A Typing Result for Stateful Protocols

(Preprint)

Andreas Viktor Hess Sebastian Mödersheim

DTU Compute, Lyngby, Denmark

Version of April 11, 2018

Abstract

There are several typing results that, for certain classes of protocols, show it is without loss of attacks to restrict the intruder to sending only well-typed messages. So far, all these typing results hold only for relatively simple protocols that do not keep a state beyond single sessions, excluding stateful protocols that, e.g., maintain long-term databases. Recently, several verification tools for stateful protocols have been proposed, e.g., Set-π, AIF-ω, and SAPIC/Tamarin, but for none of these a typing result has been established. The main contribution of this paper is a typing result, for a large class of stateful protocols, based on a symbolic protocol model. We illustrate how to connect several formalisms for stateful protocols to this symbolic model. Finally, we discuss how the conditions of our typing result apply to existing protocols, or can be achieved by minor modifications.

1 Introduction

Many automated protocol verification methods [7, 8, 11, 27, 28] rely on a typed model in which the attacker can only send well-typed messages. Such a restriction to a typed model can also significantly reduce verification time and in some approaches [9, 5] protocol verification even becomes decidable in a typed model. There are in fact several results [1, 19, 18, 13, 24, 2] that show the *relative soundness* of a typed model if the protocol satisfies reasonable and sufficient conditions of a syntactic nature (i.e., can be checked without an exploration of the state space of the protocol). These *typing results* are of the form: if a protocol, that satisfies the sufficient conditions, has an attack then it has a well-typed attack, in which the attacker only sends well-typed messages. In other words, if the protocol is secure in a typed model then it is secure in an untyped model.

In a nutshell, when proving a typing result, one shows (for a given class of protocols) that from an ill-typed attack we can construct a similar well-typed attack, i.e., every ill-typed message that the intruder sends can be replaced

with a well-typed one so that all the remaining steps of the attack can still be performed in a similar way. To avoid messy and round-about arguments, all existing typing results argue via a constraint-based representation of the intruder. In these constraints, all messages sent and received by the intruder may contain variables where the corresponding honest agent would accept any value. Every attack is then a solution of such a constraint. There is a sound, complete, and terminating reduction procedure for such intruder constraints. It thus suffices to show that for the considered class of protocols, this reduction procedure will never instantiate any variable with a term of a different type than the variable has. If the procedure leaves any variables uninstantiated (i.e., its concrete value does not matter for the attack to work) then the intruder may as well choose a well-typed value here. This therefore allows to conclude that if there is a solution (i.e., attack), then there is a well-typed one. This can also be extended with equality and inequality constraints on messages, see e.g. [1].

All mentioned typing results, however, only apply to simple protocols in which agents do not maintain a global state, but have state only local to a single session, like a session key.[1] A more interesting and general class of protocols is one in which agents can additionally manipulate a global mutable state, e.g., maintain sets of public keys. In such protocols updating the global state during one session might influence other running sessions. We call such protocols *stateful*. Currently there exists several tools and approaches for verifying stateful protocols [17, 23, 25, 3, 20, 4, 21]. If we consider as a global state a database to which entries can be added (without bound) and deleted, and where negative checks are allowed (i.e., that no entry of a particular form is present), then this is not possible with a straightforward extension of existing typing results. While one could encode the positive operations and checks as special messages, the negative ones essentially amount to checking that a particular operation or message did not occur, and this negation is at odds with the intruder constraints needed to perform the main proof argument of the typing result.

The main contribution of this paper is a typing result for a large class of protocols with a global state that consists of a countable collection of sets, even when admitting deletion and negative checks. This is done in a precise and declarative way that uses existing typing results for stateless protocols as a basis. To have a simple and yet powerful formalism to work with, we introduce a notion of strands with set operations to model both honest agents and intruder constraints. In the intruder constraints, this represents at which point particular set operations occurred during an attack. We then show that we can reduce the satisfiability of these intruder constraints to the satisfiability of constraint systems without set operations. We can then make use of existing typing results.

A second contribution of this work is thus the formalisms with set operations for honest agents and for intruder constraints which are useful beyond the typing result to represent and work with stateful protocols. While this formalism is deliberately reduced to the essentials, we also show how to connect

[1] An exception is [24], but this paper contains some mistakes and their result does not hold in this generality as we explain in the appendix.

other more complex formalisms for stateful protocols, namely using rewriting and process calculi, so that our typing result can be also applied accordingly in these languages.

The paper is organized as follows. After preliminaries in section II we introduce in section III a new strand-based protocol model for stateful protocols. In section IV we extend intruder constraints with set operations and define a reduction mechanism on constraints that we prove sound and complete. We prove our main theorem, the typing result, in section V. Finally, we have case studies and connections to other formalisms in section VI and VII. All the proofs of our technical results are in the appendices.

2 Preliminaries

2.1 Term Algebra

We formally define a term algebra over a *signature* Σ containing symbols with associated arities and over a countable set of variables $\mathcal{V}$. The set of terms over Σ with variables from $\mathcal{V}$ is denoted by $\mathcal{T}_\Sigma(\mathcal{V})$ and we normally use the lower-case letters t, s, m, and e to denote arbitrary terms. A term $t \in \mathcal{T}_\Sigma(\mathcal{V})$ is then either a variable $t \in \mathcal{V}$ or a composed term of the form $f(t_1, \ldots, t_n)$ for some $f \in \Sigma$ of arity n and $t_i \in \mathcal{T}_\Sigma(\mathcal{V})$. When we later define our protocol model we will allow agents to send messages, modify sets, and emit events. We use terms to represent all of these different concepts and so we do not at this level distinguish between them.

The set Σ is partitioned into the *public symbols* Σ_{pub} (which the intruder has access to) and the *private symbols* Σ_{priv} (which the intruder cannot access). By Σ^n we denote the set of symbols of arity n. Similarly, Σ^n_{pub} (respectively Σ^n_{priv}) denotes the public (respectively private) symbols of arity n. The set of constants $\mathcal{C}$ is defined as Σ^0. The set of *free variables* $fv(t)$ of a term t is defined as usual and we say that t is *ground* iff $fv(t) = \emptyset$. As usual we extend fv to sets of terms. Constants will usually be denoted by the lower-case letters a, b, c, i, and k. We write f, g, and h as meta-variables ranging over non-constant symbols of Σ and we use sans serif to denote the actual elements of Σ, e.g., ring and crypt. We define substitutions as (finite or countably infinite) mappings from variables to terms, and we write $\delta(x)$ for the application of substitution δ to variable x. We usually use the letters δ and σ to denote substitutions. Substitutions are further extended to terms and sets of terms homomorphically as expected. The *domain* of a substitution δ is the set of variables which are not mapped to themselves: $dom(\delta) = \{x \in \mathcal{V} \mid \delta(x) \neq x\}$. The *image* of a substitution δ is then defined as usual: $img(\delta) = \delta(dom(\delta)) = \{\delta(x) \mid x \in dom(\delta)\}$ and we say that δ is *ground* when $fv(img(\delta)) = \emptyset$. For substitutions with finite domain we often write them as $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$. Note that we divert slightly from the conventional definition by also allowing for substitutions with infinite domain. Finally, a substitution δ is called a *unifier* of two terms t and t' iff $\delta(t) = \delta(t')$.

