, B) \wedge \mathit{steps}(\kappa, \mathsf{Running}) \wedge \\ &\langle A, B, \mathsf{exp}(g, x), Gb\rangle^{@\iota} = \langle A, B, Ga, \mathsf{exp}(g, y)\rangle^{@\kappa}) \wedge \\ &(\forall \lambda.\, (\mathit{role}(\lambda, A) \wedge \mathit{steps}(\lambda, \mathsf{Commit}) \wedge \\ &\langle A, B, \mathsf{exp}(g, x), Gb\rangle^{@\lambda} = \langle A, B, \mathsf{exp}(g, x), Gb\rangle^{@\iota}) \Rightarrow \lambda = \iota)\end{aligned}$$

Remark 3.12. An alternative formulation of our protocol semantics and property language, suggested by one of the reviewers, is obtained by viewing each variable and fresh value as an unary function symbol and keeping the thread identifier variables as the only variables of the property language. The set of network messages would thus become $\mathcal{N}^{\mathrm{alt}} = \mathcal{T}_{\Sigma \cup \mathcal{V}_P \cup \mathcal{F}_P}(\mathcal{X}, \mathcal{A} \cup \mathcal{C} \cup \mathcal{F}^{\bullet} \cup \mathit{TID})$. We briefly discuss how such a setup could look like and how it compares to ours.

The substitutions $\sigma : \mathcal{V}^{\mathit{TID}} \rightharpoonup \mathcal{N}$ in the states would be replaced by first-order structures $\sigma : \mathcal{V} \to (\mathit{TID} \rightharpoonup \mathcal{N}^{\mathrm{alt}})$ interpreting the function symbols associated with the protocol variables as type-respecting functions mapping thread identifiers to network messages. We would leave the function symbols in $\mathcal{F}_P$ uninterpreted as a simple way to model the uniqueness of fresh values. More precisely, the interpretation $\|t\|_{(\sigma,\vartheta)}$ of a network message t would be $\|V(\iota)\|_{(\sigma,\vartheta)} = \sigma(V)(\vartheta(\iota))$ for $V \in \mathcal{V}_P$, $\|n(\iota)\|_{(\sigma,\vartheta)} = n(\vartheta(\iota))$ for $n \in \mathcal{F}_P$ and extended homomorphically to all terms. Note that this interpretation is isomorphic to ours if we use thread variables and identifiers in $\mathcal{N}^{\mathrm{alt}}$ only as arguments of the function symbols in $\mathcal{V}_P$ and $\mathcal{F}_P$.

We see two possibilities for dealing with protocol specifications in such an approach. The first possibility is to keep protocol specifications unchanged, i.e., using messages from $\mathcal{M}$, but replace the indexing of variables and fresh values in network messages by function application, i.e., we would have $V^{\iota} = V(\iota)$ and $n^{\iota} = n(\iota)$ for variables and fresh values and extend this to all terms as expected. One could keep the syntax of the property language, but would adapt its interpretation. For example, term equations would still be written $t^{@\iota} = u^{@\kappa}$ for $t, u \in \mathcal{M}$, but the semantics would become $\|t^{\iota}\|_{(\sigma,\vartheta)} =_E \|u^{\kappa}\|_{(\sigma,\vartheta)}$. While remaining very close to our formulation, the disadvantage of this approach is the non-uniform treatment of messages in specifications (with variables and substitutions as before) and network messages (with the new interpretation). This would complicate the development of our abstraction theory, as it applies abstractions to both protocol messages in specifications and network messages in traces.

The second possibility is to also use messages from $\mathcal{N}^{\mathrm{alt}}$ in protocol specifications. In this approach, every protocol role would be parametrized by a thread-id variable ι, which is used as an argument of all function symbols in $\mathcal{V}_P$ and $\mathcal{F}_P$ in the role. This variable would be instantiated with some $i \in \mathit{TID}$ to create the actual thread i (cf. definition of initial state). Indexing would no longer be needed. We could adapt the property language accordingly. For example, term equations would be written as $t = u$ for $t, u \in \mathcal{N}^{\mathrm{alt}}$ and interpreted as $\|t\|_{(\sigma,\vartheta)} =_E \|u\|_{(\sigma,\vartheta)}$. This would yield a more uniform picture again at the price of cluttering all variables and fresh values in protocol specifications with thread-id variables.

In both cases, the operational semantics and numerous details would have to be carefully adapted. We believe that our setup strikes a good balance between an economic notation for protocol specifications and a uniform treatment of different kinds of messages in our abstraction theory.

4. Security protocols abstractions

In this section, we present two kinds of protocol abstractions:

Typed abstractions transform a term's structure by removing or reordering fields and by removing or splitting cryptographic operations. The types enable a fine-grained selection of the transformation to apply. The same transformation is applied to all terms of a given type and its subtypes.

Untyped abstractions complement typed ones with two additional kinds of simplifications: atom/variable removal abstractions and redundancy removal abstractions. The former remove unprotected atoms or variables while the latter remove terms that the intruder can derive.

Typically, we will use typed abstractions to simplify the cryptographic structure of terms followed by untyped abstractions to remove atoms and variables as well as redundancies.

In Section 4.1, we give an overview of the different kinds of abstractions and their combined use. We then proceed with the formal definitions and results for our protocol abstractions that we will apply in the following chapters. Our main results are soundness theorems for the typed and untyped abstractions. They ensure that any attack on a given property of the original protocol translates to an attack on the abstracted protocol. Similar to [28], we follow a modular approach for proving this property. We first define a general notion of protocol abstraction for which we prove a general soundness theorem under certain conditions (Section 4.2). These conditions concern the preservation of intruder deducibility as well as of equalities and disequalities. We then go on to define each concrete kind of abstraction and prove its soundness (Sections 4.3–4.5). We illustrate the usefulness of our definitions on our running example. For the soundness proofs it then suffices to establish the conditions of the general soundness theorem. As we will see, each such soundness result in turn imposes certain conditions, which we will introduce and motivate by examples.

Upon first reading, readers may choose to skip the remainder of this section after reading the following overview and proceed to the next sections to get an impression of how we will use the abstractions.

4.1. Overview

Typed abstractions are our main mechanism to simplify the cryptographic structure of terms by removing protections that are not required to achieve a given property. We specify typed abstractions by a list of recursive equations. The following example illustrates a range of typical forms of defining equations. Messages are transformed according to the first matching pattern. If no pattern matches then the top-level symbol is transformed homomorphically. Typed abstractions leave atoms and variables untouched.

Example 4.1 (Typed abstractions)**.** Consider a simplified variant of the $\mathsf{IKE}_{\mathsf{m}}$ protocol from Section 2 and Example 3.8, where in the first two messages each role sends the constant *sA1*, its Diffie–Hellman half-key, and a nonce and we authenticate the final two messages using signatures instead of MACs. We focus here on the final two events of the initiator A:

$$\mathsf{send}(\{\!|A, B, sA2, [m3, sA1, na, Nb, \mathsf{exp}(g, x), SKa]_{\mathsf{pri}(A)}|\!\}_{SKa}) \cdot$$
$$\mathsf{recv}(\{\!|B, sA2, [m4, sA1, na, Nb, Gb, SKa]_{\mathsf{pri}(B)}|\!\}_{SKa})$$

where $SKa = \mathsf{kdf}(\mathsf{exp}(Gb, x), na, Nb)$ and *m3* and *m4* are tagging constants distinguishing the two messages. Suppose our goal is to verify that the initiator non-injectively agrees with the responder on

na, $\mathsf{exp}(g, x)$, and Gb. For this purpose, we aim at simplifying these events as follows:

$$\mathsf{send}([m3, \mathsf{exp}(g, x), \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(A)}) \cdot$$
$$\mathsf{recv}([m4, na, Gb, \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)})$$

Note that we drop na and Nb from the third message and Nb from the fourth. To achieve this, we first specify a typed abstraction using the following four equations:

$$f(\{\!|X|\!\}_Y) = f(X)$$
$$f(\mathsf{kdf}(X, Y)) = \mathsf{kdf}(f(X))$$
$$f([T_3, S, N_1, N_2, Y]_{\mathsf{pri}(Z)}) = \langle [f(T_3), f(Y)]_{\mathsf{pri}(f(Z))}, f(S), f(N_1), f(N_2)\rangle$$
$$f([T_4, S, N_1, N_2, Y]_{\mathsf{pri}(Z)}) = \langle [f(T_4), f(N_1), f(Y)]_{\mathsf{pri}(f(Z))}, f(S), f(N_2)\rangle$$

where all variables have type msg except for $T_3 : \gamma_{m3}$ and $T_4 : \gamma_{m4}$. The equation for symmetric encryption simply drops the encryption and the one for the key derivation function kdf drops the second component of the pair underneath it. There are also two equations for signatures. They both pull the tuple components S and N_2 out of the signature. The first equation additionally pulls out N_1. Note that, when transforming the protocol's events, the variable Y will match the pair of messages consisting of the Diffie–Hellman half-key and the session key. Note also that the patterns on the left-hand side of these two equations are identical except for the types of the variables T_3 and T_4, which respectively match the tags $m3$ and $m4$. Only the types allow us to distinguish the third and the fourth protocol messages and to transform them in different ways. Sound typed abstractions cannot remove arbitrary fields: while the equation for kdf removes Y, those for the signatures pull some tuple components out of the signature instead of removing them (as we would like to).

Let us now apply this typed abstraction to the fourth message in A's role. We elide the application of f to atoms and variables (where f is the identity) and on pairs, pri and exp (where f behaves homomorphically).

$$\begin{aligned} & f(\{\!|B, sA2, [m4, sA1, na, Nb, Gb, SKa]_{\mathsf{pri}(B)}|\!\}_{SKa}) \\ &\quad = \langle B, sA2, f([m4, sA1, na, Nb, Gb, SKa]_{\mathsf{pri}(B)})\rangle \\ &\quad = \langle B, sA2, [m4, na, Gb, f(SKa)]_{\mathsf{pri}(B)}, sA1, Nb\rangle \\ &\quad = \langle B, sA2, [m4, na, Gb, f(\mathsf{kdf}(\mathsf{exp}(Gb, x), na, Nb))]_{\mathsf{pri}(B)}, sA1, Nb\rangle \\ &\quad = \langle B, sA2, [m4, na, Gb, \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)}, sA1, Nb\rangle \end{aligned}$$

The third message abstracts to $\langle A, B, sA2, [m3, \mathsf{exp}(g, x), \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)}, sA1, na, Nb\rangle$.

Generally speaking, in order to preserve the deducibility of messages, typed abstractions cannot remove fields that are extractable (e.g., by projection, decryption, or signature verification), whereas removing the non-extractable fields under a hash-type function such as kdf poses no problems. We use untyped abstractions to remove redundant or unprotected message elements including those we have pulled out of cryptographic operations using typed abstractions.

Example 4.2 (Atom-and-variable removal)**.** Applying the typed abstraction above to the fourth protocol message yielded $t = \langle B, sA2, [m4, na, Gb, \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)}, sA1, Nb\rangle$. To obtain the desired result $t' = [m4, na, Gb, \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)}$, we want to remove the fields B, $sA1$, $sA2$, and Nb. The atom-and-variable removal abstraction rem_T is parametrized by a set T of atoms and variables and removes all cryptographically unprotected occurrences of the elements of T from a message (i.e., those visible within t without any decrypting). Soundness requires that the transformed messages must not contain any protected occurrence of the elements of T. In our case, we set $T = \{A, B, sA1, sA2, nb, Nb\}$ to obtain $rem_T(t) = t'$ and we observe that the soundness condition is satisfied for this choice. Applying rem_T to the abstracted third protocol message yields $\langle [m3, \mathsf{exp}(g, x), \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)}, na\rangle$. The soundness condition forbids the inclusion of the nonce na in T, since na also occurs protected by the signature in t'.

Example 4.3 (Redundancy removal)**.** Since na is sent in the clear along with the Diffie–Helman half-key in the first protocol message, we use a redundancy removal abstraction to remove the redundant occurrence of na in the abstracted third message. Redundancy removal abstractions are functions on messages that return a special value nil for removed messages. They can remove message elements from a role that the intruder can deduce from his initial knowledge or from elements that he has learned earlier from the same role. Since na and Na already occur in the first message of the initiator or responder roles, this condition holds for the function that removes na from $\langle [m3, Gb, \mathsf{kdf}(\mathsf{exp}(Gb, x))]_{\mathsf{pri}(B)}, na\rangle$ and Na from role B's third message while leaving all other messages unchanged. As an alternative to the atom-and-variable removal in Example 4.2, we could also remove the elements of T using a redundancy removal abstraction that removes all occurrences of A, B, $sA1$, and $sA2$ (we assume the intruder knows all agents and constants) and all but the first occurrences of nb and Nb (similar to what we did with na and Na above).

We have chosen to factor out the removal of atoms and variables as well as redundancies from the typed abstractions, since this substantially simplifies their definition and soundness proofs.

4.2. General soundness theorem for protocol abstractions

We start by defining a general form of protocol abstraction that encompasses all of our concrete abstractions. We then prove a general soundness theorem for these abstractions, which we later instantiate to obtain concrete soundness results.

4.2.1. General protocol abstractions

A general protocol abstraction consists of two functions. The first functions transforms the terms in the protocol definition and in protocol executions, while the second one transforms properties. For some but not all concrete abstractions these functions will coincide. We introduce the set $\mathcal{T} = \mathcal{M} \cup \mathcal{N}$, which includes all terms that may occur in protocol specifications, properties, symbolic traces, or ground traces. In the definition below, we use the special symbol nil to mark messages that are removed.

Definition 4.4 (General protocol abstraction)**.** A *(general) protocol abstraction* is a pair $\mathcal{G} = (g_{prot}, g_{prop})$ where $g_{prot} : \mathcal{T} \to \mathcal{T} \cup \{\mathsf{nil}\}$ and $g_{prop} : \mathcal{T} \to \mathcal{T} \cup \{\mathsf{nil}\}$. We define the application of $\mathcal{G}$ to events, traces, and protocols by applying the appropriate component of $\mathcal{G}$ to the terms they contain as follows.

(i) For events: $\mathcal{G}(sig) = sig$ for $sig \in Sig$ and, for $ev \in \{\mathsf{send}, \mathsf{recv}\}$, $\mathcal{G}(ev(t)) = \mathsf{nil}$ if $\mathcal{G}(t) = \mathsf{nil}$ and $\mathcal{G}(ev(t)) = ev(g_{prot}(t))$ otherwise.

(ii) For event sequences: $\mathcal{G}(\epsilon) = \epsilon$ and $\mathcal{G}(e \cdot tl) = \mathcal{G}(tl)$ if $\mathcal{G}(e) = \mathsf{nil}$ and $\mathcal{G}(e \cdot tl) = \mathcal{G}(e) \cdot \mathcal{G}(tl)$ otherwise; this is extended to traces and threads in the expected way.
(iii) $\mathcal{G}(P) = \{(R, \mathcal{G}(P(R))) \mid R \in dom(P) \wedge \mathcal{G}(P(R)) \neq \epsilon\}$ for protocols P.
(iv) For the atomic predicates of our property language:

$$\mathcal{G}(\iota = \kappa) = (\iota = \kappa) \qquad \mathcal{G}(role(\iota, A)) = role(\iota, A)$$
$$\mathcal{G}(t^{@\iota} = u^{@\kappa}) = (g_{prop}(t)^{@\iota} = g_{prop}(u)^{@\kappa}) \qquad \mathcal{G}(steps(\iota, e)) = steps(\iota, \mathcal{G}(e))$$
$$\mathcal{G}(secret(\iota, t)) = secret(\iota, g_{prop}(t)) \qquad \mathcal{G}((\iota, e) \prec (\kappa, e')) = (\iota, \mathcal{G}(e)) \prec (\kappa, \mathcal{G}(e'))$$
$$\mathcal{G}(honest(\iota, A)) = honest(\iota, A)$$

We extend this mapping homomorphically to all formulas. Note that the terms in a formula's events are abstracted by g_{prot}, while those in equations and secrecy predicates are abstracted using g_{prop}.

Although general protocol abstractions have two independent fields, our concrete typed and untyped abstractions will use only special forms. For typed abstractions and atom-variable removal abstraction, we will have $g_{prot} = g_{prop}$ and for redundancy removal abstractions $g_{prop} = id$ (the identity function).

4.2.2. Soundness of general protocol abstractions

To justify the soundness of our abstractions $\mathcal{G}$, we show that any attack on a property ϕ of the original protocol P is reflected as an attack on the property $\mathcal{G}(\phi)$ of the abstracted protocol $\mathcal{G}(P)$. We decompose this into reachability preservation (RP) and attack preservation (AP) as follows. We require that, for all reachable states (tr, th, σ) of P, there is a substitution σ' such that

(RP) $(\mathcal{G}(tr), \mathcal{G}(th), \sigma')$ is a reachable state of $\mathcal{G}(P)$, and
(AP) $(tr, th, \sigma) \not\models \phi$ implies $(\mathcal{G}(tr), \mathcal{G}(th), \sigma') \not\models \mathcal{G}(\phi)$.