2.2 Intruder Model

We now define a Dolev-Yao style intruder deduction relation where $M \vdash t$ means that the intruder can derive the term t from the set of terms M called the *intruder knowledge*. Our model is similar to standard Dolev-Yao models but we parameterize ours over arbitrary signatures Σ instead of fixing a particular set of cryptographic primitives. Note also that we work in the free algebra; two terms are equal iff they are syntactically equal. For these reasons we additionally parameterize over an analysis theory Ana that serves as an *analysis interface*. For instance, to decrypt the message $\mathsf{crypt}(k, m)$ and obtain m we can require that the inverse key $\mathsf{inv}(k)$ must be provided, and we write $\mathsf{Ana}(\mathsf{crypt}(k, m)) = (\{\mathsf{inv}(k)\}, \{m\})$ to formally express this. Note that this would be similar to introducing a destructor dcrypt and an algebraic equation $\mathsf{dcrypt}(\mathsf{inv}(k), \mathsf{crypt}(k, m)) \approx m$ if we were not using the free algebra. More generally, if $\mathsf{Ana}(t) = (K, T)$ then the analysis of the term t results in the terms in T provided that all "keys" in K can be derived. Given such an Ana we define the deduction relation $\vdash$ as the least relation closed under the following rules:

$$\frac{}{M \vdash t}\ (Axiom),\ t \in M \qquad \frac{M \vdash t_1 \quad \cdots \quad M \vdash t_n}{M \vdash f(t_1, \ldots, t_n)}\ (Compose),\ f \in \Sigma^n_{pub}$$

$$\frac{M \vdash t \quad M \vdash k_1 \quad \cdots \quad M \vdash k_n}{M \vdash t_i}\ (Decompose), t_i \in T,\ K = \{k_1, \ldots, k_n\},\ \mathsf{Ana}(t) = (K, T)$$

Here, the $(Axiom)$ rule expresses that the intruder can derive any message in his knowledge. The rule $(Compose)$ allows the intruder to *compose* messages with any public symbol. For instance, if the intruder can derive a key k and a message m from a given intruder knowledge M (that is, $M \vdash k$ and $M \vdash m$) then he can asymmetrically encrypt m with k, i.e., $M \vdash \mathsf{crypt}(k, m)$. This rule also subsumes derivation of public constants, e.g., agent names. The final rule, $(Decompose)$, defines *decomposition* or *analysis* of terms, and it expresses that the intruder can decompose a derivable message t if he can derive the required keys K.

While the intruder deduction relation is defined for ground terms only, the analysis interface is defined on terms that might contain variables. The analysis interfaces we consider will be subject to some restrictions:

Ana_1: $\mathsf{Ana}(x) = (\emptyset, \emptyset)$ for variables $x \in \mathcal{V}$,

Ana_2: $\mathsf{Ana}(f(t_1, \ldots, t_n)) = (K, T)$ implies K is finite, $T \subseteq \{t_1, \ldots, t_n\}$, and $fv(K) \subseteq fv(f(t_1, \ldots, t_n))$, and

Ana_3: $\mathsf{Ana}(f(t_1, \ldots, t_n)) = (K, T)$ implies $\mathsf{Ana}(\delta(f(t_1, \ldots, t_n))) = (\delta(K), \delta(T))$.

The first requirement, Ana_1, ensures that variables cannot be decomposed. Ana_2 consists of two parts. First, all terms in T must be immediate subterms of the term being decomposed, and so the intruder cannot obtain any new terms by

decomposing something that he composed himself, and it is a technical requirement that is crucial in proofs of typing results. In fact, the typing result of [24] has a counter-example (Appendix B) because they lack this requirement. The second part restricts the set of keys K to be finite and to not introduce any new variables, but the keys are otherwise independent of the term being decomposed. This is useful when modeling asymmetric decryption as we can then require the intruder to derive the inverse key $\mathsf{inv}(k)$ of the key k used for the encryption. Ana_3 expresses that decomposition is invariant under substitution.

In concrete examples of this paper we use the following Ana theory on the usual set of cryptographic primitives: $\mathsf{Ana}(\mathsf{crypt}(k,m)) = (\{\mathsf{inv}(k)\},\{m\})$, $\mathsf{Ana}(\mathsf{scrypt}(k,m)) = (\{k\},\{m\})$, $\mathsf{Ana}(\mathsf{sign}(k,m)) = (\emptyset,\{m\})$, $\mathsf{Ana}(\langle t,t'\rangle) = (\emptyset,\{t,t'\})$ where $\langle\cdot,\cdot\rangle \in \Sigma^2$ is a pairing operator, and $\mathsf{Ana}(t) = (\emptyset,\emptyset)$ for all other terms t.

3 Stateful Protocols

In this section we will define our protocol model. There are several protocol models based on strands where protocol execution is defined in terms of a state transition system, e.g., [15, 1, 19]. In these works a state is a set (or multiset) of strands that represents the honest agents, and a representation of the intruder knowledge. We extend this model with strands that work with sets to model long-term mutable state information. Thus a distinguishing feature of our strands is that honest agents can query and update sets. A protocol state in our model will thus contain not only short-term session information but also the long-term contents of sets, and we call protocols based on these strands *stateful*.

3.1 Strands with Sets

We now define the syntax of *strands with sets* as an extension of [1] (the part of the syntax marked with $\star$ corresponds to the strands of [1]):

$$\begin{aligned}\ell \;&::=\; \phi.\ell \mid \overbrace{\psi.\ell \mid 0}^{\star}\\ &\text{with } \psi ::= \mathsf{send}(t) \mid \mathsf{receive}(t) \mid t \doteq t' \mid \forall\bar{x}.\; t \not\doteq t'\\ &\text{and } \phi ::= \mathsf{insert}(t,s) \mid \mathsf{delete}(t,s) \mid t \mathrel{\dot\in} s \mid \forall\bar{x}.\; t \mathrel{\dot\notin} s \mid\\ &\qquad\quad \mathsf{assert}(e) \mid \mathsf{event}(e) \mid \forall\bar{x}.\; \neg\mathsf{event}(e)\end{aligned}$$

where $t,t',s,e \in \mathcal{T}_\Sigma(\mathcal{V})$, and $\bar{x}$ ranges over finite sequences $x_1,\ldots,x_n$ of variables from $\mathcal{V}$. Strands built according to the above grammar but using only the cases marked with $\star$ are referred to as *ordinary strands*. A strand consists of a sequence of *steps* and we use here a process calculus notation where we delimit steps by periods and mark the end of a strand with a 0. We normally omit writing the end-marker 0 when it is obvious from the context. We will also omit writing the quantifier $\forall\bar{x}$ whenever $\bar{x}$ is the empty sequence.

The steps can be categorized into four parts: the message transmission steps (send and $\mathsf{receive}$), the equality checks ($\doteq$ and $\not\doteq$), the set operations (insert, delete, $\mathrel{\dot\in}$, and $\mathrel{\dot\notin}$), and the event steps (assert, event, and $\neg\mathsf{event}$). The most

basic ones are the message transmission steps which denote transmission over an insecure network. A $\mathsf{send}(t)$ step then means that an agent transmits t and $\mathsf{receive}(t)$ means that an agent waits for a *message pattern* (since it might contain variables) of the form t. Like [1], we extend strands with equalities and inequalities—they represent checks that must hold true to proceed—and a construct for emitting events; $\mathsf{assert}(e)$. We further extend strands with steps for checking whether an event has happened or not; $\mathsf{event}(e)$ and $\neg\mathsf{event}(e)$. Finally, the main novel addition to the concept of strands are the set operations. They allow for updates (insert and delete) and queries ($\dot{\in}$ and $\dot{\notin}$) of sets. Here, the delete operation allows for removal of elements that have previously been inserted into a set, and so the contents of sets do not grow monotonically during transitions. This is in contrast to the messages that the intruder has seen, i.e., the messages sent by honest agents; we cannot force the intruder to forget a message he has learned. Similarly, we cannot retract an asserted event. Thus the set of events and the messages sent over the network grow monotonically during transitions.