We will define the substitution σ' as $g(\sigma) = g \circ \sigma$ for some function $g : \mathcal{N} \to \mathcal{N}$ on network messages. These two properties will require some assumptions about P, ϕ, and $\mathcal{G}$. We start by defining and explaining the conditions on formulas. We first introduce some auxiliary sets of elements of a formula ϕ:

– Sec_ϕ be the set of all terms t that occur in formulas $secret(\iota, t)$ in ϕ,
– Eq_ϕ be the set of tuples (ι, κ, t, u) such that the equation $t^{@\iota} = u^{@\kappa}$ occurs in ϕ and let $EqTerm_\phi = \{t, u \mid \exists \iota, \kappa.\, (\iota, \kappa, t, u) \in Eq_\phi\}$ be the set of underlying terms, and
– Evt_ϕ be the set of events occurring in ϕ.

Let Eq_ϕ^+ and Eq_ϕ^- respectively be the sets of tuples representing equations with a positive and a negative occurrence in ϕ and let $EqTerm_\phi^+$ and $EqTerm_\phi^-$ be the corresponding sets of terms. Similarly, we define the subset Evt_ϕ^+ of elements of Evt_ϕ with a positive occurrence in ϕ.

Definition 4.5 (Safe formulas)**.** Let $g : \mathcal{N} \to \mathcal{N}$ be a function on network messages. We define ϕ to be *safe* for P and $(\mathcal{G}, g)$ if, for all well-typed ground substitutions σ, the following conditions hold:

(a) $\mathsf{nil} \notin \mathcal{G}(Sec_\phi \cup EqTerm_\phi \cup Evt_\phi)$,
(b) g is the identity function on $\mathcal{A}$,
(c) for all $(\iota, \kappa, t, u) \in Eq_\phi^-$ and thread-id interpretations ϑ, we have that

$$t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma \text{ implies } g_{prop}(t^{\vartheta(\iota)})g(\sigma) =_E g_{prop}(u^{\vartheta(\kappa)})g(\sigma),$$

(d) for all $(\iota, \kappa, t, u) \in Eq_\phi^+$ and thread-id interpretations ϑ, we have that

$$g_{prop}(t^{\vartheta(\iota)})g(\sigma) =_E g_{prop}(u^{\vartheta(\kappa)})g(\sigma) \text{ implies } t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma,$$

(e) for all $e(t) \in Evt_\phi^+$ and $e(u) \in Evt(\mathcal{M}_P)$, we have $g_{prot}(t) = g_{prot}(u)$ implies $t = u$.

Condition (a) ensures that nil does not occur in the abstracted formula. Condition (b) ensures that the two substitutions agree on agent variables. Condition (c) requires equality preservation for negatively occurring equations. Condition (d) expresses the injectivity of the abstraction on the terms in positively occurring equalities. This condition is required to preserve attacks on agreement properties. In other words, it prevents abstractions from fixing attacks on agreement by identifying two terms that differ in the original protocol. Finally, condition (e) is required for properties involving event orderings and *steps* predicates. It states that the abstraction must not identify an event occurring positively in the property with a distinct protocol event.

We now state the soundness theorem for the general abstractions.

Theorem 4.6 (General soundness theorem)**.** *Let P be a protocol, ϕ a property, $\mathcal{G} = (g_{prot}, g_{prop})$ a protocol abstraction, and g a function on network messages. Suppose the following conditions hold:*

(i) *For all states $(tr, th, \sigma) \in reach(P, IK_0)$, thread id's i, agent variables R, role suffixes tl, and terms t such that $th(i) = (R, \mathsf{recv}(t) \cdot tl)$ and $g_{prot}(t) \neq \mathsf{nil}$, we have*

$$IK(tr)\sigma, IK_0 \vdash_E t^i\sigma \text{ implies } IK(\mathcal{G}(tr))g(\sigma), IK_0' \vdash_E g_{prot}(t^i)g(\sigma),$$

(ii) *For all states $(tr, th, \sigma) \in reach(P, IK_0)$, thread id's i, and terms $t \in Sec_\phi$ such that $g_{prop}(t) \neq \mathsf{nil}$ we have*

$$IK(tr)\sigma, IK_0 \vdash_E t^i\sigma \text{ implies } IK(\mathcal{G}(tr))g(\sigma), IK_0' \vdash_E g_{prop}(t^i)g(\sigma), \text{ and}$$

(iii) *ϕ is safe for P and $(\mathcal{G}, g)$.*

Then for all states $(tr, th, \sigma) \in reach(P, IK_0)$ we have

1. $(\mathcal{G}(tr), \mathcal{G}(th), g(\sigma)) \in reach(\mathcal{G}(P), IK_0')$, *and*
2. $(tr, th, \sigma) \not\models \phi$ *implies* $(\mathcal{G}(tr), \mathcal{G}(th), g(\sigma)) \not\models \mathcal{G}(\phi)$.

Condition (i) ensures that derivability is preserved for received messages. Similarly, condition (ii) ensures the deducibility preservation for claimed secrets. Condition (i) is needed to establish conclusion 1 and conditions (ii) and (iii) are required for conclusion 2. Below we sketch the proof of this theorem. The full proof can be found in the full version [39].

Proof Sketch. To show point 1 (reachability preservation), let $(tr, th, \sigma) \in reach(P, IK_0)$. We establish $(\mathcal{G}(t), \mathcal{G}(th), g(\sigma)) \in reach(\mathcal{G}(P), IK_0')$ by induction on the number n of transitions leading to the state (tr, th, σ). The base case ($n = 0$) is straightforward. For the inductive case, assume (tr', th', σ) is reachable in k steps and there is a transition $(tr', th', \sigma) \to (tr, th, \sigma)$. By the induction hypothesis, we know that $(\mathcal{G}(tr'), \mathcal{G}(th'), g(\sigma)) \in reach(\mathcal{G}(P), IK_0')$. If $g_{prot}(t) = \mathsf{nil}$ then we have $\mathcal{G}(tr) = \mathcal{G}(tr')$ and $\mathcal{G}(th) = \mathcal{G}(th')$ and hence $(\mathcal{G}(tr), \mathcal{G}(th), g(\sigma)) \in reach(\mathcal{G}(P), IK_0')$. Otherwise, we have $g_{prot}(t) \neq \mathsf{nil}$.

We consider three cases according to the rule r that has been applied in step $k+1$. The cases for the rules *SEND* and *SIGNAL* are straightforward. For the remaining case $r = RECV$, we know by the rule's premises that $th'(i) = (R, \mathsf{recv}(t) \cdot tl)$ and $IK(tr')\sigma, IK_0 \vdash_E t^i\sigma$ for some r, t, and tl. Using assumption (i) with the induction hypothesis, we establish the two premises of rule *RECV* required to obtain $(\mathcal{G}(tr'), \mathcal{G}(th'), g(\sigma)) \to (\mathcal{G}(tr), \mathcal{G}(th), g(\sigma))$. This implies the conclusion for this case. Hence, we have established point 1.

To show point 2 (attack preservation), we proceed by induction on the structure of ϕ and use assumptions (ii) and (iii). □

In the following subsections, we discuss each kind of protocol abstraction and the associated soundness result. For these proofs, it suffices to define the function g and to establish the conditions (i)–(iii) of the theorem above. In each case we introduce the assumptions that are needed for this purpose and motivate them by examples.

4.3. Typed protocol abstractions

Our typed abstractions are specified by a list of recursive equations subject to some conditions on their shape. We define their semantics in terms of a simple Haskell-style functional program. We use both pattern matching on terms and subtyping on types to select the equation to be applied to a given term. This ensures that terms of related types are transformed in a uniform manner.

4.3.1. Syntax and semantics

Let $\mathcal{W} = \{\mathcal{W}_\tau\}_{\tau \in \mathcal{Y}}$ be a family of *pattern variables* disjoint from $\mathcal{V}$. We define the set of *patterns* by $\mathcal{P} = \mathcal{T}(\mathcal{W}, \emptyset)$. A pattern $p \in \mathcal{P}$ is called *linear* if each (pattern) variable occurs at most once in p. We extend the typing function Γ to patterns by setting $\Gamma(X) = \tau$ if and only if $X \in \mathcal{W}_\tau$ and then lifting it homomorphically to all patterns. Our typed message abstractions are instances of the following recursive function specifications.

Definition 4.7. A *function specification* $F_f = (f, E_f)$ consists of an unary function symbol $f \notin \Sigma^1$ and a list of equations

$$E_f = [f(p_1) = u_1, \ldots, f(p_n) = u_n],$$

where each $p_i \in \mathcal{P}$ is a linear pattern such that $u_i \in \mathcal{T}_{\Sigma^{\geqslant 1} \cup \{f\}}(vars(p_i))$ for all $i \in \widetilde{n}$, i.e., u_i consists of variables from p_i and function symbols from $\Sigma^{\geqslant 1} \cup \{f\}$.

Definition 4.8. For $c \in \Sigma^{\geqslant 1}$, an equation $f(c(p_1, \ldots, p_n)) = u$ of E_f is called a c-equation and it is called *homomorphic* if $u = c(f(Z_1), \ldots, f(Z_n))$ and $p_i = Z_i$ are variables of type *msg*. We say that F_f is *homomorphic for* $c \in \Sigma^{\geqslant 1}$ if all c-equations in E_f are homomorphic.

The function specification $F_f^0 = (f, E_f^0)$ consists of a homomorphic equation for each $c \in \Sigma^{\geqslant 1}$ and the final equation $f(Z) = Z$ with $Z : msg$.

We use vectors (lists) of terms $\overline{t} = [t_1, \ldots, t_n]$ for $n > 0$. We define $set(\overline{t}) = \{t_1, \ldots, t_n\}$ and $\widehat{f}(\overline{t}) = \langle f(t_1), \ldots, f(t_n)\rangle$, the elementwise application of a function f to a vector where the result is converted to a tuple (with the convention $\langle t\rangle = t$). We define the *splitting* function by $split(\langle t, u\rangle) = split(t) \cup split(u)$ on pairs and $split(t) = \{t\}$ on other terms t. We call the elements of $split(t)$ the *fields* of t. We extend *split* to vectors by $split(\overline{t}) = split(set(\overline{t}))$.

Definition 4.9 (Typed abstraction)**.** A *typed abstraction* is a function specification of the form $F_f = (f, E_f^+)$ where $E_f^+ = E_f \cdot E_f^0$ and each equation in E_f has the form

$$f(c(p_1, \ldots, p_n)) = \langle e_1, \ldots, e_d \rangle \qquad (\star)$$

where for each $i \in \widetilde{d}$ we have either

(a) $e_i = f(q)$ such that $q \in \mathit{split}(p_j)$ for some $j \in \widetilde{n}$, or
(b) $e_i = c(\widehat{f}(\overline{q_1}), \ldots, \widehat{f}(\overline{q_n}))$ with $c \neq \langle \cdot, \cdot \rangle$ such that, for all $j \in \widetilde{n}$, we have $\mathit{set}(\overline{q_j}) \subseteq \mathit{split}(p_j)$ and, whenever p_i is not a pair, we have $\overline{q_i} = [p_i]$, i.e., $\widehat{f}(\overline{q_i}) = f(p_i)$.

The concatenation of E_f with E_f^0 ensures the totality of typed abstractions. The shape of the terms e_i in equation ($\star$) ensures that the abstractions can only weaken the cryptographic protection of terms but never strengthen it. Each defining equation maps a term with top-level symbol c to a tuple whose components have the form (a) or (b). In both forms, we can only apply f recursively on fields of the patterns p_i. Form (a) allows us to pull fields out of the scope of c, hence removing c's protection. Using form (b) we can reorder, duplicate, or remove fields in each argument of c. We cannot however turn a non-pair argument of c into a pair such as in $f(c(x)) = c(\langle f(x), f(x) \rangle)$. Furthermore, for the case where c is pairing we have to use form (a) to obtain the simple shape $f(\langle p_1, p_2 \rangle) = \widehat{f}(\overline{q})$ with $\mathit{set}(\overline{q}) \subseteq \mathit{split}(\langle p_1, p_2 \rangle)$.

Example 4.10. We present a typed abstraction $F_f = (f, E_f \cdot E_f^0)$ illustrating a representative selection of the possible message transformations. Suppose $X : \gamma_c$, $Y : \mathit{nonce}$, and $Z, U, V : \mathit{msg}$ and let E_f consist of the following three equations:

$$f(\mathsf{kdf}(X, U, V)) = \mathsf{kdf}(f(X), f(U))$$
$$f([Y, Z]_{\mathsf{pri}(U)}) = \langle f(Y), f(Z) \rangle$$
$$f(\{\!|X, Y, Z|\!\}_U) = \langle \{\!|f(X), f(Z)|\!\}_{f(U)}, f(Y) \rangle$$

The patterns' types filter the matching terms: X and Y only match the constant c respectively a nonce. The first equation removes the field V from a kdf hash. The second equation removes the signature. The third one pulls the field Y out of an encryption. These are typical examples of typed abstractions that are generated by our abstraction heuristics described in Section 5. Our theory also supports other forms of typed abstractions such as the following two:

$$f(\langle X, Y, Z \rangle) = \langle f(Y), f(X), f(Z) \rangle$$
$$f(\{\!|X, Y, Z|\!\}_U) = \langle \{\!|f(X), f(Y)|\!\}_{f(U)}, \{\!|f(Z)|\!\}_{f(U)} \rangle$$

The first equation swaps the first two fields in n-tuples for $n \geqslant 3$. In practice, such a re-ordering abstraction is useful to avoid type confusions, which may lead to spurious attacks. The second one splits an encryption: the pair $\langle f(X), f(Y) \rangle$ and $f(Z)$ are encrypted separately with the key $f(U)$.

The semantics of a typed abstraction F_f is given by the Haskell-style functional program f (Program 1). We are overloading the symbol f here: we use it as a function symbol in E_f^+ as well as the name of the functional program constructed from the equations in E_f^+. The **case** statement has a clause

$$p \mid \Gamma(t) \preccurlyeq \Gamma(p) \Rightarrow u$$

fun $f(t)$ = **case** t **of**
$\quad \| \; p_1 \mid \Gamma(t) \preccurlyeq \Gamma(p_1) \Rightarrow u_1$
$\quad \vdots$
$\quad \| \; p_k \mid \Gamma(t) \preccurlyeq \Gamma(p_k) \Rightarrow u_k$

Program 1. Program f resulting from $F_f = (f, E_f)$, where $[f(p_1) = u_1, \ldots, f(p_k) = u_k] = E_f^+$.

for each equation $f(p) = u$ of E_f^+. Note that occurrences of f in u correspond to recursive calls of the program f. Such a clause is enabled if

(1) the term t matches the pattern p, i.e., $t = p\theta$ for some substitution θ, and
(2) its type $\Gamma(t)$ is a subtype of $\Gamma(p)$.

The first enabled clause is executed. Hence, the equations E_f^0 serve as fall-back clauses, which cover the terms not handled by E_f. In particular, the last clause $f(Z) = Z$ handles exactly the atoms and variables.

We will often identify the typed abstraction $F_f = (f, E_f)$ and the functional program f. The corresponding general protocol abstraction according to Definition 4.4 is then simply $\mathcal{G} = (f, f)$.

Example 4.11. Consider the typed abstraction given by the first three equations from Example 4.10, including the types of the variables. Suppose we would like to use the associated program f (as specified in Program 1) to abstract the term $t = \{\!|c, n, W|\!\}_{\mathsf{kdf}(c,k,A)}$, which is composed of the constant $c : \gamma_c$, the nonces $n, k : \mathit{nonce}$, the message variable $W : \mathit{msg}$, and the agent variable $A : \alpha$. The resulting reduction sequence and the corresponding subtyping conditions are as follows:

$$
\begin{aligned}
f(t) &= f(\{\!|c, n, W|\!\}_{\mathsf{kdf}(c,k,A)}) && \{\!|\gamma_c, \beta_n, \mathit{msg}|\!\}_{\mathsf{kdf}(\gamma_c,\beta_k,\alpha)} \preccurlyeq \{\!|\gamma_c, \mathit{nonce}, \mathit{msg}|\!\}_{\mathit{msg}} \\
&= \langle \{\!|c, W|\!\}_{f(\mathsf{kdf}(c,k,A))}, n \rangle && \mathsf{kdf}(\gamma_c, \beta_n, \mathit{msg}) \preccurlyeq \mathsf{kdf}(\gamma_c, \mathit{msg}, \mathit{msg}) \\
&= \langle \{\!|c, W|\!\}_{\mathsf{kdf}(c,k)}, n \rangle
\end{aligned}
$$

Note that we have elided the reduction steps for pairs and for atomic messages, which use the corresponding fallback equations in E_f^0. Both subtyping conditions clearly hold. However, for the slightly different term $u = \{\!|n, c, W|\!\}_{\mathsf{kdf}(d,k,A)}$ for $d : \gamma_d$ we obtain $f(u) = u$, since in this case the two corresponding subtyping conditions do not hold. Therefore, only the homomorphic fallback equations in E_f^0 apply, which have trivial subtyping conditions.

4.3.2. Finding abstractions

Finding abstractions is fully automated by our tool using a heuristic that we will describe in Section 5. To show a concrete application of typed abstractions to our running example while giving a first idea of our heuristics, we use here the following simplified abstraction strategy: We start by identifying the terms that appear in the $\mathit{secret}(\cdot, \cdot)$ predicates and equations of the desired properties. Then we determine the cryptographic operations that are essential to achieve these properties and try to remove all other terms and operations. In this process, we have to be careful not to over-abstract the protocol, since this may easily introduce false negatives (i.e., spurious attacks). Therefore, apart from preserving the necessary cryptographic operations, we also avoid the introduction of new pairs of unifiable protocol terms.