The set of *terms occurring in a strand* ℓ is denoted by $\mathit{trms}(\ell)$. The *events* of a strand ℓ is the set of terms $\mathit{ev}(\ell)$ defined as $\mathit{ev}(\ell) = \{e \mid \mathsf{assert}(e) \text{ occurs in } \ell\}$. The *free variables*, denoted by $\mathit{fv}(\ell)$, are the variables occurring in ℓ which are not bound by a universal quantifier, and when $\mathit{fv}(\ell) = \emptyset$ then ℓ is said to be *closed*. In many formalisms like process calculi, variables in a $\mathsf{receive}$ step would also be considered as bound variables. Since we, however, also express pattern matching here (since we allow arbitrary terms in $\mathsf{receive}$ steps, and, in particular, the same variable can occur in several $\mathsf{receive}$ steps and more than once), we like to refer to all such variables as free variables, anyway. We will later introduce a notion of well-formed constraints that requires all free variables to first occur in a $\mathsf{receive}$ step or a positive check, and thus corresponds to a notion of closedness in other formalisms. Moreover, given a substitution δ we can apply it to a constraint ℓ as expected, written $\delta(\ell)$, by applying δ to every free occurrence of a variable in ℓ. Note that the variables of a substitution δ might clash with the bound variables occurring in a strand ℓ, e.g., for $\delta = [y \mapsto f(x)]$ and $\ell = \forall x.\ x \not\approx y$ we have that $\delta(\ell) = \forall x.\ x \not\approx f(x)$. However, we can always avoid these issues by variable-renaming. For simplicity we therefore assume that the bound and free variables of strands are disjoint. Note also that we restrict ourselves here to a "bare metal" formalism by discarding all notions that are not relevant to our typing result. For instance, we have no notion of repetition, since one can simply consider an infinite set of such strands. We then also do not need a construct for creating fresh constants since we can simply consider a set of strands with uniquely chosen constants.

3.2 A Keyserver Example

Before we proceed with the formal definition of our protocol model we introduce a small keyserver protocol example adapted from [25]. In this protocol each participant u has an associated keyring $\mathsf{ring}(u)$ of currently used public keys. Any agent (or user) can register public keys with a trusted keyserver and these public

keys can later be revoked. The lifetime of a key may span multiple sessions, but whenever it is revoked the corresponding private key will be publicly known, and it should therefore not be used in a later session. Thus the keyserver needs to maintain the current status of keys and to model this feature we consider sets $\mathsf{valid}(u)$ and $\mathsf{revoked}(u)$ containing the valid respectively revoked keys for each user u. As an initial rule of the protocol we model an out-of-band registration of fresh keys (e.g., the user physically visits the server). Suppose we have a (countably infinite) set of constants that represents the users. For every user u and for every $j \in \mathbb{N}$ we then declare the strand:

$$\mathsf{insert}(pk_{u,j}, \mathsf{ring}(u)).\mathsf{insert}(pk_{u,j}, \mathsf{valid}(u)).\mathsf{send}(pk_{u,j}) \tag{T1}$$

where each $pk_{u,j}$ is a public key. Here, j is a "session number" and $pk_{u,j}$ represents a fresh public key the user u "has created in session j". This strand thus represents that a user u can create a fresh key $pk_{u,j}$ and insert it into its keyring, and the server then additionally inserts the key into its own set of valid keys. Lastly, the key is made public by sending it out.

We will later define the semantics of protocols by a state transition system, where in the initial state all sets are empty and no messages have been sent. Then for user $u = a$ and session $j = 1$, the above strand would get us to a new state where $pk_{a,1}$ is contained in $\mathsf{ring}(a)$ and $\mathsf{valid}(a)$ and the message $pk_{a,1}$ has been sent. Note that we do not have any built-in notion of set ownership, so we can model here strands that represent a mutual action of a user and the server.

As a second rule we model a key-revocation mechanism consisting of two separate strands: one for the users and one for the server. In the first strand the condition $PK_{u,j} \mathrel{\dot\in} \mathsf{ring}(u)$ expresses that $PK_{u,j}$ can be any value in the keyring. Not having any other condition, this models that the user can arbitrarily select a key from its keyring. Then it generates a fresh key $npk_{u,j}$, inserts it into its keyring, and sends the new key to the server, signed with the old key $PK_{u,j}$:

$$\begin{aligned} &PK_{u,j} \mathrel{\dot\in} \mathsf{ring}(u).\mathsf{insert}(npk_{u,j}, \mathsf{ring}(u)).\\ &\quad \mathsf{send}(\mathsf{sign}(\mathsf{inv}(PK_{u,j}), \langle u, npk_{u,j}\rangle)) \end{aligned} \tag{T2}$$

for each user u and for each session $j \in \mathbb{N}$. (Note that we also parameterize the variables; later on, we will require that different strands have different variables.) Rule T2 is, for instance, applicable to our concrete state where key $pk_{a,1}$ has been registered: it gets us to a new state where $npk_{a,1}$ has been added to $\mathsf{ring}(a)$ and the message $\mathsf{sign}(\mathsf{inv}(pk_{a,1}), \langle a, npk_{a,1}\rangle)$ has now been sent.

Afterwards, in the second strand, it is the keyserver's turn to act and its actions are initiated by an incoming message of the form $\mathsf{sign}(\mathsf{inv}(PK_i), \langle U_i, NPK_i\rangle)$:

$$\begin{aligned} &\mathsf{receive}(\mathsf{sign}(\mathsf{inv}(PK_i), \langle U_i, NPK_i\rangle)).\\ &(\forall A_i.\ NPK_i \mathrel{\dot\notin} \mathsf{valid}(A_i)).(\forall A_i.\ NPK_i \mathrel{\dot\notin} \mathsf{revoked}(A_i)).\\ &PK_i \mathrel{\dot\in} \mathsf{valid}(U_i).\mathsf{insert}(NPK_i, \mathsf{valid}(U_i)).\\ &\mathsf{insert}(PK_i, \mathsf{revoked}(U_i)).\mathsf{delete}(PK_i, \mathsf{valid}(U_i)).\\ &\mathsf{send}(\mathsf{inv}(PK_i)) \end{aligned} \tag{T3}$$

for each $i \in \mathbb{N}$. Again, this rule is applicable to the concrete state reached above, moves the value $pk_{a,1}$ from $\mathsf{valid}(a)$ to $\mathsf{revoked}(a)$, and inserts $npk_{a,1}$ into $\mathsf{valid}(a)$. Finally, the server discloses the private key $\mathsf{inv}(pk_{a,1})$; while this is of course not done in an actual implementation, it expresses that this protocol is secure even if the intruder learns the private key to an old revoked key.