Example 4.12 (from $\mathsf{IKE_m}$ to $\mathsf{IKE_m^1}$)**.** To preserve the secrecy of the DH key $\mathsf{exp}(\mathsf{exp}(g, x), y)$ and the agreement on *na*, *nb*, $\mathsf{exp}(g, x)$, and $\mathsf{exp}(g, y)$, we have to keep either the mac or the symmetric encryption with *SK* (see Examples 3.8 and 3.11). We want to remove as many other fields and operations as possible (e.g., prf). We choose to remove the encryption as this allows us to later remove additional fields (e.g., *sA2*) using untyped abstractions. We keep *o* in *AUTHa* to prevent unifiability with *AUTHb* and hence potential false negatives. This leads us to the typed abstraction $F_{f_1} = (f_1, E_{f_1})$ where E_{f_1} is defined by the equations

$$
\begin{aligned}
&f_1(\{\!|X, Y|\!\}_Z) = \langle f_1(X), f_1(Y)\rangle \qquad X : \alpha\\
&f_1(\mathsf{mac}(X_1, \ldots, X_8)) = \mathsf{mac}(\widehat{f_1}([X_1, X_3, X_5, X_6, X_7, X_8])) \qquad X_3 : \gamma_o\\
&f_1(\mathsf{mac}(Y_1, \ldots, Y_8)) = \mathsf{mac}(\widehat{f_1}([Y_1, Y_5, Y_6, Y_7, Y_8])) \qquad Y_3 : \mathit{nonce}\\
&f_1(\mathsf{kdf}(Z_1, \ldots, Z_5)) = \mathsf{kdf}(f_1(Z_3))\\
&f_1(\mathsf{prf}(U, Z)) = f_1(U) \qquad U : \mathsf{kdf}(\mathit{msg})
\end{aligned}
$$

where we omitted the homomorphic clauses for the symbols exp, sh, and $\langle\cdot, \cdot\rangle$. The types of some pattern variables are indicated on the right-hand side. All the remaining variables are of type *msg*. Applying f_1 to $\mathsf{IKE_m}$ we obtain $\mathsf{IKE_m^1}$. Here is the abstracted initiator role.

$$
\begin{aligned}
S_{\mathsf{IKE_m^1}}(A) = {}& \mathsf{send}(\mathit{sPIa}, o, \mathit{sA1}, \mathsf{exp}(g, x), \mathit{na}) \cdot\\
& \mathsf{recv}(\mathit{sPIa}, \mathit{SPIb}, \mathit{sA1}, \mathit{Gb}, \mathit{Nb}) \cdot \mathsf{Running} \cdot\\
& \mathsf{send}(\mathit{sPIa}, \mathit{SPIb}, A, B, \mathit{AUTHaa}', \mathit{sA2}, \mathit{tSa}, \mathit{tSb}) \cdot\\
& \mathsf{recv}(\mathit{sPIa}, \mathit{SPIb}, B, \mathit{AUTHba}', \mathit{sA2}, \mathit{tSa}, \mathit{tSb}) \cdot \mathsf{Secret} \cdot \mathsf{Commit}
\end{aligned}
$$

where $\mathit{SKa}' = \mathsf{kdf}(\mathsf{exp}(\mathit{Gb}, x))$ is the session key and the authenticators are defined by $\mathit{AUTHaa}' = \mathsf{mac}(\mathsf{sh}(A, B), o, \mathsf{exp}(g, x), \mathit{na}, \mathit{Nb}, \mathit{SKa}')$ and $\mathit{AUTHba}' = \mathsf{mac}(\mathsf{sh}(A, B), \mathit{Gb}, \mathit{Nb}, \mathit{na}, \mathit{SKa}')$. In a second step, we will remove most fields in the roles of $\mathsf{IKE_m^1}$ using untyped abstractions.

4.3.3. Soundness of typed abstractions

We now turn to showing the soundness of the typed abstractions. We do this by establishing conditions (i)–(iii) of our general soundness theorem (Theorem 4.6). The main ingredients that we need for this purpose are the preservation of intruder deduction, equalities, and disequalities. These properties will not hold without restrictions on the protocol, the property, and the typed abstraction. We first formulate these properties, their scope, and introduce these restrictions informally. We then state our soundness theorem. We defer the detailed motivation and formal definitions of the restrictions to the subsequent subsections.

Remark 4.13. For the correct interpretation of the properties of typed abstractions, it is important to remark that, given a term $t \in \mathcal{T}$, the expression $f(t)$ denotes the term in $\mathcal{T}$ obtained by evaluating the functional program f on t. This is in contrast to a purely syntactical reading of $f(t)$ such as in the equations E_f. Note that the term $f(t)$ itself is not an element of $\mathcal{T}$, since $f \notin \Sigma$.

Suppose σ is the substitution component of a concrete state, $T \subseteq \mathcal{M}^{TID}$ is a set of terms and $t, u \in \mathcal{M}^{TID}$ be terms.

– *Deducibility preservation*. Here, we require that

$$T\sigma \vdash_E t\sigma \implies f(T)f(\sigma) \vdash_E f(t)f(\sigma) \tag{P1}$$

This is needed to simulate the execution of receive events in the abstract protocol (condition (i) of Theorem 4.6) and for the preservation of secrecy (condition (ii) of Theorem 4.6). This property holds for typed abstractions f that are compatible with the rewrite theory (or R, Ax*-compatible* for short). This requires, for example, that f cannot remove fields that are extractable from a constructor using a rewrite rule (such as in decryption). We will discuss this property in more detail in Section 4.3.6.

– *Equality preservation.* This means that

$$t\sigma =_E u\sigma \implies f(t)f(\sigma) =_E f(u)f(\sigma) \tag{P2}$$

This property is needed for proving deducibility preservation and for the preservation of equalities in protocol properties (condition (c) of Definition 4.5 needed in condition (iii) of Theorem 4.6). This property holds if f is compatible with the axioms Ax and with the variants $[\![t]\!]_{R,Ax}$ of the term t, i.e., f preserves axioms and the equality associated with each variant of t. We denote by $cdom(F_f)$ the set of terms for which f is *variant-compatible*. Equality preservation is the topic of Section 4.3.5.

– *Disequality preservation.* This can be formulated as the reverse direction of equality preservation:

$$f(t)f(\sigma) =_E f(u)f(\sigma) \implies t\sigma =_E u\sigma \tag{P3}$$

Disequality preservation is needed to prevent that abstractions "fix" attacks on agreement properties (condition (d) of Definition 4.5 needed in condition (iii) of Theorem 4.6). In Section 4.3.7 and the full version [39], we present syntactic criteria for this property.

To establish these properties, we will use the following *substitution property*, which we will discuss in detail in Section 4.3.4. For terms t and well-typed and R, Ax-normal substitutions θ:

$$f(t\theta) = f(t)f(\theta) \tag{P4}$$

This property requires that t is in the *uniform domain* of f, written $t \in udom(F_f)$. This ensures that a term t and its instances $t\theta$ are uniformly transformed using the same equations of E_f.

Finally, we can state our soundness result for typed abstractions.

Theorem 4.14 (Soundness of typed abstractions)**.** *Let F_f be a R, Ax-compatible typed abstraction. Assume further that*

(i) $f(IK_0) \subseteq IK_0'$,
(ii) $\mathcal{M}_P \cup Sec_\phi \cup EqTerm_\phi^- \subseteq udom(F_f) \cap cdom(F_f)$, *and*
(iii) $f(t^{\vartheta(\iota)})f(\sigma) =_E f(u^{\vartheta(\kappa)})f(\sigma)$ *implies* $t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma$ *for all* $(\iota, \kappa, t, u) \in Eq_\phi^+$, *thread-id interpretations* ϑ, *and* R, Ax*-normal well-typed ground substitutions* σ, *and*
(iv) $f(t) = f(u)$ *implies* $t = u$, *for all* $e(t) \in Evt_\phi^+$ *and* $e(u) \in Evt(\mathcal{M}_P)$.

Then for all states $(tr, th, \sigma) \in reach(P, IK_0)$, *we have*

1. $(f(tr), f(th), f(\sigma)) \in reach(f(P), IK_0')$, *and*
2. $(tr, th, \sigma) \not\models \phi$ *implies* $(f(tr), f(th), f(\sigma)) \not\models f(\phi)$.

Proof. It suffices to establish conditions (i)–(iii) of Theorem 4.6 for $\mathcal{G} = (f, f)$ and $g = f$. Let $(tr, th, \sigma) \in reach(P, IK_0)$. We can assume without loss of generality that σ is R, Ax-normal.

Let $t \in \mathcal{M}_P \cup Sec_\phi$. Using assumptions (i)–(ii) and property (P1) (formalized in Corollary 4.33 below), we derive that $IK(tr)\sigma, IK_0 \vdash_E t^i\sigma$ implies $f(IK(tr))f(\sigma), IK_0' \vdash_E f(t^i)f(\sigma)$. Since $f(IK(tr)) = IK(f(tr))$, conditions (i) and (ii) of Theorem 4.6 hold.

To prove that condition (iii) of Theorem 4.6 is satisfied, we have to establish conditions (a)–(e) in Definition 4.5. We look at each of these conditions in turn.

- Condition (a): holds trivially since $\mathsf{nil} \notin ran(f)$.
- Condition (b): clearly holds since σ is well-typed and f is the identity on atoms.
- Condition (c): Here, $f(t^{\vartheta(\iota)})f(\sigma) =_E f(u^{\vartheta(\kappa)})f(\sigma)$ follows from $t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma$ by assumption (iii) and properties (P2) and (P4) (formalized in Theorems 4.23 and 4.18 below).
- Condition (d): holds by assumption (iii).
- Condition (e): holds by assumption (iv).

This completes the proof of the theorem. □

In the following, we discuss each of the properties (P1)–(P4) in more detail. We give examples motivating the restrictions under which they hold and we formally define these restrictions. We then establish that the properties hold under the respective restrictions. We start our discussion with the substitution property. Readers who wish to first get an overview of our abstractions before delving into the technical details may want to skip to Section 4.4.

4.3.4. Substitution Property (P4)

The following example shows that the substitution property does not hold unconditionally.

Example 4.15. Let $F_f = (f, E_f)$ be a typed abstraction such that E_f consists of the two equations $f(\mathsf{h}(X : \gamma_c)) = f(X)$ and $f(\mathsf{h}(Y : msg)) = \mathsf{h}(f(Y))$ where c is a constant and we have annotated the variables X and Y with their types for convenience. Let $t = \mathsf{h}(Z)$ and $\theta = \{Z \mapsto c\}$ where $Z : msg$. Then we have $f(t\theta) = f(\mathsf{h}(c)) = c \neq \mathsf{h}(c) = \mathsf{h}(Z\theta) = f(t)f(\theta)$.

The problem in this example is caused by the terms t and $t\theta$ being transformed by two distinct clauses. To avoid this, we must ensure that t and all its instance $t\theta$ are transformed uniformly, i.e., match the same clauses of E_f. We therefore require that

(i) the patterns in E_f do not overlap (pattern disjointness), and
(ii) all recursive calls of f on composed terms during the transformation of t are handled by the clauses of E_f, without recourse to the fall-back clauses in E_f^0.

This is formalized in the following two definitions.

Definition 4.16. A function specification $F_f = (f, E_f)$, where $E_f = [f(p_1) = u_1, \ldots, f(p_n) = u_n]$, is *pattern-disjoint* if the types in Π_f are pairwise disjoint, i.e., $\Gamma(p_i)\downarrow \cap \Gamma(p_j)\downarrow = \emptyset$ for all $i, j \in \widetilde{n}$ such that $i \neq j$.

Note that the abstractions defined in Examples 4.10 and 4.12 are pattern-disjoint, while the one in Example 4.15 is not. Let $\Pi_f = \Pi(E_f)$, where $\Pi(L) = \{\Gamma(p) \mid (f(p) = u) \in L\}$ denotes the set of pattern types of a list of equations L.

Definition 4.17 (Uniform domain)**.** We define the *uniform domain* of F_f by

$$udom(F_f) = \{t \in \mathcal{T} \mid \Gamma(Rec(F_f, t)) \subseteq \Pi_f{\downarrow} \cup \mathcal{Y}_{at}\}$$

where $Rec(F_f, t)$ is the set of terms u such that $f(u)$ is called in the computation of $f(t)$.

We will require that the protocol terms $t \in \mathcal{M}_P$ belong to $udom(F_f)$, which ensures that their instances $t\theta$ with R, Ax-normal substitutions θ are transformed uniformly. Since our protocol and property semantics do not distinguish states with $=_E$-equal substitutions, we can assume without loss of generality that σ is R, Ax-normal for all reachable states (tr, th, σ) of the protocol P.

Theorem 4.18 (Substitution property)**.** *Suppose that F_f is pattern-disjoint. Let $t \in udom(F_f)$ and θ be a well-typed and R, Ax-normal substitution. Then $f(t\theta) = f(t)f(\theta)$.*

We henceforth assume that F_f is pattern-disjoint. This concludes our discussion of (P4) and we now turn our attention to equality preservation.

4.3.5. Equality preservation (P2)

Using the substitution property, we can reduce (P2) to the property stating that $t\sigma =_E u\sigma$ implies $f(t\sigma) =_E f(u\sigma)$ for well-typed and R, Ax-normal substitutions σ. Using the decomposition of the equational theory (Σ, E) into (Σ, R, Ax), we further reduce this to the following two properties:

(P2.a) If $t =_{Ax} u$ then $f(t) =_{Ax} f(u)$ for all terms t and u.
(P2.b) $f(t\sigma) =_E f((t\sigma){\downarrow_{R,Ax}})$ for all terms t and well-typed R, Ax-normal substitutions σ.

Neither of these properties holds in this generality (recall Remark 4.13). The following example illustrates a violation of (P2.a).

Example 4.19. Let na and nb be nonces and let $F_f = (f, E_f)$ be a typed abstraction such that $E_f = [f(h(X)) = f(X)]$ with $X : \mathsf{exp}(msg, \beta_{na})$. We consider two terms $t = \mathsf{h}(\mathsf{exp}(\mathsf{exp}(g, na), nb))$ and $u = \mathsf{h}(\mathsf{exp}(\mathsf{exp}(g, nb), na))$. Then we have $f(t) = \mathsf{h}(\mathsf{exp}(\mathsf{exp}(g, na), nb))$ and $f(u) = \mathsf{exp}(\mathsf{exp}(g, nb), na)$. Hence, $t =_{Ax} u$ but $f(t) \neq_{Ax} f(u)$. The reason is that t and u are not transformed uniformly. In particular, t is transformed by a clause in E_f^0 which keeps t unchanged, while u is transformed by the clause in E_f which removes the hash function h.

To solve the problem described in Example 4.19, we introduce the notion of *Ax-closedness*, which requires that F_f is homomorphic for the constructors in $funsym(Ax)$ and that top-level constructors of axioms must not occur strictly inside any patterns' type. This is sufficient to prove property (P2.a).

Definition 4.20 (Ax-closedness)**.** F_f is *Ax-closed* if it is homomorphic for $funsym(Ax)$ and, for all equations $f(p) = u$ of E_f, we have $topsym(\Gamma(subterm(p) \setminus \{p\})) \cap topsym(Ax) = \emptyset$.

Note that the abstraction F_1 from Example 4.12 is Ax_{cs}-closed, since it is homomorphic for the only constructors exp and sh occurring in Ax_{cs} and these constructors occur at most in the top position of any of F_1's pattern types. We henceforth assume that F_f is Ax-closed. The following example exhibits a violation of (P2.b).

Example 4.21. Let $F_f = (f, E_f)$ be a typed abstraction which drops all symmetric encryptions, i.e., $E_f = [f(\{\!|X|\!\}_K) = f(X)]$, and let $t = \{\!|\{\!|m|\!\}_k|\!\}_k^{-1}$ for atomic terms m and k. Then $f(t) = \{\!|m|\!\}_k^{-1}$, but $f(t{\downarrow}_{R,Ax}) = f(m) = m$. Clearly, we have that $f(t) \neq_E f(t{\downarrow}_{R,Ax})$.

To establish (P2.b) for a term t, we make use of the finite variant property of our rewrite theory.

Definition 4.22 (Variant-compatibility)**.** We say that F_f is *variant-compatible* for t if, for all $(t', \theta) \in [\![t]\!]_{R,Ax}$, we have (i) $t', t\theta \in udom(F_f)$ and (ii) $f(t\theta) =_E f(t')$. We denote by $cdom(F_f)$ the set of terms for which f is variant-compatible.

For terms $t \in cdom(F_f)$, we can show (P2.b) using the substitution property. Note that variant-compatibility for t is checkable since $[\![t]\!]_{R,Ax}$ is finite due to the finite variant property. The theory including Diffie–Hellman exponentiation from Example 3.2 and the XOR theory in Example 3.3 both have the finite variant property.

Theorem 4.23 (Equality preservation)**.** *Suppose that F_f is pattern-disjoint and Ax-closed. Let $t, u \in cdom(F_f)$ and σ be a well-typed R, Ax-normal substitution. Then $t\sigma =_E u\sigma$ implies $f(t\sigma) =_E f(u\sigma)$.*

This concludes our treatment of (P2). We proceed with deducibility preservation.