3.3 Transaction Strands

One may wonder about the execution model for the strands from the previous example, in particular if that could cause race conditions on the checks and modifications of the sets if parallel execution of several strands leads to some interleaving of the respective set operations. Suppose for instance, in our keyserver example, that we register the key $pk_{a,1}$ using strand T1, and then send out the messages $\mathsf{sign}(\mathsf{inv}(pk_{a,1}), \langle a, npk_{a,i}\rangle)$ for $i \in \{1, 2\}$ using T2. Then $pk_{a,1}$ is in $\mathsf{valid}(a)$ and $\mathsf{ring}(a)$ contains the keys $pk_{a,1}$, $npk_{a,1}$, and $npk_{a,2}$. If we now run two instances of the strand T3, one for each of the signatures, and we assume that they are executed step-by-step instead of one atomic block, then we could end up in a state where both $npk_{a,1}$ and $npk_{a,2}$ have been registered at the keyserver (i.e., inserted into $\mathsf{valid}(a)$) but only one public key, $pk_{a,1}$, has been revoked, because both instances of T3 can perform all their checks before updating their databases. In fact, as we will define formally in the next subsection, we adopt a *transaction semantics*: a *transaction strand* (or just transaction) is defined to be a strand of the form $\mathsf{receive}(T).L.\mathsf{send}(T')$ where T and T' are finite sets of terms, L is a strand that does not contain any send or $\mathsf{receive}$ steps, and where we write $\mathsf{receive}(\{t_1, \ldots, t_n\})$ as an abbreviation for $\mathsf{receive}(t_1).\cdots.\mathsf{receive}(t_n)$ (similarly for send steps). The idea is that such a transaction is always performed atomically, i.e., as a single transition. This reflects, in our opinion, very well the normal work-flow of a web server with a database: the server receives an incoming request, performs some lookups and checks on its database (possibly aborting the transaction), then performs some modifications on its database, and sends a reply (which may be also a request to another server). The key is that the server serializes the handling of such transactions (to avoid said race conditions). A transaction semantics allows us to abstract from the implementation of such serialization mechanisms and thus focus on the verification of a larger system. Another example are crypto APIs, where a token receives an API command, performs some lookups and checks in its memory (possibly aborting the transaction), performs some updates to its memory and then gives out a result. Also here, we typically do not want to reason about race conditions from several API calls in parallel.

This is indeed slightly different from the "philosophy" of many process calculus approaches (e.g., StatVerif [3] and Set-π [12]) where one would have to introduce explicit locking mechanisms. Also, the original notion of strand spaces by Guttman [31] is actually based on a notion of only a partial (instead of a total) order on send and $\mathsf{receive}$ steps in an execution; if we regard however set operations as interactions with a database with locking, then we obtain the partial order that our transaction semantics defines.

3.4 Transition Systems

Now that we have introduced the elements of our protocols we define a *protocol* $\mathcal{S}$ to be a countable set of transaction strands where no variable occurs in two different strands. The set of terms $trms(\mathcal{S})$ occurring in $\mathcal{S}$ is defined as expected.

Before giving the formal definition of the transition system we will first define the notion of a *database mapping* D to be a finite set of pairs (t, s) of terms, and for closed strands ℓ we define the ground database mapping $db(\ell)$ as

$$db(\ell) = \{(t,s) \mid \mathsf{insert}(t,s).\ell' \text{ is a suffix of } \ell \text{ and } \mathsf{delete}(t,s) \text{ does not occur in } \ell'\}$$

Let $D = \{(t_1, s_1), \ldots, (t_n, s_n)\}$ be a database mapping and ℓ a closed strand, then we may write $db(\mathsf{insert}(D).\ell)$ as a shorthand for $db(\mathsf{insert}(t_1, s_1).\cdots.\mathsf{insert}(t_n, s_n).\ell)$.

States in the (ground) transition system are of the form $(\mathcal{S}; M, D, E)$ where $\mathcal{S}$ is a protocol, M is the set of messages that has been sent over the network and that we also refer to as the intruder knowledge, D is a database mapping representing the state of all databases, and E is the set of events that have occurred. The initial state is $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset)$ for a protocol $\mathcal{S}_0$.

Definition 1. *A transition relation on states is defined as:*

$$(\mathcal{S}; M, D, E) \xRightarrow{\sigma,\ell} (\mathcal{S} \setminus \{\ell\}; M \cup \sigma(T'), D', E')$$
where $D' = db(\mathsf{insert}(D).\sigma(L))$ *and* $E' = E \cup ev(\sigma(L)))$

if the following conditions are met:

C1: $\ell = \mathsf{receive}(T).L.\mathsf{send}(T') \in \mathcal{S}$ *is a transaction strand,*

C2: σ *is a ground substitution with domain* $fv(\ell)$*,*

C3: $M \vdash \sigma(t)$ *for all terms* $t \in T$*,*

C4: $\sigma(t) = \sigma(t')$ *for all steps* $t \doteq t'$ *occurring in* L*,*

C5: $\sigma(\delta(t)) \neq \sigma(\delta(t'))$ *for all steps* $\forall \bar{x}.\ t \not\doteq t'$ *occurring in* L *and all ground substitutions* δ *with domain* $\bar{x}$*,*

C6: $\sigma((t,s)) \in db(\mathsf{insert}(D).\sigma(L'))$ *for all prefixes* $L'.(t \dot{\in} s)$ *of* L*,*

C7: $\sigma(\delta((t,s))) \notin db(\mathsf{insert}(D).\sigma(L'))$ *for all prefixes* $L'.(\forall \bar{x}.\ t \dot{\notin} s)$ *of* L *and all ground substitutions* δ *with domain* $\bar{x}$*,*

C8: $\sigma(t) \in E \cup ev(\sigma(L'))$ *for all prefixes* $L'.\mathsf{event}(t)$ *of* L*,*

C9: $\sigma(\delta(t)) \notin E \cup ev(\sigma(L'))$ *for all prefixes* $L'.(\forall \bar{x}.\ \neg\mathsf{event}(t))$ *of* L *and all ground substitutions* δ *with domain* $\bar{x}$*.*

Here the first side-condition C1 simply ensures that ℓ is actually a transaction strand of the protocol, and the second condition C2 ensures that σ is actually an assignment of the free variables in ℓ to concrete values. Condition C3 states that the intruder must be able to derive the messages that ℓ expects to receive. The conditions C4 to C9 state that all checks and set updates performed by ℓ are satisfied under σ. As the effect of a transition the strand ℓ is removed from $\mathcal{S}$, the intruder learns $\sigma(T')$, the asserted events of $\sigma(L)$ are added to the successor state, and the databases are updated according to the set operations of $\sigma(L)$.

Note that the whole transaction strand ℓ is "consumed" in each transition because we want the strands of protocols to be atomic transactions. This is different from other strand-based approaches in which a transition only eliminates one step of a strand and in which strands might contain multiple transactions (e.g., from a state containing the protocol $\{PK \doteq pk_{a,1}.\mathsf{receive}(npk_{a,1}).\mathsf{send}(PK)\}$ we can reach a state containing $\{\mathsf{receive}(npk_{a,1}).\mathsf{send}(PK)\}$ and where PK must be mapped to $pk_{a,1}$). Defining our protocol semantics on a transactional level, however, is without loss of generality: it is always possible to refine a strand into smaller transaction strands while preserving the causal relationship of the original strand (i.e., transaction $i+1$ of a strand with n transactions can only be performed after transaction i, for any $i \in \{1,\ldots,n-1\}$). For instance, one can insert additional message-transmissions between steps, e.g., the strand $PK \doteq pk_{a,1}.\mathsf{receive}(npk_{a,1}).\mathsf{send}(PK)$ can be split into two transactions, namely $PK \doteq pk_{a,1}.\mathsf{send}(f(PK))$ and $\mathsf{receive}(f(PK)).\mathsf{receive}(npk_{a,1}).\mathsf{send}(PK)$ where f is a fresh private symbol of arity one that we here use to preserve the causal relationship and to carry state information. In general, to split a strand $\ell_1.\cdots.\ell_n$ containing transaction strands ℓ_i we can add additional steps that carry state information from ℓ_i to ℓ_{i+1} and which ensure that ℓ_i can only be performed after ℓ_{i+1}: $\ell_i.\mathsf{send}(\mathsf{state}_{\ell_i}(x_1,\ldots,x_m))$ and $\mathsf{receive}(\mathsf{state}_{\ell_i}(x_1,\ldots,x_m)).\ell_{i+1}$, where $\mathsf{state}_{\ell_i} \in \Sigma^m_{priv}$ is private and unique to ℓ_i and where $fv(\ell_i) = \{x_1,\ldots,x_m\}$. Such a transformation can also be used to link transactions $\ell_1,\ldots,\ell_n$ together, or to split a transaction strand into smaller transactions if one wishes to have greater granularity in state transitions. For tools based on transaction strands such an encoding would be useful; it would be convenient for users if they are allowed to specify strands containing multiple transactions. In this paper, however, we will not provide such an input language for a tool—rather, we have decided to keep the protocol model simple by only allowing single-transaction strands. This decision is legitimate, in our opinion, since the above encoding for linking transactions can easily be automated and be transparent to end-users.

Finally, we note that protocol goals such as secrecy can also be encoded as strands. For instance, we can extend our running keyserver example with strands

$$\mathsf{receive}(\mathsf{inv}(PK'_i)).PK'_i \doteq \mathsf{valid}(h).\mathsf{assert}(\mathsf{attack})$$

for each honest user h and $i \in \mathbb{N}$, and an event attack that denotes when an attack has happened. Hence, if the private key of a valid public key for an honest agent is leaked then there is a violation of secrecy, and in those cases

we emit the event attack using the construct assert. In other words, if there is a reachable state $(\mathcal{S}; M, D, E)$ in which $\mathsf{attack} \in E$ then the protocol has a vulnerability.

4 Symbolic Constraints

At the core of all typing results is a sound and complete constraint reduction system. It was originally used as an efficient procedure for model-checking of security protocols [22, 29, 6], but is also used as a proof technique when proving relative soundness results such as [14, 24, 2, 1, 19]. The constraints themselves arise from the symbolic exploration of the protocol state space where each symbolic state contains a constraint that represents the steps taken in the protocol so far. Any solution to a reachable constraint then represents (one or several) concrete runs of the protocol. In this section we consider constraints for stateful protocols.

4.1 Syntax and Semantics

The most basic parts of a symbolic constraint are requirements on the intruder to produce messages that honest agents expect to receive. For instance, if the messages $m_1, \ldots, m_n$ (where each m_i might contain variables) have been sent out and some agent expects to receive a message pattern t it is standard to represent as a constraint the requirement on the intruder to produce t given the m_i. Any solution $\mathcal{I}$ to such a constraint is an assignment of the variables $fv(\{m_1, \ldots, m_n, t\})$ to ground terms such that $\mathcal{I}(\{m_1, \ldots, m_n\}) \vdash \mathcal{I}(t)$ holds. In [19] there was the idea to represent a (finite) set of such constraints by a strand, with send steps for messages the intruder has to generate, and receive steps for messages that the intruder learns (all in the order this happens), e.g., the constraint we just explained can be represented as the strand $\mathsf{receive}(m_1).\cdots.\mathsf{receive}(m_n).\mathsf{send}(t)$. We additionally want to handle strands with sets, and so we also just insert all the set operations (and similarly the checks and event assertions) into the intruder strands in the order they happen in a concrete execution. With this, our constraints are just like the strands for honest agents but with the direction of send and receive steps inverted, i.e., a send step from an honest agent becomes a receive step in our constraints and vice versa. For these reasons we define the syntax of our constraints to range over strands. Similarly to the ordinary strands we call constraints that only contains receive, send, equalities, and inequalities for *ordinary constraints*. We will often reuse the operations defined on strands for symbolic constraints, since they share the same syntax, and we also make the assumption that the bound variables occurring in a constraint are disjoint from its free variables. Moreover, we define the *intruder knowledge* $ik(\mathcal{A})$ of a constraint $\mathcal{A}$ as the set of received messages: $ik(\mathcal{A}) = \{t \mid \mathsf{receive}(t) \text{ occurs in } \mathcal{A}\}$.

An *interpretation* $\mathcal{I}$ *for a constraint* $\mathcal{A}$ (or just an *interpretation* if $dom(\mathcal{I}) = \mathcal{V}$) is now defined to be a substitution such that $fv(\mathcal{A}) \subseteq dom(\mathcal{I})$ and $img(\mathcal{I})$ is

ground. We then inductively define a model relation $\models_{M,D,E}$ between interpretations and constraints where M, D, and E are respectively the initial intruder knowledge, state of databases, and events:

Definition 2 (Constraint semantics)**.**

$$
\begin{array}{l}
\mathcal{I} \models_{M,D,E} 0 \;\; \textit{iff} \;\; \textit{true} \\
\mathcal{I} \models_{M,D,E} \mathsf{send}(t).\mathcal{A} \;\; \textit{iff} \;\; M \vdash \mathcal{I}(t) \;\; \textit{and} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} \mathsf{receive}(t).\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I} \models_{M \cup \{\mathcal{I}(t)\},D,E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} t \doteq t'.\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I}(t) = \mathcal{I}(t') \;\; \textit{and} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} (\forall \bar{x}.\; t \neq t').\mathcal{A} \;\; \textit{iff} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \;\; \textit{and} \\
\quad \mathcal{I}(\delta(t)) \neq \mathcal{I}(\delta(t')) \;\; \textit{for all ground} \; \delta \; \textit{with domain} \; \bar{x} \\
\mathcal{I} \models_{M,D,E} \mathsf{insert}(t,s).\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I} \models_{M,D \cup \{\mathcal{I}((t,s))\},E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} \mathsf{delete}(t,s).\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I} \models_{M,D \setminus \{\mathcal{I}((t,s))\},E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} t \mathrel{\dot\in} s.\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I}((t,s)) \in D \;\; \textit{and} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} (\forall \bar{x}.\; t \mathrel{\dot\notin} s).\mathcal{A} \;\; \textit{iff} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \;\; \textit{and} \\
\quad \mathcal{I}(\delta((t,s))) \notin D \;\; \textit{for all ground} \; \delta \; \textit{with domain} \; \bar{x} \\
\mathcal{I} \models_{M,D,E} \mathsf{assert}(e).\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I} \models_{M,D,E \cup \{\mathcal{I}(e)\}} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} \mathsf{event}(e).\mathcal{A} \;\; \textit{iff} \;\; \mathcal{I}(e) \in E \;\; \textit{and} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \\
\mathcal{I} \models_{M,D,E} (\forall \bar{x}.\; \neg\mathsf{event}(e)).\mathcal{A} \;\; \textit{iff} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \;\; \textit{and} \\
\quad \mathcal{I}(\delta(e)) \notin E \;\; \textit{for all ground} \; \delta \; \textit{with domain} \; \bar{x}
\end{array}
$$

Finally, we say that an interpretation $\mathcal{I}$ is a *model of (or solution to) a constraint* $\mathcal{A}$, written $\mathcal{I} \models \mathcal{A}$, iff $\mathcal{I} \models_{\emptyset,\emptyset,\emptyset} \mathcal{A}$.