4.3.6. Deducibility preservation (P1)

To preserve reachability and secrecy properties, our typed protocol abstractions need to preserve term deducibility, i.e., whenever a term t is deducible from a set of terms T then $f(t)$ is also deducible from $f(T)$. The following series of examples illustrates the main issues involved in the proof of deducibility preservation. The proof assumes T and t are R, Ax-normal and, without loss of generality that terms derived using the composition rule are immediately normalized. Thus, the interesting case is when a composition creates a term to which a rewrite rule $l \to r$ is applicable.

Example 4.24 (Preserving decryption)**.** Consider the composition rule Comp instantiated for asymmetric decryption, which derives $T \vdash_E \{X\}_K^{-1}$ from $T \vdash_E X$ and $T \vdash_E K$. We have to make sure that, for all instances of this rule, we can preserve this deduction under f. The most interesting instance is $X = \{u\}_{\mathsf{pk}(a)}$ and $K = \mathsf{pri}(a)$, in which case the conclusion can be reduced using the rewriting rule $\{\{u\}_{\mathsf{pk}(a)}\}_{\mathsf{pri}(a)}^{-1} \to u$ modeling decryption. In this case, we can produce the following derived standard rule for asymmetric decryption, which we call Adec.

$$\dfrac{\dfrac{T \vdash_E \{u\}_{\mathsf{pk}(a)} \quad T \vdash_E \mathsf{pri}(a)}{T \vdash_E \{\{u\}_{\mathsf{pk}(a)}\}_{\mathsf{pri}(a)}^{-1}}\ \mathsf{Comp} \qquad \{\{u\}_{\mathsf{pk}(a)}\}_{\mathsf{pri}(a)}^{-1} =_E u}{T \vdash_E u}\ \mathsf{Eq}$$

To preserve this derived rule under f, we have to show that we can derive $f(T) \vdash_E f(u)$ from $f(T) \vdash_E f(\{u\}_{\mathsf{pk}(a)})$ and $f(T) \vdash_E f(\mathsf{pri}(a))$. This clearly works if F_f is homomorphic for all four constructors on the left hand side of the decryption rewrite rule. Let us consider the more interesting case where $u = \langle u_1, u_2 \rangle$ and f pulls u_2 outside the encryption:

$$f(\{u_1, u_2\}_{\mathsf{pk}(a)}) = \langle \{f(u_1)\}_{f(\mathsf{pk}(a))}, f(u_2) \rangle.$$

By further assuming that f transforms decryptions, pairs, pk, and pri homomorphically, we obtain the required derivation as follows.

$$\dfrac{\dfrac{\dfrac{f(T) \vdash_E f(\{u_1, u_2\}_{\mathsf{pk}(a)})}{f(T) \vdash_E \{f(u_1)\}_{\mathsf{pk}(f(a))}}\,\mathsf{Proj}_1 \qquad f(T) \vdash_E \mathsf{pri}(f(a))}{f(T) \vdash_E f(u_1)}\,\mathsf{Adec} \qquad \dfrac{f(T) \vdash_E f(\{u_1, u_2\}_{\mathsf{pk}(a)})}{f(T) \vdash_E f(u_2)}\,\mathsf{Proj}_2}{f(T) \vdash_E f(\langle u_1, u_2\rangle)}\,\mathsf{Comp}$$

Here, the derived rules Proj_i are used for projection. These are formed by applying the composition rule (Comp) followed by a reduction (Eq).

Generally speaking, we have to ensure that if a composed term $t = d(t_1, \ldots, t_n)$ can be reduced to a term u then (the fields of) $f(u)$ can still be derived from $f(t_1), \ldots, f(t_n)$. The next examples illustrate that we must impose on f further restrictions related to the rewrite theory (in addition to Ax-closedness).

Example 4.25 (Dropping fields)**.** Consider the derivation of rule Adec in Example 4.24 and $u = \langle u_1, u_2\rangle$. Suppose f is now modified to drop u_2 from the encryption, i.e., $f(\{u_1, u_2\}_{\mathsf{pk}(a)}) = \{f(u_1)\}_{f(\mathsf{pk}(a))}$. Since $f(u_2)$ is lost, this clearly prevents us from deriving $f(T) \vdash_E f(u)$ in general.

This example shows that we cannot drop fields from argument positions of a constructor that can be extracted by a rewrite rule (here decryption).

Example 4.26 (Transforming non-enclosing constructors)**.** Suppose f transforms asymmetric encryptions and pk homomorphically, but drops the private key constructor pri, i.e., $f(\mathsf{pri}(X)) = f(X)$. Clearly, we cannot extract $f(u)$ from $\{f(u)\}_{\mathsf{pk}(f(a))}$ using the key $f(a)$, since $\{\{f(u)\}_{\mathsf{pk}(f(a))}\}^{-1}_{f(a)}$ is irreducible.

The problem here is that the decryption rewrite rule is no longer applicable. This can be avoided by requiring that f is homomorphic for the constructors of the left-hand side l of the rewrite rule other than those enclosing the extracted term in l (here, the key constructors pk and pri).

Example 4.27 (Non-linear variables)**.** This example illustrates another way to destroy the applicability of a rewrite rule by abstraction. Consider the rule $X \oplus (X \oplus Y) \to Y$ of the theory of XOR from Example 3.3. Suppose that E_f includes the following $\oplus$-equation, which drops the second component of a pair in the first argument of XOR if the second argument is also an XOR:

$$f(\langle U, V\rangle \oplus (W \oplus X)) = f(U) \oplus f(W \oplus X).$$

Also suppose that $f(X \oplus Y) = f(X) \oplus f(Y)$ for all other cases. Let $t = \langle k_1, k_2\rangle \oplus (\langle k_1, k_2\rangle \oplus m)$. Clearly, t is reducible to m, but this is not the case for $f(t) = k_1 \oplus (\langle k_1, k_2\rangle \oplus m)$.

In this case, the problem is that the two instances of X in the rewrite rule are transformed differently, which destroys the matching. This suggests that if a constructor c enclosing the extracted term in l has a non-linear variable at its ith argument position then the equations of f must not split the ith argument of c.

The examples above (partly) motivate the following definitions.

Definition 4.28. We call a typed abstraction $F_f = (f, E_f)$:

- *field-preserving* for position i of c if, for all equations of F_f of the form $f(c(p_1, \ldots, p_n)) = \langle e_1, \ldots, e_d \rangle$ and all $q \in \mathit{split}(p_i)$, there is a $j \in \widetilde{d}$ such that either $e_j = f(q)$ or $e_j = c(\ldots, \widehat{f}(\overline{q_i}), \ldots)$ and $q \in \mathit{set}(\overline{q_i})$.
- *non-splitting* for position i of c if p_i is not a pair for all equations of F_f of the form $f(c(p_1, \ldots, p_n)) = \langle e_1, \ldots, e_d \rangle$.

Note that if F_f is non-splitting for i of c then it is (trivially) field-preserving for position i of c. Moreover, if F_f is homomorphic for c then it is non-splitting for all argument positions i of c.

Definition 4.29 (Extractable position)**.** We say that a rewrite rule $l \to r \in R$ *extracts* position $i \in \widetilde{n}$ of $c \in \Sigma^n$ if there are terms $t_1, \ldots, t_n$ such that $c(t_1, \ldots, t_n) \in \mathit{subterm}(l)$ and $r = t_i$. We call i an *extractable position* of c if there is a rewrite rule $l \to r \in R$ that extracts position i from c.

For example, the projection rewrite rule $\pi_1(\langle X, Y \rangle) \to X$ extracts position 1 of pairs.

Definition 4.30 (Compatibility with rewrite theory)**.** A typed abstraction F_f is *compatible with a rewrite rule* $l \to r$ if one of the following conditions holds:

(C1) $l = c(u_1, \ldots, u_n)$ and $r = u_i$ for some $i \in \widetilde{n}$ such that $c \notin \mathit{topsym}(Ax)$,

(C2) $l = d(u_1, \ldots, u_{j-1}, c(v_1, \ldots, v_n), u_{j+1}, \ldots, u_m)$ and $r = v_i$ for some $j \in \widetilde{m}$ and $i \in \widetilde{n}$ such that $c \notin \mathit{topsym}(Ax)$, none of the v_i's is a pair, and the following conditions hold:

  (a) F_f is field-preserving for the extracted position i of c,
  (b) F_f is non-splitting for all positions i of c such that v_i is a non-linear variable of l, and
  (c) F_f is homomorphic for all $c' \in \mathit{funsym}(\{u_1, \ldots, u_{j-1}, v_1, \ldots, v_n, u_{j+1}, \ldots, u_m\})$.

(C3) l has an arbitrary shape and either

  (a) r is a constant,
  (b) $l \in \mathit{cdom}(F_f)$ and F_f is homomorphic for $\mathit{topsym}(l)$, or
  (c) $r \in \mathit{cdom}(F_f)$ and F_f is homomorphic for $\mathit{funsym}(l, r)$.

We say that F_f is *compatible with the rewrite theory* (Σ, Ax, R), or R, Ax*-compatible* for short, if F_f is pattern-disjoint, Ax-closed, and compatible with all rewrite rules in R.

We illustrate this definition with an example.

Example 4.31. Let us check that the typed abstraction $F_{f_1} = (f_1, E_{f_1})$ from Example 4.12 is compatible with the rewrite theory $\mathcal{R}_{cs} = (\Sigma_{cs}, Ax_{cs}, R_{cs})$ from Example 3.2. As already previously stated, F_{f_1} is pattern-disjoint and Ax_{cs}-closed. It remains to check that it is compatible with all rewrite rules in R_{cs}.

Let us consider the symmetric decryption rule $\{\!|\{\!|X|\!\}_K|\!\}_K^{-1} \to X$. We check that this rule satisfies condition (C2). We have $d = \{\!|\cdot|\!\}_\cdot^{-1}$, $u_1 = \{\!|X|\!\}_K$, $u_2 = K$, $c = \{\!|\cdot|\!\}_\cdot$, $v_1 = X$, and $v_2 = K$. First, we confirm that there are no symmetric encryptions at the top-level of any axiom and that none of v_1 and v_2 is a pair. Second, we check conditions (C2.a–c) in turn. Condition (C2.a) holds, since the only relevant equation of F_{f_1} is $f_1(\{\!|X, Y|\!\}_Z) = \langle f_1(X), f_1(Y) \rangle$, which is clearly field-preserving for the cleartext position 1 extracted by the rewrite rule. Furthermore, the only non-linear variable in l is K and F_1 is non-splitting for the relevant key position 2 of symmetric encryption. Hence, (C2.b) also holds. Condition (C2.c) holds vacuously, since the set of function symbols $\mathit{funsym}(\{v_1, v_2, u_2\})$ is empty.

Next, we verify that F_{f_1} is compatible with the signature verification rule $\mathsf{ver}([X]_{\mathsf{pri}(Y)}, \mathsf{pk}(Y)) \to X$. Since F_{f_1} is homomorphic for all constructors occurring in this rule and its right-hand side is a variable, it immediately follows that this rule satisfies condition (C3.c). Alternatively, we can show that it satisfies (C2). The compatibility of F_{f_1} with the asymmetric decryption and projection rules is justified similarly.

We first establish a version of deducibility preservation without substitutions.

Theorem 4.32 (Deducibility preservation)**.** *Let F_f be a R, Ax-compatible typed abstraction and let $T \cup \{t\}$ be a set of R, Ax-normal terms such that T contains all constants, i.e., $\mathcal{C} \subseteq T$. Then we have $T \vdash_E t$ implies $f(T) \vdash_E f(t)$.*

By combining this theorem with Theorems 4.23 and 4.18, we can now derive (P1) which we formalize as the following corollary.

Corollary 4.33 (Deducibility preservation with substitutions)**.** *Let F_f be a R, Ax-compatible typed abstraction. Suppose σ is a R, Ax-normal well-typed ground substitution and $T \cup \{u\}$ is a set of terms such that* (i) *$f(IK_0) \subseteq IK_0'$ and* (ii) *$T \cup \{u\} \subseteq udom(F_f) \cap cdom(F_f)$. Then we have that $T\sigma, IK_0 \vdash_E u\sigma$ implies $f(T)f(\sigma), IK_0' \vdash_E f(u)f(\sigma)$.*

This completes our discussion of (P1). Next, we discuss syntactic criteria for the disequality preservation in condition (iii) of Theorem 4.14.

4.3.7. Syntactic criteria for disequality preservation (P3)

Condition (iii) of Theorem 4.14 requires that, for all $(\iota, \kappa, t, u) \in Eq_\phi^+$, all thread-id interpretations ϑ, and all R, Ax-normal well-typed ground substitutions σ, we have

$$f(t^{\vartheta(\iota)})f(\sigma) =_E f(u^{\vartheta(\kappa)})f(\sigma) \implies t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma. \qquad (\mathcal{I})$$

Since the universal quantification over substitutions makes this condition hard to check in practice, we propose syntactic criteria for its verification.

Here, we present such a criterion that is applicable if t and u do not contain any message variables. Assuming that $f(t) = t$ and $f(u) = u$, we can derive $t^{\vartheta(\iota)}f(\sigma) =_E u^{\vartheta(\kappa)}f(\sigma)$ from the premise of condition ($\mathcal{I}$). Since we have that $f(X\sigma) = X\sigma$ for all non-message variables $X \in dom(\sigma)$, we obtain $t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma$ as required. Hence, we have just proved the following simple syntactic criterion.

Proposition 4.34. *Let $(\iota, \kappa, t, u) \in Eq_\phi^+$ such that* (i) *$(vars(t) \cup vars(u)) \cap \mathcal{V}_{msg} = \emptyset$, and* (ii) *$f(t) = t$ and $f(u) = u$. Then, for all thread-id interpretations ϑ and well-typed ground substitutions σ, we have that $f(t^{\vartheta(\iota)})f(\sigma) =_E f(u^{\vartheta(\kappa)})f(\sigma)$ implies $t^{\vartheta(\iota)}\sigma =_E u^{\vartheta(\kappa)}\sigma$.*

Note that for this criterion to be applicable, we require that f is the identity for the terms in positively occurring equations. This is often the case, as these terms typically have a simple structure, e.g., nonces or timestamps. However, this criterion cannot be used to justify the soundness of the typed abstraction from Example 4.12 with respect to the property ϕ_{auth} from Example 3.11. Although we can expand the equality of the two tuples in that example into a conjunction of six simpler equations, we can only apply the criterion above to the first four of these. The last two contain message variables and require a more general syntactic criterion for condition ($\mathcal{I}$). In the full version [39], we present such a criterion, which covers the case where message variables may occur on one side of the equation.

4.4. Atom-and-variable removal abstractions

Typed abstractions offer a wide range of possibilities to transform cryptographic operations including subterm removal, splitting, and pulling fields outside of such an operation. We complement these abstractions with two kinds of *untyped abstractions*. The first type, discussed here, allows us to remove unprotected atoms and variables of any type. The second type removes redundancy in the form of intruder-derivable terms and is discussed in the next subsection.

4.4.1. Specification of atom-and-variable removal

We first present the formal definition of atom-and-variable removal abstractions, then we motivate some restrictions needed for soundness, and finally we illustrate the application of atom-variable removal on our running example.

An atom-and-variable removal abstraction does not remove all occurrences of an atom or a variable from a given term t, but those that are fields of t. Intuitively, these unprotected atoms and variables do not themselves provide any security properties and can therefore safely be removed. This intuition is most obvious for atom removal: the intruder already knows all constants and agent names and he can replace unprotected fresh values by his own ones of the same type. In the following definition, we formulate atom-and-variable removal abstracions, where we use the abbreviation $av(t) = atoms(t) \cup vars(t)$.

Definition 4.35. An *atom-and-variable removal abstraction* is a general abstraction $\mathcal{G} = (rem_T, rem_T)$, where $T \subseteq av(\mathcal{M}_P)$ is a parameter denoting the set of atoms and variables to be removed and $rem_T : \mathcal{T} \to \mathcal{T} \cup \{\mathsf{nil}\}$ is defined by

(i) $rem_T(u) = \mathsf{nil}$ if $u \in T \cup T^{TID}$

(ii)

$$rem_T(\langle t_1, t_2\rangle) = \begin{cases} rem_T(t_1) & \text{if } rem_T(t_2) = \mathsf{nil} \\ rem_T(t_2) & \text{if } rem_T(t_1) = \mathsf{nil} \\ \langle rem_T(t_1), rem_T(t_2)\rangle & \text{otherwise} \end{cases}$$

(iii) $rem_T(t) = t$ for all other terms.

By point (i) any term in $T \cup T^{TID}$ is removed. Note that this covers unindexed terms in protocol specifications and security properties and indexed terms during execution. Point (ii) allows us to remove pairs or their components. Point (iii) ensures that all other terms remain unchanged. Note that, for all terms t, $rem_T(t)$ either does not contain nil or equals nil. Hence by Definition 4.4, nil does not occur in abstracted roles and traces and therefore $rem_T(P)$ is a protocol (see Definition 3.7).

Due to point (iii) of Definition 4.35, atom-and-variable removal abstractions cannot remove an atom or variable from a non-pair term. It is even unclear how to define this in general. Let us attempt to define a hypothetical variant rem'_T of rem_T. Consider a non-pair composed term $t = c(a_1, \ldots, a_n)$ and suppose rem'_T maps some but not all arguments of t to nil. One may think of two possible definitions for $rem'_T(t)$: (1) $rem'_T(t) = \mathsf{nil}$ or (2) $rem'_T(t)$ is the tuple consisting of the non-nil arguments of c. The following two examples consider each of the two definitions in turn and show that neither of them preserves deducibility.

Example 4.36. Consider the terms $t = \langle na, \{\!|nb|\!\}_{na}\rangle$ and $u = nb$ containing the nonces na and nb. Let $T = \{na\}$. Then, we have $rem'_T(t) = \mathsf{nil}$ and $rem'_T(u) = nb$. Moreover, we also have $t \vdash_E u$, but $rem'_T(t) \vdash_E rem'_T(u)$ does not hold, as nb is not deducible from nil.

Example 4.37. Suppose that $\mathsf{h}_1, \mathsf{h}_2 \in \Sigma^2$ are binary hash functions and Ax contains the following axiom:

$$\mathsf{h}_1(\mathsf{h}_2(X, Y), Z) \simeq \mathsf{h}_1(\mathsf{h}_2(X, Z), Y).$$

Consider the two terms $t = \mathsf{h}_1(\mathsf{h}_2(n_1, n_2), n_3)$ and $u = \mathsf{h}_1(\mathsf{h}_2(n_1, n_3), n_2)$ where n_1, n_2 and n_3 are nonces and let $T = \{n_3\}$. Then we have $rem'_T(t) = \mathsf{h}_2(n_1, n_2)$, and $rem'_T(u) = \mathsf{h}_1(n_1, n_2)$. Moreover, we have $t =_{Ax} u$, but $rem'_T(t) =_{Ax} rem'_T(u)$ fails to hold. Hence, t and u are derivable from each other, while neither of $rem'_T(t)$ and $rem'_T(u)$ is derivable from the other.

Similar counterexamples can also be constructed if variable removal abstractions are considered. This highlights the necessity of point (iii) in Definition 4.35.

The following example shows that the soundness of rem_T calls for a restriction of the occurrences of the removed atoms and variables. Namely, they may occur exclusively as fields of a term, i.e., we cannot remove an atom or variable that also occurs under a cryptographic operation in the same term.

Example 4.38. Consider terms $t = \langle na, \mathsf{h}_1(na)\rangle$ and $u = \mathsf{h}_2(na)$, where h_1 and h_2 are hash function symbols and na is a nonce. With $T = \{na\}$ we have $rem_T(t) = \mathsf{h}_1(na)$ and $rem_T(u) = \mathsf{h}_2(na)$. Moreover, we also have $t \vdash_E u$, but $rem_T(t) \vdash_E rem_T(u)$ fails to hold.

This example motivates the following definition.

Definition 4.39 (Clear terms)**.** A term u is *clear* in a term t if $u \notin subterm(split(t) \setminus \{u\})$, i.e., u occurs at most as a field in t. For sets of terms T and U, we say that T *is clear* in a term t if every term in T is clear in t and that T is *clear in a set of terms* U if T is clear in every term in U.

Note that u is also clear in t if it does not appear at all in t. Our soundness result requires that all variables and atoms in T are clear in the terms to which rem_T is applied. Moreover, it requires that the elements of T do not appear in the properties of interest.

In the following example, we illustrate the use of atom-and-variable removal abstractions to transform $\mathsf{IKE}^1_\mathsf{m}$ into $\mathsf{IKE}^2_\mathsf{m}$.

Example 4.40 ($\mathsf{IKE}^1_\mathsf{m}$ to $\mathsf{IKE}^2_\mathsf{m}$)**.** We use atom-and-variable removal to simplify the protocol $\mathsf{IKE}^1_\mathsf{m}$. First, we recall the specification of (the initiator role of) $\mathsf{IKE}^1_\mathsf{m}$.

$$\begin{aligned} S_{\mathsf{IKE}^1_\mathsf{m}}(A) = {} & \mathsf{send}(\underline{sPIa}, o, \underline{sA1}, \mathsf{exp}(g, x), na) \cdot \\ & \mathsf{recv}(\underline{sPIa}, \underline{SPIb}, \underline{sA1}, Gb, Nb) \cdot \mathsf{Running} \cdot \\ & \mathsf{send}(\underline{sPIa}, \underline{SPIb}, A, B, AUTHaa', \underline{sA2}, \underline{tSa}, \underline{tSb}) \cdot \\ & \mathsf{recv}(\underline{sPIa}, \underline{SPIb}, B, AUTHba', \underline{sA2}, \underline{tSa}, \underline{tSb}) \cdot \mathsf{Secret} \cdot \mathsf{Commit} \end{aligned}$$

To highlight the changes in this abstraction step, we have underlined the terms to be removed from $\mathsf{IKE}^1_\mathsf{m}$: the constants $sA1$, $sA2$, tSa, and tSb, the fresh values $sPIa$ and $sPIb$, and the variables $SPIa$ and $SPIb$. We use the atom-and-variable removal abstraction rem_T with parameter $T = \{sA1, sA2, tSa, tSb, sPIa, sPIb, SPIa, SPIb\}$. Note that we can neither remove the constant o nor the variables A and B, since these terms are not clear in the authenticators $AUTHaa'$ and $AUTHab'$. Applying

rem_T to $\mathsf{IKE}^1_\mathsf{m}$, we obtain the (initiator role of the) protocol $\mathsf{IKE}^2_\mathsf{m}$ as given below.

$$S_{\mathsf{IKE}^2_\mathsf{m}}(A) = \mathsf{send}(o, \mathsf{exp}(g, x), na) \cdot \mathsf{recv}(Gb, Nb) \cdot \mathsf{Running} \cdot \mathsf{send}(A, B, AUTHaa') \cdot \mathsf{recv}(B, AUTHba') \cdot \mathsf{Secret} \cdot \mathsf{Commit}$$

Note that the session keys and the authenticators are non-pair composed terms and hence remain untouched. We later use a redundancy removal to further simplify $\mathsf{IKE}^2_\mathsf{m}$ by removing intruder-derivable occurrences of the constant o and the agent variables A and B from the role descriptions.

4.4.2. Soundness for atom-and-variable removal abstractions

We now turn our attention to the soundness result for atom-and-variable removal abstraction. This result requires that we restrict our attention to *well-formed* protocols. To define this predicate on protocols, we first introduce the notion of *accessible variables*.

Definition 4.41 (Accessible variables)**.** We say that a variable X is *accessible* in a term t if either

(i) $t = X$ or
(ii) $t = c(t_1, \ldots, t_n)$ for some $c \in \Sigma^n$, some position $i \in \widetilde{n}$ of c is extractable, and X is accessible in t_i.

Intuitively, a variable X is accessible in a term t if there is a path from t's root to an occurrence of X consisting of only extractable positions. This is to ensure that if X is accessible then it is potentially deducible. For example, X is accessible in $\{\!|X|\!\}_k$ since an agent can derive X from $\{\!|X|\!\}_k$ using the rewrite rule $\{\!|\{\!|X|\!\}_K|\!\}_K^{-1} \to X$, of course provided it also knows k. In contrast, X is not accessible in $\mathsf{h}(X)$ since there is no way to deduce X from $\mathsf{h}(X)$. However, X is accessible in $\langle X, \mathsf{h}(X)\rangle$ since it is accessible using the first projection. We now give the formal definition of *well-formed protocols*.

Definition 4.42. A protocol P is *well-formed* if all non-agent variables first occur in receive events, i.e., for all roles $R \in dom(P)$ and all send and receive events $ev(t)$ in role $P(R)$ and all non-agent variables $X \in vars(t) \setminus \mathcal{V}_\alpha$, there is an event $\mathsf{recv}(u)$ in $P(R)$ such that $\mathsf{recv}(u)$ equals or precedes $ev(t)$ in $P(R)$ and X is accessible in u.

A well-formed protocol captures the intuition that an agent must know what he sends and the elements that he receives into variables are accessible, e.g., by decrypting a ciphertext. Our notion of well-formedness is a weaker form of executability, which would additionally require that the agent also knows the relevant keying material. Hence, all practical protocols satisfy this condition.

Our soundness result for atom-and-variable removal abstractions is stated in the following theorem.

Theorem 4.43 (Soundness for atom-and-variable removal abstractions)**.** *Let P be a well-formed protocol, $\phi \in \mathcal{L}_P$ a property, $T \subseteq av(\mathcal{M}_P)$ a set of atoms and variables such that*

(i) *T is clear in $\mathcal{M}_P$,*
(ii) *$T \cap av(EqTerm_\phi) = \emptyset$,*
(iii) *$\mathsf{nil} \notin rem_T(Sec_\phi \cup Evt_\phi)$, and*
(iv) *$IK_0 \subseteq IK'_0$,*
(v) *for all $e(t) \in Evt^+_\phi$ and $e(u) \in Evt(\mathcal{M}_P)$, we have $rem_T(t) = rem_T(u)$ implies $t = u$.*

Then for all states $(tr, th, \sigma) \in reach(P, IK_0)$, there is a ground substitution σ' such that

1. $(rem_T(tr), rem_T(th), \sigma') \in reach(rem_T(P), IK_0')$,
2. $(tr, th, \sigma) \not\models \phi$ *implies* $(rem_T(tr), rem_T(th), \sigma') \not\models rem_T(\phi)$.

To preserve attacks, condition (i) ensures that the removed atoms and variables are clear in all protocol terms. Condition (ii) requires that no removed atom or variable occurs in the property's equalities. Together with condition (iii) it implies condition (a) of the definition of safe formulas (Definition 4.5). Condition (iv) requires that the initial knowledge of the intruder in the abstract protocol subsumes that in the original protocol. Finally, condition (v) reflects condition (e) of the definition of safe formulas (Definition 4.5).

We prove Theorem 4.43 by composing two separate soundness results for atom removal and for variable removal abstractions, respectively. Their statements and proofs appear in the full version [39].

4.5. Redundancy removal abstractions

The second kind of untyped abstractions are redundancy removal abstractions. A redundancy removal abstraction rd enables the elimination of redundancies within each role of a protocol. Intuitively, a protocol term t appearing in a role r can be abstracted to $rd(t)$ if t and $rd(t)$ are derivable from each other under the intruder knowledge T containing the terms preceding t in r and the initial knowledge IK_0. For example, we can simplify $r = \mathsf{send}(t) \cdot \mathsf{recv}(\langle t, u \rangle)$ to $\mathsf{send}(t) \cdot \mathsf{recv}(u)$. In contrast to atom-and-variable removal, redundancy removal can also remove composed terms. It is therefore a very effective ingredient for automatic abstraction, which we describe in Section 6.

4.5.1. Specification of redundancy removal abstractions

We now formally define our class of redundancy removal abstractions.

Definition 4.44. A *redundancy removal abstraction* for a protocol P is a general abstraction $\mathcal{G} = (rd, id)$ where id is the identity function on $\mathcal{T}$ and the function $rd : \mathcal{T} \to \mathcal{T} \cup \{\mathsf{nil}\}$ satisfies two conditions:

(i) for all $R \in dom(P)$, we have that $RD_{rd}(IK_0, P(R))$ holds, where the predicate $RD_{rd}(T, S)$ is inductively defined by the following three rules:

$$\frac{}{RD_{rd}(T, \epsilon)} \qquad \frac{RD_{rd}(T, r)}{RD_{rd}(T, s \cdot r)}\ s \in Sig$$

$$\frac{RD_{rd}(T \cup \{t\}, r) \quad T, \mathcal{V}_\alpha, rd(t) \vdash_E t \quad T, \mathcal{V}_\alpha, t \vdash_E rd(t)}{RD_{rd}(T, ev(t) \cdot r)}\ ev \in \{\mathsf{send}, \mathsf{recv}\}$$

Note that in these rules, $rd(t)$ is removed from the deducibility conditions if it equals nil. We also define $rd(t^i) = rd(t)^i$ for all $i \in TID$ and $t \in \mathcal{M}_P$.

(ii) for all terms $t \notin \mathcal{M}_P \cup \mathcal{M}_P^{TID}$, we have $rd(t) = t$.

Intuitively, the predicate $RD_{rd}(T, ev(t) \cdot r)$ ensures for a protocol message t that the intruder is able to derive t from $rd(t)$ and his knowledge T, and vice versa. The first rule says that $RD_{rd}(T, \epsilon)$ always holds. This captures the intuition that any redundancy removal works for the empty role description. The second rule allows us to ignore all the signals events as they do not affect the intruder's knowledge. In the last rule, the first premise requires that the predicate holds for T plus the term t in the first element of

the event sequence, and the tail r. By adding t to T, we capture the fact that the intruder learns t after the event $ev(t)$ has been executed. The second premise ensures that t is derivable from T, $\mathcal{V}_\alpha$, and $rd(t)$. The set of agent variables $\mathcal{V}_\alpha$ is added to T to symbolically represent the intruder knowledge of all agents. The third premise is the same as the second one, except that the roles of t and $rd(t)$ are swapped. We will usually identify the pair (rd, id) with its first, non-trivial component rd.

In the following example, we illustrate the use of redundancy removal abstractions to further simplify the protocol $\mathsf{IKE}^2_{\mathsf{m}}$.

Example 4.45. First, we recall $\mathsf{IKE}^2_{\mathsf{m}}$ whose role descriptions are given below, where the authenticator terms $AUTHxx'$ correspond to abstractions of the corresponding $AUTHxx$ terms, resulting from the first abstraction step described in Example 4.12.

$$\begin{aligned} S_{\mathsf{IKE}^2_{\mathsf{m}}}(A) &= \mathsf{send}(\underline{o}, \mathsf{exp}(g,x), na) \cdot \mathsf{recv}(Gb, Nb) \cdot \mathsf{Running} \cdot \\ &\quad \mathsf{send}(\underline{A}, \underline{B}, AUTHaa') \cdot \mathsf{recv}(\underline{B}, AUTHba') \cdot \mathsf{Commit} \\ S_{\mathsf{IKE}^2_{\mathsf{m}}}(B) &= \mathsf{recv}(\underline{o}, Ga, Na) \cdot \mathsf{send}(\mathsf{exp}(g,y), nb) \cdot \\ &\quad \mathsf{recv}(\underline{A}, \underline{B}, AUTHab') \cdot \mathsf{Running} \cdot \mathsf{send}(\underline{B}, AUTHbb') \cdot \mathsf{Commit} \end{aligned}$$

To remove the underlined terms, we use the following redundancy removal abstraction rd:

$$\begin{aligned} &rd(\langle \underline{o}, \mathsf{exp}(g,x), na\rangle) = \langle \mathsf{exp}(g,x), na\rangle \\ &rd(\langle \underline{o}, Ga, Na\rangle) = \langle Ga, Na\rangle \\ &rd(\langle \underline{A}, \underline{B}, AUTHaa'\rangle) = AUTHaa' \\ &rd(\langle \underline{A}, \underline{B}, AUTHab'\rangle) = AUTHab' \\ &rd(\langle \underline{B}, AUTHba'\rangle) = AUTHba' \\ &rd(\langle \underline{B}, AUTHbb'\rangle) = AUTHbb' \\ &rd(t) = t \text{ for all other messages } t \end{aligned}$$

It is not difficult to see that rd satisfies the conditions of Definition 4.44. Applying rd to $\mathsf{IKE}^2_{\mathsf{m}}$, we obtain the protocol $\mathsf{IKE}^3_{\mathsf{m}}$ specified as follows.

$$\begin{aligned} S_{\mathsf{IKE}^3_{\mathsf{m}}}(A) &= \mathsf{send}(\mathsf{exp}(g,x), na) \cdot \mathsf{recv}(Gb, Nb) \cdot \mathsf{Running} \cdot \\ &\quad \mathsf{send}(AUTHaa') \cdot \mathsf{recv}(AUTHba') \cdot \mathsf{Commit} \\ S_{\mathsf{IKE}^3_{\mathsf{m}}}(B) &= \mathsf{recv}(Ga, Na) \cdot \mathsf{send}(\mathsf{exp}(g,y), nb) \cdot \\ &\quad \mathsf{recv}(AUTHab') \cdot \mathsf{Running} \cdot \mathsf{send}(AUTHbb') \cdot \mathsf{Commit} \end{aligned}$$

In Fig. 3, we depict the message sequence chart of this protocol with all abbreviations expanded.

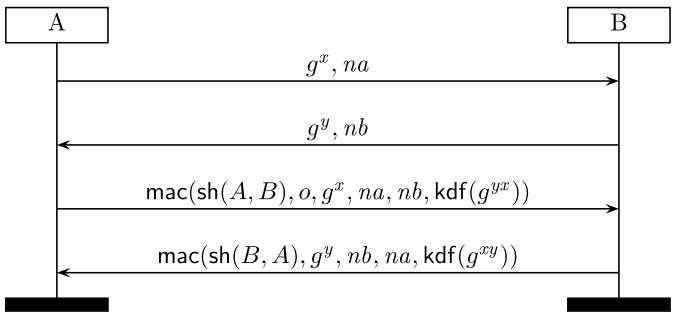

Fig. 3. The $\mathsf{IKE}^3_\mathsf{m}$ protocol.

4.5.2. Soundness for redundancy removal abstractions

The soundness result for redundancy removal abstractions is stated in the following theorem.