We can now prove some useful lemmas about the constraint semantics. First, we have a lemma that we frequently apply in proofs (without explicitly referencing it) that allows us to split and merge constraints:

Lemma 1. *Given a ground set of terms M, a ground database mapping D, a ground set E of asserted events, an interpretation $\mathcal{I}$, and symbolic constraints $\mathcal{A}$ and $\mathcal{A}'$, the following holds:*

$$
\begin{array}{l}
\mathcal{I} \models_{M,D,E} \mathcal{A}.\mathcal{A}' \;\; \textit{iff} \; \mathcal{I} \models_{M,D,E} \mathcal{A} \;\; \textit{and} \; \mathcal{I} \models_{M',D',E'} \mathcal{A}' \\
\textit{where} \; M' = M \cup ik(\mathcal{I}(\mathcal{A})), D' = db(\mathsf{insert}(D).\mathcal{I}(\mathcal{A})), \\
\textit{and} \; E' = E \cup ev(\mathcal{I}(\mathcal{A}))
\end{array}
$$

Secondly, we can prove a useful relationship between the side-conditions C1 to C9 of the ground transition system and the constraint semantics. First we define the notion of the *dual* of a strand S by "swapping" the direction of receive and send steps. Formally, $dual(\mathfrak{s})$ denotes the dual of the strand step $\mathfrak{s}$ defined such that $dual(\mathsf{receive}(t)) = \mathsf{send}(t)$, $dual(\mathsf{send}(t)) = \mathsf{receive}(t)$, and $dual(\mathfrak{s}) = \mathfrak{s}$ for any other step $\mathfrak{s}$. It is then extended homomorphically to strands as expected. We will interpret dual strands as symbolic constraints and under this interpretation we can prove the following relationship:

Lemma 2. *Given a ground state $(\mathcal{S}; M, D, E)$, a transaction strand $\ell \in \mathcal{S}$ (condition C1), and a ground substitution σ with domain $fv(\ell)$ (condition C2), then the conditions C3 to C9 hold if and only if $\sigma \models_{M,D,E} dual(\ell)$.*

4.2 Symbolic Transition System

Now that we have defined the syntax and semantics of constraints we can construct a protocol transition system in which we build up constraints during transitions. In this symbolic transition system a symbolic state $(\mathcal{S}; \mathcal{A})$ consists of a protocol $\mathcal{S}$ and a constraint $\mathcal{A}$, and the initial state $(\mathcal{S}_0; 0)$ then consists of the initial protocol $\mathcal{S}_0$ and the empty constraint 0. During transitions we then build up a constraint by interpreting dual honest-agent strands as constraints:

Definition 3. *A transition relation on symbolic states is defined as:*

$$(\mathcal{S}; \mathcal{A}) \stackrel{\ell}{\Longrightarrow}^{\bullet} (\mathcal{S} \setminus \{\ell\}; \mathcal{A}.dual(\ell)) \text{ if } \ell \in \mathcal{S}$$

We will now impose a well-formedness requirement on protocols; variables in honest-agent strands must either originate from a received message or in a positive check (e.g., a set query). In ordinary protocols there is nothing nondeterministic in the behavior of honest agents, so all free variables in their strands shall first occur in messages they receive. Now that we add set operations, we extend well-formedness naturally to set comprehensions: a set-membership check like $x \mathrel{\dot\in} s$ allows the agent to non-deterministically choose any element from s for x—unless x is already constrained before, thus limiting the choice accordingly.

We also require that reachable constraints in the symbolic transition system are of a well-formed kind that is dual to the well-formedness of protocols; every free variable of a constraint represents either a message that depends on choices the intruder can make (e.g., variables originating from send steps), or originates from a positive check. To that end we formally define constraint well-formedness first and then use this definition to define protocol well-formedness:

Definition 4. *A constraint $\mathcal{A}$ is* well-formed w.r.t. variables X *(or simply* well-formed *when $X = \emptyset$) iff $wf_X(\mathcal{A})$ where*

$$\begin{array}{lll}
wf_X(0) & \textit{iff} & \textit{true} \\
wf_X(\mathsf{send}(t).\mathcal{A}) & \textit{iff} & wf_{X \cup fv(t)}(\mathcal{A}) \\
wf_X(\mathsf{receive}(t).\mathcal{A}) & \textit{iff} & fv(t) \subseteq X \textit{ and } wf_X(\mathcal{A}) \\
wf_X(t \doteq t'.\mathcal{A}) & \textit{iff} & fv(t') \subseteq X \textit{ and } wf_{X \cup fv(t)}(\mathcal{A}) \\
wf_X(\mathsf{insert}(t,s).\mathcal{A}) & \textit{iff} & fv(t) \cup fv(s) \subseteq X \textit{ and } wf_X(\mathcal{A}) \\
wf_X(\mathsf{delete}(t,s).\mathcal{A}) & \textit{iff} & fv(t) \cup fv(s) \subseteq X \textit{ and } wf_X(\mathcal{A}) \\
wf_X(t \mathrel{\dot\in} s.\mathcal{A}) & \textit{iff} & wf_{X \cup fv(t) \cup fv(s)}(\mathcal{A}) \\
wf_X(\mathsf{assert}(t).\mathcal{A}) & \textit{iff} & fv(t) \subseteq X \textit{ and } wf_X(\mathcal{A}) \\
wf_X(\mathsf{event}(t).\mathcal{A}) & \textit{iff} & wf_{X \cup fv(t)}(\mathcal{A}) \\
wf_X(\mathfrak{a}.\mathcal{A}) & \textit{iff} & wf_X(\mathcal{A}) \textit{ otherwise}
\end{array}$$

Here the set X collects the variables that have occurred in send steps or positive checks. In other words, every free variable of a well-formed constraint originates from either a send step, a $\mathrel{\dot\in}$ step, an event step, or at the left-hand side of a $\doteq$ step (or a negative check such as an inequality, but in those cases the new variables cannot be used elsewhere). We can then reuse the definition

of well-formedness of constraints to formally define a notion of well-formedness of protocols:

Definition 5. *A protocol $\mathcal{S}$ is* well-formed *iff for all strands $\ell \in \mathcal{S}$ the symbolic constraint $dual(\ell)$ is well-formed.*

Well-formedness of reachable constraints is now easy to prove. We write $\overset{w}{\Longrightarrow}^*$ here to denote the reflexive-transitive closure of $\dot{\Longrightarrow}$ where the label $w = (\sigma_1, \ell_1), \ldots, (\sigma_n, \ell_n)$ denotes a sequence of transition labels, and similarly $\overset{w'}{\Longrightarrow}^{\bullet *}$ denotes the reflexive-transitive closure of $\dot{\Longrightarrow}^\bullet$ where $w' = \ell_1, \ldots, \ell_n$.