Theorem 4.46 (Soundness for redundancy removal abstractions)**.** *Let P be a protocol, $\phi \in \mathcal{L}_P$ a property, and $rd \in RD_P$ a redundancy removal abstraction. Suppose that*

(i) $IK_0 \subseteq IK'_0$,
(ii) $\mathsf{nil} \notin rd(Evt_\phi)$,
(iii) *for all* $e(t) \in Evt^+_\phi$ *and* $e(u) \in Evt(\mathcal{M}_P)$, *we have* $rd(t) = rd(u)$ *implies* $t = u$.

Then for all states $(tr, th, \sigma) \in reach(P, IK_0)$, *we have*

1. $(rd(tr), rd(th), \sigma) \in reach(rd(P), IK'_0)$, *and*
2. $(tr, th, \sigma) \not\models \phi$ *implies* $(rd(tr), rd(th), \sigma) \not\models \phi$.

4.6. Well-formedness preservation for protocol abstractions

In this section, we present well-formedness preservation results for our three types of protocol abstractions. These results are required for the composition of typed abstractions, atom-and-variable removal abstractions, and redundancy removal abstractions to transform well-formed protocols.

Proposition 4.47. *Let $F_f = (f, E_f)$ be a typed abstraction. If P is well-formed then so is $f(P)$.*

Proposition 4.48. *Let T be a set of atoms and variables such that T is clear in $\mathcal{M}_P$. If P is well-formed, then so is $rem_T(P)$.*

Proposition 4.49. *Let rd be a redundancy removal abstraction and P be a well-formed protocol. Assume that for all non-agent variables $X \in \mathcal{V}_P$ and all receive events $\mathsf{recv}(t)$ in which X first occurs, we have that X is accessible in $rd(t)$. Then $rd(P)$ is well-formed.*

The proofs of these propositions can be found in the full version [39].

5. Using protocol abstractions for efficient verification

Recall that our aim is to make protocol verification more efficient. Given a protocol and a property, our high-level idea is to construct a simpler version of the protocol and the property that is easier to

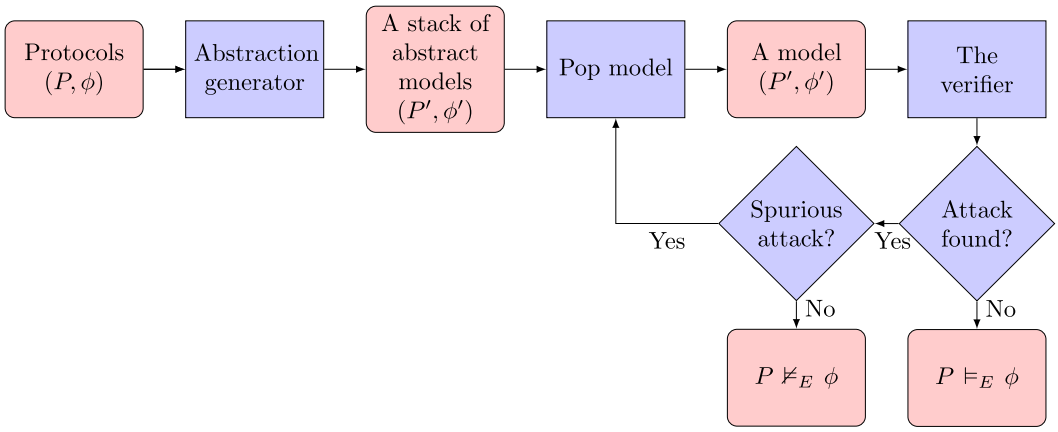

Fig. 4. The abstraction workflow for the analysis of security protocols.

verify. In particular, if the simpler version is a sound abstraction of the original, then we can conclude that the original also satisfies its property.

In the previous section, we gave sufficient conditions for abstractions to be sound. However, not all sound abstractions are useful for verification. In particular, if an abstraction is vulnerable to an attack that does not apply to the original, then we might waste verification time to find this attack, without being able to draw any conclusion about the original. Ideally, abstractions for verification extract the "core" of the cryptographic protocol, i.e., those parts of the protocol that are instrumental in achieving the property, and omit all other constructions. In this ideal case, the abstractions would have exactly the same properties as the original.

In this section, we describe an algorithm for efficient protocol verification based on such abstractions. Because we do not have a direct construction algorithm for sound abstractions, we use heuristics to generate reasonable abstractions and then check if they meet the soundness conditions. The workflow of our algorithm is described in Fig. 4: we first generate a stack of successively more abstract protocols and properties, with at the bottom the original, and at the top an abstract protocol that we hope represents the core of the protocol required to establish the property.

We then verify the protocols and the properties in this stack top-down, based on the assumption that it is more efficient to analyze a more abstract protocol. We provide empirical evidence for this in the next section. If we can successfully verify a protocol from the stack, we know the original protocol meets its property, and we can stop the analysis. If we find an attack, we try to reconstruct the attack on the original. If this is possible, we know the original protocol does not satisfy the property. If not, the attack is spurious, and we proceed to the next protocol on the stack, which is less abstract than the previous one.

We describe in Section 5.1 how we generate abstractions and in Section 5.2 how we check for spurious attacks.

5.1. Generating abstractions for verification

Our heuristics to generate abstractions uses three strategies, corresponding to our three types of abstractions, which we apply in order. After applying a strategy, we check if the resulting abstraction is sound. We discuss the three strategies in turn.

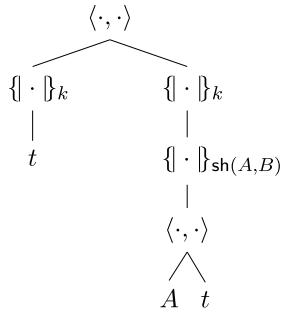

Fig. 5. Structure of u.

5.1.1. Simplifying or removing constructors that might not be needed to establish the property

Many protocols use (cryptographic) constructors that are, at most, needed to guarantee some (but not all) of its desired properties. To see this, consider the following example.

Example 5.1 (The purpose of cryptographic constructors)**.** Let k be a session key and t an arbitrary term. Let u be defined as $(\{|t|\}_k, \{|\{|A, t|\}_{\mathsf{sh}(A,B)}|\}_k)$. In Fig. 5 we give a graphical representation of the structure of u.

If the security property encodes that t needs to be authenticated, we look for the strongest mechanism that could guarantee this. Within u, this would be the symmetric encryption with the long-term key $\mathsf{sh}(A, B)$, since we do not need to rely on the secrecy of the session key. Thus, within u, authentication of t can be guaranteed by this constructor only. If we are *only* interested in authentication of t, we can consider removing t from the protection of all other constructors, which in this case are the encryptions with k.

If the security property encodes that t needs to be secret, the situation changes, since secrecy needs to be guaranteed for all occurrences of t, and not just one. Thus, in the left branch, secrecy of t is guaranteed on the basis of the session key k, whereas in the right branch, secrecy is guaranteed on the basis of both constructors. Thus, within u, t's secrecy depends on the secrecy of another term, and not just the long-term key. When we want to abstract the term u sent in a protocol without introducing new attacks, we need to ensure we do not make the situation worse. Thus, we would not modify the left branch. However, in the right branch we could remove t from the protection of either one of the constructors, since the overall guarantee within u would still be the same.

We will exploit this intuition by first determining which (sub)terms are relevant for establishing the desired property. We represent this by assigning *security labels* to each of them. In a second step, we give an algorithm that moves subterms out of their encapsulating constructor as long as their security labels are not increased.

For the first step, we first define which constructors serve which purpose. For example, a hash function does not authenticate its subterms, but it does not reveal its subterms either, and hence may be used in the context of secrecy. We differentiate between two main objectives (authentication and confidentiality) and assign one of three labels for each.

Security labels. We define the set of (security) labels $\mathit{Label} = \{\mathsf{NO}, \mathsf{MAYBE}, \mathsf{YES}\}$, with a total order $\leqslant_{lb}$ such that $\mathsf{NO} \leqslant_{lb} \mathsf{MAYBE} \leqslant_{lb} \mathsf{YES}$. The lowest label NO encodes that the property is not met, the highest label YES that it can be met, and the middle label MAYBE that it depends on the properties of another term (e.g., a session key).

Table 2
Security labels for different cryptographic operations, encoding what they might achieve for their strict subterms

Top-level constructor of t	Confidentiality $\ell_c(t)$	Authentication $\ell_a(t)$
symmetric encryptions or MACs with long-term keys	YES	YES
MACs with session keys	YES	MAYBE
symmetric encryptions with session keys	MAYBE	MAYBE
public-key encryptions or hashes	YES	NO
signatures	NO	YES
others	NO	NO

The labels for constructors (i.e., the guarantees they establish for their subterms) are specified by the functions ℓ_a and ℓ_c defined in Table 2. When extending these labels to a complete protocol, the simplest case occurs for authentication, where we simply determine the label of the strongest constructor that provides authenticity for the target term t. Intuitively, t needs to be authenticated only once in the protocol.

We define an auxiliary function *pathmax* that takes a term x, a position p, and a labelling function f, and returns the maximum of f applied to all subterms from the root along the path to p. Formally, we define $\mathit{pathmax}(x, p, f) = \max(\{f(x|_{p_1}) \mid \exists p_2.\ p_2 \neq \epsilon \wedge p_1 \cdot p_2 = p\})$. We will use *pathmax* to take the maximum of f over all constructors within x that might authenticate $x|_p$ or keep it confidential.

Definition 5.2 (Protocol authentication label)**.** Let P be a protocol, ϕ a property, and t a term. We define the protocol authentication label $\mathit{authlabel}(P, \phi, t)$ as follows:

1. $\mathit{authlabel}(P, \phi, t) = \mathsf{NO}$, if $IK_0, \mathcal{V}_\alpha \vdash_E t$ or $t \notin \mathit{subterm}(\mathcal{M}_P) \cap \mathit{subterm}(\mathit{EqTerm}_\phi)$, and
2. $\mathit{authlabel}(P, \phi, t) = \max(\{\mathit{pathmax}(u, p, \ell_a) \mid u \in \mathcal{M}_P \wedge u|_p = t\})$, otherwise.

For confidentiality, we cannot take the maximum over all positions, since we need to ensure that *all* occurrences of t are protected. Thus, we consider the labels of all paths on which t occurs, and take the minimum.

Definition 5.3 (Protocol confidentiality label)**.** Let P be a protocol, ϕ a property, and t a term. We define the protocol authentication label $\mathit{conflabel}(P, \phi, t)$ as follows:

1. $\mathit{conflabel}(P, \phi, t) = \mathsf{NO}$, if $IK_0, \mathcal{V}_\alpha \vdash_E t$ or $t \notin \mathit{subterm}(\mathcal{M}_P) \cap \mathit{subterm}(\mathit{Sec}_\phi)$, and
2. $\mathit{conflabel}(P, \phi, t) = \min(\{\mathit{pathmax}(u, p, \ell_c) \mid u \in \mathcal{M}_P \wedge u|_p = t\})$, otherwise.

Example 5.4. Let us consider the terms u and t in Fig. 5. Suppose that $u \in \mathcal{M}_P$, $t \in \mathit{subterm}(\mathit{Sec}_\phi \cap \mathit{EqTerm}_\phi)$, and $IK_0, \mathcal{V}_\alpha \nvdash_E t$. Let P be a protocol such that (a) u occurs in $\mathcal{M}_P$ and (b) all occurrences of t in $\mathcal{M}_P$ are within u. Then we have $\mathit{authlabel}(P, \phi, t) = \mathsf{YES}$ and $\mathit{conflabel}(P, \phi, t) = \mathsf{MAYBE}$.

We use the label definitions to construct an abstraction in the following way. First, we compute the authentication and confidentiality labels for all terms in the protocol and property. Second, we construct candidate abstractions in which we pull subterms out of their constructors (e.g., abstracting $\{\!|x_1, x_2, x_3|\!\}_k$ to $\langle x_2, \{\!|x_1, x_3|\!\}_k\rangle$). For each candidate, we compute the new labels. Our main criterion for applying an abstraction is that,

> *for each term, the labels in the candidate abstraction are not lower than those of the corresponding terms in the original.*

Additionally, we can remove a constructor entirely, if all its arguments can be pulled out. To prevent the introduction of spurious attacks, we do not perform abstractions that turn two non-unifiable subterms into unifiable ones. In the full version [39], we discuss in more detail how to generate an abstraction based on security labels.

5.1.2. Removing atoms or variables that might not be needed to establish the property

In many cases, there are atoms or variables that occur in the protocol messages but that do not occur in the security property ϕ. They might therefore be redundant and we generate an abstraction in which they are removed from the protocol messages. Such simplifications can be achieved by atom-and-variable removal abstractions. In the full version [39], we present an algorithm that identifies unnecessary atoms and variables, and removes them from the protocol messages.

5.1.3. Removing redundant terms based on preceding intruder knowledge

A somewhat related case occurs for terms in a protocol message m that the intruder can derive from his previous knowledge. A sufficient condition for this is that they can be derived from the combination of the initial intruder knowledge and the messages sent before m in the same role. As before, they might be redundant and we generate an abstraction in which they are removed from the protocol messages. In the full version [39], we explain how to eliminate such redundancies using redundancy removal abstractions.

5.2. Checking for spurious attacks

Our abstractions are sound, but not complete. Therefore, we may encounter false negatives, i.e., spurious attacks. To check whether an attack on a security property ϕ in an abstract model corresponds to a real attack in the original one, we perform the following steps. First, for each thread in the attack trace, we construct a (symbolic) trace whose events correspond to those occurring in the abstract thread. Then, we ask the verifier to search for an attack in the original protocol such that this attack contains only threads that are computed in the previous step. Formally, let (tr, th, σ) be the state that is corresponding to the attack found in the abstract model and $ID \subseteq TID$ be the set of thread identifiers in tr. For each $i \in ID$, let e_i be the last event of thread i in tr, e'_i be the corresponding event in the original protocol description, and let tr_i be the symbolic trace such that $tr_i = (i, ev_1) \cdot (i, ev_2) \cdots (i, ev_m)$, where ev_j is the j-th event in the role $P(\pi_1(th(i)))$ of the original protocol P and $ev_m = e'_i$. Intuitively, tr_i is the original symbolic trace corresponding to the abstract trace obtained by projecting the attack trace tr to thread i's events. The verifier checks whether there exists a concrete attack consisting only of the events in the traces tr_i for $i \in ID$.

6. Implementation and case studies

In this section, we explain how we have implemented our abstraction mechanism for the Scyther tool. The resulting tool is available online [36]. We then validate the effectiveness of our method on a large number of real-world case studies.

6.1. Implementation for the Scyther tool

Scyther [14] is a leading automated security protocol verification tool. It supports verification for both a bounded and an unbounded number of threads. It also supports multi-protocol analysis, i.e., verifying

a composition of multiple protocols. Scyther takes as input a security protocol description specified by a set of linear role scripts, which include the intended security properties. The tool supports both user-defined types and hash functions. These features match our setting very well.

In this section, we first present the correspondence between claim events in Scyther and our security property formulas. Then, we describe an extension of the labeling mechanism and the abstraction heuristics. In the full version [39], we demonstrate the application of our abstraction heuristics on an example.

6.1.1. Claim events and security properties

In Scyther, security properties are specified by means of *claim events*, which are integrated into protocol role specifications. Intuitively, claim events express the intended security goal that an agent executing a given protocol role expects to achieve. For our implementation, we consider the following types of claim events that are used to express secrecy and various forms of authentication properties. We adopt the definitions of these properties from [13,14,32]. All these properties include the additional premise that both the agent owning the thread executing the claim and its (intended) communication partner are honest, which we do not repeat below.

1. claim(A, Secret, t) expresses the *secrecy* of a term t for role A, i.e., whenever an agent a executes a role A thread up to the claim event, term t cannot be derived by the adversary.
2. claim(A, Alive) expresses the *aliveness* property for role A, i.e., whenever an agent a executes a role A thread up to the claim event, apparently with an agent b, then b has previously been running a protocol thread.
   Note that this property still holds even when b was running the protocol with someone else (not a). Strengthening aliveness leads us to the notion of *weak agreement* property.
3. claim(A, Weakagree) expresses *weak agreement* property for role A, i.e., whenever an agent a executes a role A thread to the claim event, apparently with an agent b, then b has previously been running a protocol thread, apparently with a.
   Neither aliveness nor weak agreement guarantee that agents agree on their respective roles or on any data exchanged. This additional requirement is captured by *non-injective agreement*.
4. claim(A, Commit, B, m) and claim(B, Running, A, m') are used to formalize *non-injective agreement* as defined by Lowe [32]. We say that a protocol guarantees non-injective agreement for role A with role B on a message m if, whenever a executes a role A thread up to the Commit claim event, apparently with b in role B, then b has previously run a role B thread (at least) up to the Running claim, apparently with a in role A, and the instances of m and m' agree according to the local views of these two agents' threads.
5. claim(A, Niagree) expresses another form of non-injective agreement stating that role A satisfies non-injective agreement if for each role A thread reaching the claim in some trace, there exist threads for all other roles of the protocol, such that all events causally preceding the claim (according to the protocol specification) must have occurred before the claim (in the trace) and each pair of matching send and receive events agree on the messages they contain.
6. claim(A, Nisynch) expresses the *non-injective synchronization* property. This claim strengthens claim(A,Niagree) by additionally requiring that the order of the events preceding claim(A,Nisynch) must be correct as found in the protocol description, i.e., the send events occur before the corresponding receive events.

Note that non-injective agreement specified by claim(A, Niagree) is different from that specified by the Running and Commit signals. The property does not require agreement on a specified set of data

values. Instead, it requires agreement on the messages exchanged between the agents, which implies agreement on the data contained in those messages.

We now explain how to formalize these properties in our security property language using an example.

Example 6.1. Consider the Needham–Schroeder public-key (NSPK) protocol from [35]. We mimic the claim events by introducing the corresponding signal events with the following set of signals:

$$Sig = \{\mathsf{Create}, \mathsf{Secret}, \mathsf{Alive}, \mathsf{Weakagree}, \mathsf{Commit}, \mathsf{Running}, \mathsf{Niagree}, \mathsf{Nisynch}\}.$$

The signal event Create models the creation of a new protocol thread, which mimics the semantics of the Create event defined in [13, page 27]. The remaining signals represent the corresponding claim events. Our formalization of the Needham–Schroeder public-key protocol is now given as follows.

$$\begin{aligned} NS(A) = {} & \mathsf{Create} \cdot \mathsf{send}(\{A, na\}_{\mathsf{pk}(B)}) \cdot \mathsf{recv}(\{na, Nb\}_{\mathsf{pk}(A)}) \cdot \mathsf{Running} \cdot \mathsf{send}(\{Nb\}_{\mathsf{pk}(B)}) \cdot \\ & \mathsf{Commit} \cdot \mathsf{Secret} \cdot \mathsf{Alive} \cdot \mathsf{Weakagree} \cdot \mathsf{Niagree} \cdot \mathsf{Nisynch} \\ NS(B) = {} & \mathsf{Create} \cdot \mathsf{recv}(\{A, Na\}_{\mathsf{pk}(B)}) \cdot \mathsf{Running} \cdot \mathsf{send}(\{Na, nb\}_{\mathsf{pk}(A)}) \cdot \mathsf{recv}(\{nb\}_{\mathsf{pk}(B)}) \cdot \\ & \mathsf{Commit} \cdot \mathsf{Secret} \cdot \mathsf{Alive} \cdot \mathsf{Weakagree} \cdot \mathsf{Niagree} \cdot \mathsf{Nisynch} \end{aligned}$$

We formalize the secrecy, aliveness, weak agreement, non-injective agreement, and non-injective synchronization properties for role A as follows.

1. **Secrecy** of *na*:

$$\begin{aligned} \phi^{NS}_{sec} = {} & \forall \iota.\, (role(\iota, A) \wedge honest(\iota, \{A, B\}) \wedge steps(\iota, \mathsf{Secret})) \\ & \Rightarrow secret(\iota, na) \end{aligned}$$

2. **Aliveness**:

$$\begin{aligned} \phi^{NS}_{\mathsf{alive}} = {} & \forall \iota.\, (role(\iota, A) \wedge honest(\iota, \{A, B\}) \wedge steps(\iota, \mathsf{Alive})) \\ & \Rightarrow (\exists \kappa.\, steps(\kappa, \mathsf{Create}) \wedge \\ & \quad ((role(\kappa, A) \wedge A^{@\kappa} = B^{@\iota}) \vee \\ & \quad (role(\kappa, B) \wedge B^{@\kappa} = B^{@\iota}))) \end{aligned}$$

3. **Weak agreement**:

$$\begin{aligned} \phi^{NS}_{\mathsf{wagree}} = {} & \forall \iota.\, (role(\iota, A) \wedge honest(\iota, \{A, B\}) \wedge steps(\iota, \mathsf{Weakagree})) \\ & \Rightarrow (\exists \kappa.\, steps(\kappa, \mathsf{Create}) \wedge \\ & \quad ((role(\kappa, A) \wedge A^{@\kappa} = B^{@\iota} \wedge B^{@\kappa} = A^{@\iota}) \vee \\ & \quad (role(\kappa, B) \wedge B^{@\kappa} = B^{@\iota} \wedge A^{@\kappa} = A^{@\iota}))) \end{aligned}$$

4. **Non-injective agreement** (on na and nb) **based on Running and Commit claims**:

$$\begin{aligned}\phi^{NS}_{\mathsf{cm}} = \forall \iota.\, (&role(\iota, A) \wedge honest(\iota, \{A, B\}) \wedge steps(\iota, \mathsf{Commit})) \\ \Rightarrow\, (&\exists \kappa.\, role(\kappa, B) \wedge steps(\kappa, \mathsf{Running}) \wedge \\ &\langle A, B, na, Nb\rangle^{@\iota} = \langle A, B, Na, nb\rangle^{@\kappa})\end{aligned}$$

5. **Non-injective agreement specified by** $\mathsf{claim}(A, \mathsf{Niagree})$:

$$\begin{aligned}\phi^{NS}_{\mathsf{niagree}} = \forall \iota.\, (&role(\iota, A) \wedge honest(\iota, \{A, B\}) \wedge steps(\iota, \mathsf{Niagree})) \\ \Rightarrow\, (&\exists \kappa.\, role(\kappa, B) \wedge \\ &steps(\kappa, \mathsf{recv}(\{A, Na\}_{\mathsf{pk}(A)})) \prec steps(\iota, \mathsf{Niagree}) \wedge \\ &steps(\kappa, \mathsf{send}(\{Na, nb\}_{\mathsf{pk}(A)})) \prec steps(\iota, \mathsf{Niagree}) \wedge \\ &\langle A, B\rangle^{@\iota} = \langle A, B\rangle^{@\kappa} \wedge \\ &(\{A, na\}_{\mathsf{pk}(B)})^{@\iota} = (\{A, Na\}_{\mathsf{pk}(B)})^{@\kappa} \wedge \\ &(\{na, Nb\}_{\mathsf{pk}(A)})^{@\iota} = (\{Na, nb\}_{\mathsf{pk}(A)})^{@\kappa})\end{aligned}$$

6. **Non-injective synchronization**:

$$\begin{aligned}\phi^{NS}_{\mathsf{nisyn}} = \forall \iota.\, (&role(\iota, A) \wedge honest(\iota, \{A, B\}) \wedge steps(\iota, \mathsf{Nisynch})) \\ \Rightarrow\, (&\exists \kappa.\, role(\kappa, B) \wedge \\ &steps(\iota, \mathsf{send}(\{A, na\}_{\mathsf{pk}(B)})) \prec steps(\kappa, \mathsf{recv}(\{A, Na\}_{\mathsf{pk}(B)})) \wedge \\ &steps(\kappa, \mathsf{send}(\{Na, nb\}_{\mathsf{pk}(A)})) \prec steps(\iota, \mathsf{recv}(\{na, Nb\}_{\mathsf{pk}(A)})) \wedge \\ &\langle A, B\rangle^{@\iota} = \langle A, B\rangle^{@\kappa} \wedge \\ &(\{A, na\}_{\mathsf{pk}(B)})^{@\iota} = (\{A, Na\}_{\mathsf{pk}(B)})^{@\kappa} \wedge \\ &(\{na, Nb\}_{\mathsf{pk}(A)})^{@\iota} = (\{Na, nb\}_{\mathsf{pk}(A)})^{@\kappa})\end{aligned}$$

The last two properties are obtained by instantiating the general definitions from [13] for the A role of the Needham–Schroeder public-key protocol. To see that $\phi^{NS}_{\mathsf{nisyn}}$ strengthens $\phi^{NS}_{\mathsf{niagree}}$, note that the event ordering predicates in the latter formula are implied by those in the former together with event orderings within roles A and B, which always hold.

6.1.2. An extension of the labeling mechanism and the abstraction heuristics

In practice, it turns out that the labeling mechanism previously described is not sufficient to achieve good abstractions. There are protocols that employ cryptographic primitives in particular ways to achieve certain security goals, even though these primitives do not provide the desired properties themselves. In such cases, the heuristic may assign security labels to terms incorrectly, or accidentally remove elements that are important to achieve these properties.

Example 6.2. Let us come back to the NSPK protocol, specified (without signals) as:

$$NS(A) = \mathsf{send}(\{A, na\}_{\mathsf{pk}(B)}) \cdot \mathsf{recv}(\{na, Nb\}_{\mathsf{pk}(A)}) \cdot \mathsf{send}(\{Nb\}_{\mathsf{pk}(B)})$$
$$NS(B) = \mathsf{recv}(\{A, Na\}_{\mathsf{pk}(B)}) \cdot \mathsf{send}(\{Na, nb\}_{\mathsf{pk}(A)}) \cdot \mathsf{recv}(\{nb\}_{\mathsf{pk}(B)})$$

Suppose that we are interested in non-injective agreement for an agent in role A with an agent in role B on the nonce na. The agent variable A in the first sent message is crucial to achieve this property. However, our heuristic may pull A out of the messages $\{A, na\}_{\mathsf{pk}(B)}$ and $\{A, Na\}_{\mathsf{pk}(B)}$, as this abstraction preserves the label NO for authentication and confidentiality of A. It is not hard to see that the resulting abstracted protocol no longer provides the desired property. Furthermore, the heuristic incorrectly decides that na has authentication label NO. Thus, we may also pull na out of the encryptions in the first two events of role A, as this abstraction clearly preserves the security label of na. However, no authentication is guaranteed for the abstracted protocol.

To deal with this issue, we enable the heuristic to detect such a pattern, i.e., an asymmetric encryption that includes an agent identity which is different from the one indicated in the encryption key. In this case, at least one occurrence of the identity must be kept, and the encryption is associated with authentication label YES. Similarly, we must also keep agent identities that occur in symmetric encryptions.

6.2. Experimental results

We have validated the effectiveness of our abstractions on a total of 24 members of the IKE and ISO/IEC 9798 protocol families and on the PANA-AKA protocol [4] and the KSL protocol. We verify these protocols using five tools based on four different techniques: Scyther [14], CL-Atse [47], OFMC [8], SATMC [6], and ProVerif [9]. Only Scyther and ProVerif support verification of an unbounded number of threads. In Table 3, we present a selection of the experimental results for Scyther and refer to the full version [39] for a complete account, including results for the other tools for which we used hand-crafted abstractions. While our execution model closely fits Scyther's, there are subtle differences with the execution models and specification languages of the other tools. However, our initial results suggest that our techniques can be formally adapted to increase the efficiency of those tools as well. Our models of the IKE and ISO/IEC 9798 protocols are based on Cremers' [11,12]. Since Scyther uses a fixed signature with standard cryptographic primitives and no equational theories, the IKE models approximate the DH equational theory by oracle roles.

For our case studies, we verify several security properties including secrecy, aliveness, weak agreement, and non-injective agreement. We mark verified properties by ✓ and falsified ones by ×. An entry ✓/× means the property holds for one role but not for the other. Each row consists of two lines, corresponding to the analysis time without (line 1) and with (line 2) abstraction for 3-8 or unboundedly many (∞) threads. The times were measured on a cluster of 12-core AMD Opteron 6174 processors with 64 GB RAM each. They include computing the abstractions (4-20 ms) and the verification itself.

Verification. For 13 of the 19 original protocols that are analyzed, an unbounded verification attempt results in a timeout (TO = 8h cpu time) or memory exhaustion (ME). In 7 of these, our abstractions enabled the verification of all properties in less than 0.4 seconds and in one case in 78 seconds. However, for the first three protocols, we still get a timeout. For the large majority of the bounded verification tasks, we significantly push the bound on the number of threads and achieve massive speedups. For example, our abstractions enable the verification of the complex nested protocols IKEv2-eap and PANA-AKA.

Table 3

Experimental results for Scyther. The time is in seconds. **No**: Number of abstractions. Properties: **S**ecrecy, **A**liveness, **W**eak agreement, and **N**on-injective agreement

Protocol	No	Properties				Number of threads						
		S	*A*	*W*	*N*	3	4	5	6	7	8	∞
IKE												
IKEv1-pk2-a2	1	✓			✓	40.25	302.21	1679.69	9947.75	TO	TO	TO
						6.12	26.40	154.26	959.02	6412.25	TO	TO
IKEv1-pk-a22	1	✓			✓	15.14	80.80	244.45	530.94	979.88	1677.69	TO
						0.95	1.44	2.36	4.00	7.54	10.37	TO
IKEv2-eap	5	✓			✓	TO	TO	TO	TO	TO	TO	TO
						78.94	773.49	4345.58	18572.70	TO	TO	TO
IKEv2-mac	4	✓			✓	1.82	5.13	6.21	7.52	8.30	8.59	8.69
						0.70	1.58	1.72	1.72	1.72	1.71	1.72
IKEv2-mactosig	6	✓			✓	13.29	135.64	1076.56	7389.01	TO	TO	TO
						2.68	12.38	24.54	38.68	53.36	65.07	77.68
IKEv2-sigtomac	6	✓			✓	6.11	26.18	65.61	137.53	165.84	206.29	238.28
						1.70	7.78	28.44	44.44	55.11	66.97	67.15
IKEv1-pk-m	2				×	48.62	269.92	507.40	869.23	16254.80	TO	TO
						0.16	0.22	0.37	0.66	1.19	2.05	TO
IKEv1-pk-m2	2				✓/×	12.94	178.49	2198.81	TO	TO	TO	TO
						0.21	0.30	0.26	0.28	0.30	0.35	TO
IKEv1-sig-m	2				×	0.35	0.45	0.45	0.45	0.45	0.46	0.45
						0.35	0.33	0.34	0.34	0.34	0.35	0.39
IKEv1-sig-m-perlman	2				×	3.55	14.11	47.16	67.61	72.20	72.15	73.83
						17.59	17.61	17.53	17.53	17.59	17.53	17.58
IKEv2-sig-child	6		✓	✓	✓/×	235.11	11274.66	TO	TO	TO	TO	TO
						38.04	462.53	874.21	17713.06	TO	TO	TO
ISO/IEC												
ISO/IEC 9798-2-5	1	✓				0.79	9.12	72.75	557.77	4260.57	TO	TO
						0.07	0.11	0.12	0.11	0.11	0.11	0.11
ISO/IEC 9798-2-6	1	✓				0.59	3.82	18.84	67.38	197.42	575.42	21254.67
						0.05	0.04	0.05	0.05	0.05	0.05	0.05
ISO/IEC 9798-3-6-1	2		✓		✓	42.68	795.11	8915.40	ME	ME	ME	ME
						0.14	0.20	0.21	0.21	0.21	0.21	0.21
ISO/IEC 9798-3-6-2	1		✓		✓	2.47	8.66	19.48	33.94	48.26	60.05	70.81
						0.12	0.15	0.15	0.15	0.15	0.15	0.15
ISO/IEC 9798-3-7-1	2		✓		✓	41.63	752.82	7769.87	15863.97	ME	ME	ME
						0.15	0.20	0.21	0.21	0.21	0.21	0.21
ISO/IEC 9798-3-7-2	1		✓		✓	2.46	7.97	16.93	26.41	34.67	50.30	TO
						0.21	0.30	0.31	0.31	0.31	0.31	0.31
Others												
PANA-AKA	7	✓	✓	✓	✓	5762.53	TO	TO	TO	TO	TO	TO
						0.23	0.22	0.23	0.23	0.23	0.23	0.23
KSL	1	✓				17.81	1272.50	TO	TO	TO	TO	TO
						0.03	0.03	0.03	0.03	0.03	0.03	0.03

Scyther verifies an abstraction of IKEv2-eap for up to 6 threads and, more strikingly, completes the unbounded verification of the simplified PANA-AKA in under 0.3 seconds whereas it can handle only 4 threads of the original version.

For these protocols, our tool aggressively simplifies the original models by removing unnecessary cryptographic protections and redundant fields. The IKEv2-eap protocol consists of two roles exchanging 8 messages. The messages are large and contain up to 5 layers of cryptographic operations (such as encryptions, signatures, and hashes). However, the most abstract model generated by our tool only exchanges 5 messages (i.e., 3 messages are completely removed by untyped abstractions). The most deeply nested messages contain only 3 layers of cryptographic operations. The PANA-AKA protocol exhibits a similar complexity. It employs up to 6 layers of cryptographic operations. Even though the most abstract model for PANA-AKA still exchanges 7 messages, the messages are substantially smaller than those of the original model and use at most 3 layers of cryptographic operations. We also achieve dramatic speedups for many other protocols, most notably for IKEv1-pk-a22, ISO/IEC 9798-2-6, and ISO/IEC 9798-3-6-2. This shows that our abstractions work particularly well for protocols that have complex message structures or large numbers of exchanged messages, as these features can significantly deteriorate the performance of protocol verifiers.

More interestingly, our abstractions also perform very well on another class of protocols which have simple message structures but still render verification challenging. For example, the ISO/IEC 9798-3-6-1, ISO/IEC 9798-3-7-1 and KSL protocols contain relatively small messages with at most one layer of encryption. However, the verification attempts for the original versions of the ISO/IEC 9798-3-6-1 and ISO/IEC 9798-3-7-1 protocols both result in memory exhaustion after 7 threads. Similarly, the verification of KSL already times out for 5 threads. We attribute this difficulty to the presence of untyped variables, i.e., variables of type *msg* in our type system, in clear texts. As there is no constraint on the shapes of the messages that can be used to instantiate these variables, protocol verifiers typically need to consider all possible forms of instantiations, which potentially results in performance degradation. By removing unnecessary occurrences of untyped variables with respect to the security properties of interest, our abstractions enable the verification of KSL for an unbounded number of threads in only 0.03 seconds. Analogously, the tool successfully verifies ISO/IEC 9798-3-6-1 and ISO/IEC 9798-3-7-1 for an unbounded number of threads in 0.21 seconds.