Lemma 3 (Well-formedness of reachable symbolic constraints)**.** *If $\mathcal{S}_0$ is a well-formed protocol and $(\mathcal{S}_0; 0) \overset{w}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A})$ then $\mathcal{A}$ is a well-formed symbolic constraint and $\mathcal{S}$ is a well-formed protocol.*

We now prove that the symbolic and ground transition systems are equivalent. Essentially, if we consider for every reachable symbolic state $(\mathcal{S}; \mathcal{A})$ and every model $\mathcal{I}$ of $\mathcal{A}$ the corresponding ground state $(\mathcal{S}; ik(\mathcal{I}(\mathcal{A})), db(\mathcal{I}(\mathcal{A})), ev(\mathcal{I}(\mathcal{A})))$, then we obtain exactly the reachable states of the ground transition system:

Theorem 1 (Equivalence of transition systems)**.** *For any protocol $\mathcal{S}_0$,*

$$\begin{aligned}
&\{(\mathcal{S}; M, D, E) \mid \exists w.\ (\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \overset{w}{\Longrightarrow}^* (\mathcal{S}; M, D, E)\} = \\
&\{(\mathcal{S}; ik(\mathcal{I}(\mathcal{A})), db(\mathcal{I}(\mathcal{A})), ev(\mathcal{I}(\mathcal{A}))) \mid \\
&\qquad \exists w.\ (\mathcal{S}_0; 0) \overset{w}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A}) \text{ and } \mathcal{I} \models \mathcal{A}\}
\end{aligned}$$

4.3 Reduction to Ordinary Constraints

Definition 6 (Translation of symbolic constraints)**.** *Given a constraint $\mathcal{A}$ its translation into ordinary constraints is denoted by $tr(\mathcal{A}) = tr_{\emptyset,\emptyset}(\mathcal{A})$ where:*

$$\begin{aligned}
&tr_{D,E}(0) = \{0\} \\
&tr_{D,E}(\mathsf{insert}(t,s).\mathcal{A}) = tr_{D \cup \{(t,s)\},E}(\mathcal{A}) \\
&tr_{D,E}(\mathsf{delete}(t,s).\mathcal{A}) = \{ \\
&\quad (t,s) \doteq d_1. \cdots .(t,s) \doteq d_i. \\
&\quad (t,s) \not\doteq d_{i+1}. \cdots .(t,s) \not\doteq d_n.\mathcal{A}' \mid \\
&\qquad D = \{d_1, \ldots, d_i, \ldots, d_n\}, 0 \leq i \leq n, \\
&\qquad \mathcal{A}' \in tr_{D \setminus \{d_1,\ldots,d_i\},E}(\mathcal{A})\} \\
&tr_{D,E}(t \mathrel{\dot\in} s.\mathcal{A}) = \{(t,s) \doteq d.\mathcal{A}' \mid d \in D, \mathcal{A}' \in tr_{D,E}(\mathcal{A})\} \\
&tr_{D,E}((\forall \bar{x}.\ t \not\in s).\mathcal{A}) = \{ \\
&\quad (\forall \bar{x}.\ (t,s) \not\doteq d_1). \cdots .(\forall \bar{x}.\ (t,s) \not\doteq d_n).\mathcal{A}' \mid \\
&\qquad D = \{d_1, \ldots, d_n\}, 0 \leq n, \mathcal{A}' \in tr_{D,E}(\mathcal{A})\} \\
&tr_{D,E}(\mathsf{assert}(e).\mathcal{A}) = tr_{D,E \cup \{e\}}(\mathcal{A}) \\
&tr_{D,E}(\mathsf{event}(e).\mathcal{A}) = \{e \doteq e'.\mathcal{A}' \mid e' \in E, \mathcal{A}' \in tr_{D,E}(\mathcal{A})\} \\
&tr_{D,E}((\forall \bar{x}.\ \neg\mathsf{event}(e)).\mathcal{A}) = \{ \\
&\quad (\forall \bar{x}.\ e \not\doteq e_1). \cdots .(\forall \bar{x}.\ e \not\doteq e_n).\mathcal{A}' \mid \\
&\qquad E = \{e_1, \ldots, e_n\}, 0 \leq n, \mathcal{A}' \in tr_{D,E}(\mathcal{A})\} \\
&tr_{D,E}(\mathfrak{a}.\mathcal{A}) = \{\mathfrak{a}.\mathcal{A}' \mid \mathcal{A}' \in tr_{D,E}(\mathcal{A})\} \text{ otherwise}
\end{aligned}$$

The key to our typing result—that allows us to benefit from existing typing results—is to first reduce the problem of solving general intruder constraints (with set operations) to solving ordinary intruder constraints (without set operations). To that end we introduce a sound and complete translation mechanism that removes the stateful parts of constraints, for instance those reachable in $\dot{\Longrightarrow}^{\bullet *}$. The translation $tr(\cdot)$ is then defined in Definition 6 where D is a database mapping and E is a set of events that records what has occurred in the constraint so far. Intuitively, the set $tr_{D,E}(\cdot)$ of reduced constraints represents a disjunction of ordinary constraints, and since we cannot represent disjunctions in our constraints we use sets instead. Note also that D and E will always be finite and that this does not mean that we are restricting ourselves to only finitely many sessions. Rather, in each protocol execution only finitely many things have happened and D and E then represents the state of the sets and events respectively. Hence the translation always produces a finite set and for this reason we can interpret the set as a finite disjunction of constraints.

We will now explain how each set operation is translated (the event steps are translated similarly). The purpose of the translation $tr(\mathcal{A})$ is to capture precisely the models of $\mathcal{A}$ using only a finite number of ordinary constraints, so we will proceed with the explanation with this in mind. The simplest case is the $\mathsf{insert}(t, s)$ case, and here we record the insertion for the remaining translation. Now consider the $t \mathrel{\dot\in} s$ case. For any model $\mathcal{I}$ of $t \mathrel{\dot\in} s$ with a given database mapping $D = \{(t_1, s_1), \ldots, (t_n, s_n)\}$ (where each entry of D might contain variables) we know that $\mathcal{I}((t, s)) \in \mathcal{I}(D)$. In other words, some check $(t, s) \doteq d$ for some d in D has $\mathcal{I}$ as a model if and only if $t \mathrel{\dot\in} s$ has $\mathcal{I}$ as a model, and by then constructing one constraint for each $d_i \in D$ where we require $(t, s) \doteq d_i$ we get the desired result. For the $\forall \bar{x}.\ t \mathrel{\dot\notin} s$ case we know that $\mathcal{I}(\delta(t)) \neq \mathcal{I}(\delta(t'))$ *or* $\mathcal{I}(\delta(s)) \neq \mathcal{I}(\delta(s'))$ for any $(t', s') \in D$ and ground substitution δ with domain $\bar{x}$. In other words, $\mathcal{I}(\delta((t, s))) \neq \mathcal{I}(\delta((t', s')))$ for all $(t', s') \in D$ and this is exactly what the translation expresses. We also have to make sure that the newly introduced quantified constraints do not capture any variables of D. This is, in fact, the case for all constraints reachable in our symbolic transition system, since we have previously assumed all strands of protocols to have disjoint variables from each other and also that the bound and free variables of strands are disjoint. Thus this property also holds for the reachable constraints. The most interesting case is the translation of $\mathsf{delete}(t, s)$ steps. Since terms may contain variables we do not know a priori which insertions to remove from D, but we still need to ensure that t has actually been removed from the set s in the remaining constraint translation—otherwise the translation would be unsound. We accomplish this by partitioning the insertions D into those $\{d_1, \ldots, d_i\}$ that must be equal to (t, s) in the remaining translation and the remaining $D \setminus \{d_1, \ldots, d_i\}$ that are unequal to (t, s), and we thus add equality and inequality constraints to express this partitioning. Consequently, we then remove $\{d_1, \ldots, d_i\}$ from D for the remaining translation. Note that there will in general be cases where the choice of partitioning results in an unsatisfiable constraint, but since we construct constraints for *all* possibilities the translation still captures exactly the models of the original constraint. Note also that this

partitioning of D implies that an exponential number of constraints are constructed in this case, namely one for each subset of D. The translation is meant to be used purely as a problem reduction—in a verification procedure one could ensure that trivially unsatisfiable translations are ignored to reduce the number of produced constraints.