Apart from enormous performance gains, the speedup is more modest for a few protocols, e.g., IKEv1-pk2-a2, IKEv2-sigtomac, and IKEv2-mac. These protocols have simple message structures, e.g., using at most 3 layers of cryptographic operations and only up to 4 exchanged messages. Moreover, they use untyped variables only in protected positions, i.e., as arguments of a hash or an encryption. They therefore do not leave much room for abstractions. In fact, although the generated abstract models for these protocols have smaller message sizes, they have similar message structures compared to the original ones. Nevertheless, our abstractions enable the reduction of the verification time by an order of magnitude in some cases, e.g., for the IKEv1-pk2-a2 protocol.

Additionally, we observe that the verification time for many abstracted protocols increases much more slowly than for their originals as the number of threads increases. We obtain almost constant verification times for the six ISO/IEC 9798 protocols, whereas the time significantly increases on some originals, e.g., for the ISO/IEC 9798-3-6-1 protocol.

Falsification. For rows marked by $\times$, the second line corresponds to falsification time for the most abstract model, which is much faster than on the original one. For example, for 8 threads of the IKEv1-pk-m protocol, we reduce falsification time from a timeout to 2.05 seconds. Note that for falsification, a check for spurious attacks is needed. This subroutine renders the performance gains less substantial

than that for verification. For instance, in the unbounded case, the speedup factors are 1.15 for IKEv1-sig-m and 4.19 for IKEv1-sig-m-perlman. Note that our tool automatically checks for spurious attacks. Interestingly, all attacks found in the most abstract protocols are real, suggesting that our measures to prevent spurious attacks are effective.

Combination. For the IKEv1-pk-m2 and IKEv2-sig-child protocols, the tool verifies non-injective agreement for one role and falsifies it for the other one. Analogous to other case studies, we obtain a remarkable speedup for these protocols. Our abstractions raise the feasibility bound by 2 to 3 additional threads.

7. Related work

Hui and Lowe [28] define several kinds of abstractions similar to ours with the aim of improving the performance of the CASPER/FDR verifier. They establish soundness only for ground messages and encryption with atomic keys. We work in a more general model, cover additional properties, and treat the non-trivial issue of abstracting the open terms in protocol specifications. Other works [17,18,41] also propose a set of syntactic transformations, however without formally establishing their soundness. Using our results, we can, for instance, justify the soundness of the refinements in [18, Section 3.3].

Backes et al. [7] study the abstraction of authentication protocols formalized in the ρ-spi calculus. They propose a static analysis for authentication protocols by abstracting challenge-response messages into non-cryptographic versions expressed in a different language, called the CR calculus. Their abstraction method is based on non-increasing security labels similar to those of our heuristics. However, there are several differences with our work. First, since their sound abstractions map protocol specifications to a different language, the abstract protocols cannot be further abstracted. In our setting, protocol specifications and abstract protocols are expressed in the same language and abstractions can be composed. Second, the construction of the abstractions requires the identification of challenge-response components of a protocol, for which they do not give an algorithm. Third, since they designed a specialized technique for proving authentication properties, they cannot employ existing protocol verification tools to verify the abstract protocols. In contrast, our abstractions are composable, computed automatically by our tool, and can be verified using standard protocol verifiers. Finally, their method is restricted to agreement properties, while ours supports an expressive property specification language, which covers secrecy and a variety of authentication properties.

Guttman [24,25] studies the preservation of security properties for a rich class of protocol transformations in the strand space model. His approach to property preservation is based on the simulation of protocol analysis steps instead of execution steps. Each such analysis step explains the origin of a message. Apart from this different approach to soundness, there are other differences with our work. First, instead of working at the level of protocol messages, his protocol transformations are applied to strand space nodes and then lifted to protocol specifications and security properties. In contrast to our work, his approach does not restrict the shape of the transformed protocol message with respect to the original message. In his theory, one can, for instance, transform a hash of a message X and a key K into an encryption of X with K. We do not support such general transformations. Second, his protocol transformations are required to preserve the origination of values and the plaintext subterms of messages. The former condition means that if a value x first occurs in a transmission node then it also occurs first in the corresponding transformed node. Our soundness results do not require such conditions. For example, we can completely remove fresh values that are in clear or fields in a hash. Third, since his primary

focus was to set up a general framework to express and justify security protocol transformations, he does not provide syntactic soundness conditions, guidance for the choice of appropriate abstractions, or automated verification. It might be possible to identify a subset of his transformations for which this is possible, but this would require additional work. In contrast, our tool automatically determines suitable abstractions and checks their soundness.

Refinement is abstraction viewed in the reverse direction, i.e., from abstract to concrete. Sprenger et al. [31,45,46] have proposed a hierarchical development method for security protocols based on stepwise refinement that spans several levels of abstraction. Each development starts from abstract models of security properties and proceeds down to cryptographic protocols secure against a Dolev–Yao intruder. The development process traverses intermediate levels of abstraction based on message-less protocols and communication channels with authenticity and confidentiality properties. Security properties, once proved for a given model, are preserved by further refinements. They have applied their method to develop families of authentication and key transport protocols. The abstractions in the present paper belong to their most concrete level of cryptographic protocols. They have embedded their approach in the Isabelle/HOL theorem prover, but each refinement step essentially requires a separate soundness proof.

8. Conclusions

In this work, we propose a set of syntactic protocol transformations that allows us to abstract realistic protocols and capture a large class of attacks. Unlike previous work [28,37], our theory and soundness results accommodate equational theories and a fine-grained type system that supports untyped variables, user-defined types, and subtyping. These features allow us to accurately model protocols, capture type-flaw attacks, and adapt to different verification tools, e.g., those supporting equational theories such as ProVerif and CL-Atse. We have extended Scyther with an abstraction module, which we validated on various IKE and ISO/IEC 9798 protocols and others. We also tested our technique (with manually produced abstractions) on ProVerif, CL-Atse, OFMC, and SATMC. Our experiments show that modern protocol verifiers can substantially benefit from our abstractions, which often either enable previously infeasible verification tasks or lead to dramatic speedups. Our abstraction tool supports checking for spurious attacks, which allows us to not only verify but also falsify security protocols efficiently.

As for future work, we plan to extend our soundness results to more expressive security protocol models such as multiset rewriting. This would allow us to cover more security protocols, for instance, protocols involving loops such as the TESLA protocol [42] or non-monotonic states such as contract signing protocols [3], as well as more security properties and adversary capabilities such as perfect forward secrecy, key compromise impersonation, and adversaries capable of revealing the local state of agents. We believe that our soundness results can also be extended to support else-branches in such theories by additionally establishing preservation theorems for disequality tests. Another direction for future research could be to generalize the tool and support more protocol verifiers. Possible improvements might be gained from applying techniques from the field of counter-example guided refinement: when a spurious attack is found, it might be possible to extract information from it to guide the exploration of the generated abstractions.

Acknowledgments

We thank Mathieu Turuani and Michael Rusinowitch for our fruitful technical discussions on the topic of this paper. We are also grateful to David Basin, Ognjen Maric, Ralf Sasse, and the anonymous

reviewers for their careful proof-reading and helpful suggestions. This work was partially supported by the Air Force Office of Scientific Research, grant number FA9550-17-1-0206, and the EU FP7-ICT-2009 Project No. 256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

References

[1] O. Almousa, S.A. Mödersheim, P. Modesti and L. Viganò, Typing and compositionality for security protocols: A generalization to the geometric fragment, in: *ESORICS*, Lecture Notes in Computer Science, Springer, 2015.
[2] M. Arapinis and M. Duflot, Bounding messages for free in security protocols, in: *FSTTCS*, 2007, pp. 376–387.
[3] M. Arapinis, E. Ritter and M.D. Ryan, StatVerif: Verification of stateful processes, in: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011*, Cernay-la-Ville, France, 27–29 June, 2011, IEEE Computer Society, 2011, pp. 33–47. doi:10.1109/CSF.2011.10.
[4] J. Arkko and H. Haverinen, RFC 4187: Extensible authentication protocol method for 3rd generation authentication and key agreement (EAP-AKA), 2006, http://www.ietf.org/rfc/rfc4187.
[5] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S.E. Ponta, M. Rocchetto, M. Rusinowitch, M.T. Dashti, M. Turuani and L. Viganò, The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures, in: *TACAS*, 2012, pp. 267–282.
[6] A. Armando and L. Compagna, SAT-based model-checking for security protocols analysis, *International Journal of Information Security* **7**(1) (2008), 3–32. doi:10.1007/s10207-007-0041-y.
[7] M. Backes, A. Cortesi, R. Focardi and M. Maffei, A calculus of challenges and responses, in: *Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering, FMSE '07*, ACM, New York, NY, USA, 2007, pp. 51–60. ISBN 978-1-59593-887-9. doi:10.1145/1314436.1314444.
[8] D.A. Basin, S. Mödersheim and L. Viganò, OFMC: A symbolic model checker for security protocols, *Int. J. Inf. Sec.* **4**(3) (2005), 181–208. doi:10.1007/s10207-004-0055-7.
[9] B. Blanchet, An efficient cryptographic protocol verifier based on prolog rules, in: *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, Cape Breton, Nova Scotia, Canada, 11–13 June 2001, IEEE Computer Seciety, 2001, pp. 82–96. doi:10.1109/CSFW.2001.930138.
[10] P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: *POPL*, 1977, pp. 238–252.
[11] C. Cremers, IKEv1 and IKEv2 protocol suites, 2011, https://github.com/cascremers/scyther/tree/master/gui/Protocols/IKE.
[12] C. Cremers, ISO/IEC 9798 authentication protocols, 2012, https://github.com/cascremers/scyther/tree/master/gui/Protocols/ISO-9798.
[13] C. Cremers and S. Mauw, *Operational Semantics and Verification of Security Protocols*, Information Security and Cryptography, Springer, 2012. ISBN 978-3-540-78636-8. doi:10.1007/978-3-540-78636-8.
[14] C.J.F. Cremers, The Scyther tool: Verification, falsification, and analysis of security protocols, in: *CAV*, 2008, pp. 414–418.
[15] C.J.F. Cremers, Key exchange in IPsec revisited: Formal analysis of IKEv1 and IKEv2, in: *ESORICS*, 2011, pp. 315–334.
[16] C.J.F. Cremers, S. Mauw and E.P. de Vink, Injective synchronisation: An extension of the authentication hierarchy, *Theor. Comput. Sci.* **367**(1–2) (2006), 139–161. doi:10.1016/j.tcs.2006.08.034.
[17] A. Datta, A. Derek, J.C. Mitchell and D. Pavlovic, Abstraction and refinement in protocol derivation, in: *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*, 2004.
[18] A. Datta, A. Derek, J.C. Mitchell and D. Pavlovic, A derivation system and compositionl logic for security protocols, *Journal of Computer Security* **13** (2005), 423–482. doi:10.3233/JCS-2005-13304.
[19] D. Dolev and A.C. Yao, On the security of public key protocols, *IEEE Transactions on Information Theory* **29**(2) (1983), 198–207. doi:10.1109/TIT.1983.1056650.
[20] F. Durán and J. Meseguer, A Church-Rosser checker tool for conditional order-sorted equational maude specifications, in: *Rewriting Logic and Its Applications – 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Revised Selected Papers*, Paphos, Cyprus, March 20–21, 2010, pp. 69–85. doi:10.1007/978-3-642-16310-4_6.
[21] S. Escobar, C. Meadows and J. Meseguer, Maude-NPA: Cryptographic protocol analysis modulo equational properties, in: *FOSAD*, 2007, pp. 1–50.
[22] S. Escobar, R. Sasse and J. Meseguer, Folding variant narrowing and optimal variant termination, *J. Log. Algebr. Program.* **81**(7–8) (2012), 898–928. doi:10.1016/j.jlap.2012.01.002.

[23] J. Giesl, P. Schneider-Kamp and R. Thiemann, Automatic termination proofs in the dependency pair framework, in: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Proceedings*, Seattle, WA, USA, August 17–20, 2006, 2006, pp. 281–286. doi:10.1007/11814771_24.
[24] J.D. Guttman, Transformations between cryptographic protocols, in: *ARSPA-WITS*, 2009, pp. 107–123.
[25] J.D. Guttman, Security goals and protocol transformations, in: *Theory of Security and Applications (TOSCA), an ETAPS Associated Event*, LNCS, Vol. 6993, Springer, 2011.
[26] J.D. Guttman, Establishing and preserving protocol security goals, *Journal of Computer Security* **22**(2) (2014), 203–268. doi:10.3233/JCS-140499.
[27] D. Harkins and D. Carrel, The Internet key exchange (IKE), IETF RFC 2409 (proposed standard), 1998, Obsoleted by RFC 4306, updated by RFC 4109, http://www.ietf.org/rfc/rfc2409.txt.
[28] M.L. Hui and G. Lowe, Fault-preserving simplifying transformations for security protocols, *Journal of Computer Security* **9**(1/2) (2001), 3–46. doi:10.3233/JCS-2001-91-202.
[29] J.-P. Jouannaud and H. Kirchner, Completion of a set of rules modulo a set of equations, *SIAM J. Comput.* **15**(4) (1986), 1155–1194. doi:10.1137/0215084.
[30] C. Kaufman, P. Hoffman, Y. Nir and P. Eronen, Internet key exchange protocol version 2 (IKEv2), IETF RFC 5996, 2010, http://tools.ietf.org/html/rfc5996.
[31] J. Lallemand, D.A. Basin and C. Sprenger, Refining authenticated key agreement with strong adversaries, in: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017*, Paris, France, April 26–28, 2017, pp. 92–107. doi:10.1109/EuroSP.2017.22.
[32] G. Lowe, A hierarchy of authentication specifications, in: *IEEE Computer Security Foundations Workshop*, IEEE Computer Society, Los Alamitos, CA, USA, 1997, pp. 31–43. doi:10.1109/CSFW.1997.596782.
[33] S. Meier, C.J.F. Cremers and D.A. Basin, Strong invariants for the efficient construction of machine-checked protocol security proofs, in: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*, Edinburgh, United Kingdom, July 17–19, 2010, IEEE Computer Seciety, 2010, pp. 231–245. doi:10.1109/CSF.2010.23.
[34] S. Meier, B. Schmidt, C. Cremers and D.A. Basin, The TAMARIN prover for the symbolic analysis of security protocols, in: *CAV*, 2013, pp. 696–701.
[35] R.M. Needham and M.D. Schroeder, Using encryption for authentication in large networks of computers, *Commun. ACM* **21**(12) (1978), 993–999. doi:10.1145/359657.359659.
[36] B.T. Nguyen, The Scyther-abstraction tool, 2018, https://github.com/binhnguyen1984/scyther-abstraction.
[37] B.T. Nguyen and C. Sprenger, Sound security protocol transformations, in: *POST*, 2013, pp. 83–104.
[38] B.T. Nguyen and C. Sprenger, Abstractions for security protocol verification, in: *Principles of Security and Trust – 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, Proceedings*, London, UK, April 11–18, 2015, R. Focardi and A.C. Myers, eds, Lecture Notes in Computer Science, Vol. 9036, Springer, 2015, pp. 196–215. doi:10.1007/978-3-662-46666-7_11.
[39] B.T. Nguyen, C. Sprenger and C. Cremers, Abstractions for security protocol verification, Technical report, Department of Computer Science, ETH Zurich, 2018. doi:10.3929/ethz-b-000266360.
[40] L. Paulson, The inductive approach to verifying cryptographic protocols, *J. Computer Security* **6** (1998), 85–128. doi:10.3233/JCS-1998-61-205.
[41] D. Pavlovic and C. Meadows, Deriving secrecy in key establishment protocols, in: *Proc. 11th European Symposium on Research in Computer Security (ESORICS)*, 2006, pp. 384–403.
[42] A. Perrig, J.D. Tygar, D. Song and R. Canetti, Efficient authentication and signing of multicast streams over Lossy channels, in: *Proceedings of the 2000 IEEE Symposium on Security and Privacy, SP '00*, IEEE Computer Society, Washington, DC, USA, 2000, p. 56. ISBN 0-7695-0665-8.
[43] S. Schneider, Verifying authentication protocols with CSP, in: *10th Computer Security Foundations Workshop (CSFW '97)*, Rockport, Massachusetts, USA, June 10–12, 1997, 1997, pp. 3–17. doi:10.1109/CSFW.1997.596775.
[44] S.A. Shaikh, V.J. Bush and S.A. Schneider, Specifying authentication using signal events in CSP, *Computers & Security* **28**(5) (2009), 310–324. doi:10.1016/j.cose.2008.10.001.
[45] C. Sprenger and D. Basin, Developing security protocols by refinement, in: *Proc. 17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 361–374. doi:10.1145/1866307.1866349.
[46] C. Sprenger and D. Basin, Refining key establishment, in: *Proc. 25th IEEE Computer Security Foundations Symposium (CSF)*, 2012, pp. 230–246.
[47] M. Turuani, The CL-Atse protocol analyser, in: *RTA*, 2006, pp. 277–286.