Finally, we show that tr is indeed a reduction, i.e., that $tr(\mathcal{A})$ captures exactly the models of $\mathcal{A}$, and that tr preserves well-formedness:

Theorem 2 (Semantic equivalence of constraints and their translation)**.** *$\mathcal{I} \models \mathcal{A}$ if and only if there exists $\mathcal{A}' \in tr(\mathcal{A})$ such that $\mathcal{I} \models \mathcal{A}'$. Also, if $\mathcal{A}$ is well-formed and $\mathcal{A}' \in tr(\mathcal{A})$ then $\mathcal{A}'$ is well-formed.*

5 Lifting Typing Results to Stateful Protocols

So far everything has been untyped. We will now consider a simple type system in which we annotate terms with types. In particular, each message pattern that an honest agent in a protocol expects to receive will have an intended type, and in a typed model we restrict all substitutions to well-typed ones. In a typed model the intruder is therefore effectively restricted to only sending messages which conform to the types. For protocols that satisfy a syntactic requirement—*type-flaw resistance*—we then prove that this restriction is sound, and this result we call a *typing result*. For proving our result we use the reduction tr from constraints with sets to ordinary constraints, enabling us to use existing typing results for protocols without sets and "lift" them to stateful protocols.

5.1 Typed Model

The type system we introduce now uses a structure for types which is similar to the structure of terms (and so we will be able to reuse all notions of terms for types as expected). Recall that our notion of terms are parameterized over a set Σ of symbols. The idea is to use almost the same notion for our types, only not allowing constants $\mathcal{C} \subseteq \Sigma$ and variables in types and instead use a finite set of *atomic types* $\mathfrak{T}_a$ that could include, for instance, agent. In addition to atomic types we also have *composed types*. For instance, in our running example we use private keys of the form $\mathsf{inv}(PK)$. This term has the composed type $\mathsf{inv}(\mathsf{value})$, where PK has type value. We can also assign the type $\mathsf{inv}(\mathsf{value})$ to variables and we are therefore not limited to only using atomic keys.

We define the intended types of a protocol specification by a *typing function* Γ that assigns a type to every term; it can be any function that satisfies the following properties:

1. $\Gamma(c) \in \mathfrak{T}_a$ for every $c \in \mathcal{C}$.
2. $\Gamma(f(t_1, \ldots, t_n)) = f(\Gamma(t_1), \ldots, \Gamma(t_n))$ for every $f \in \Sigma^n \setminus \mathcal{C}$ and terms t_i.

The first of these axioms assigns atomic types to constants whereas the second axiom assigns composed types to composed terms. We also assign types to

variables and we only require here that symbols occurring in a type have been applied with the correct number of parameters, and that constants from $\mathcal{C}$ do not appear in the types of variables. Additionally, we assume the existence of an atomic type value; we will later require that all terms inserted into sets all have the same atomic type, and we use value for this purpose. The function Γ is moreover extended to sets of terms as expected.

For instance, in our running example we might define $\mathfrak{T}_a = \{\mathsf{value}, \mathsf{agent}, \mathsf{attacktype}\}$ where $\Gamma(a) = \mathsf{agent}$ for all users and servers a, $\Gamma(pk) = \mathsf{value}$ for any element pk of a set, and $\Gamma(\mathsf{attack}) = \mathsf{attacktype}$. Similarly, the variables U_i have type agent and the variables $PK_{u,j}$, PK_i, and NPK_i have type value. All short-term public keys have type value and all short-term private keys have type $\mathsf{inv}(\mathsf{value})$. Since we use terms to model families of sets we have as a consequence that, e.g., keyrings of the form $\mathsf{ring}(u)$, for users u, have type $\mathsf{ring}(\mathsf{agent})$.

For the typing result to hold we need to ensure that the intruder always has access to arbitrarily many terms of any type (otherwise he would not necessarily be able to always make a well-typed choice). More formally, we partition the set of public constants $\mathcal{C}_{pub}$ into the countably infinite sets $\mathcal{C}_{pub}^{\alpha_1}, \ldots, \mathcal{C}_{pub}^{\alpha_n}$ where $\mathfrak{T}_a = \{\alpha_1, \ldots, \alpha_n\}$ and $\Gamma(\mathcal{C}_{pub}^{\alpha_i}) = \{\alpha_i\}$ for all $i \in \{1, \ldots, n\}$. This models that the intruder has access to an unbounded supply of fresh constants of any atomic type. To ensure the same for composed types, there is a small technical problem, namely that we want functions like $\mathsf{inv}(\cdot)$ to be private, but this would lead to a quite complicated model to ensure that the intruder can do this. So for the sake of this section we make the following technical restriction: we assume that *all* non-constant function symbols $\Sigma \setminus \mathcal{C}$ are *public*. To model a private function f of arity $n > 0$, we can encode as a public function symbol f' of arity $n+1$ where the additional argument is filled in all protocol strands with a secret constant sec_f that the intruder does not know. Note that this simple encoding of private functions is merely used here in the typing result section to make the development smooth. With this construction the intruder can always generate well-typed instances of any type.

Finally, in the typed model we restrict ourselves to only consider well-typed solutions to intruder constraints. To capture this idea we define a predicate on substitutions stating that every variable is mapped to a term of the same type for substitutions satisfying this property:

Definition 7. *A substitution δ is* well-typed *iff $\Gamma(x) = \Gamma(\delta(x))$ for all $x \in \mathcal{V}$.*

Conversely, substitutions that are not well-typed are *ill-typed*.

5.2 Type-Flaw Resistance

In this subsection we will define a sufficient syntactical condition for protocols (i.e., verifying the condition does not require an exploration of the state space of a protocol) that allows us to prove our typing result for protocols that have this property. This condition will be named *type-flaw resistance* and it is similar to the typing result conditions of [1, 19].

First, we will define a set of *sub-message patterns* $SMP(M)$ for sets of message patterns M:

Definition 8 (Sub-message patterns)**.** *The set of* sub-message patterns*,* $SMP(M)$*, of a set of terms* M *is the least set closed under the following rules:*

1. *If* $t \in M$ *then* $t \in SMP(M)$
2. *If* $t \in SMP(M)$ *and* t' *is a subterm of* t *then* $t' \in SMP(M)$
3. *If* $t \in SMP(M)$ *and* δ *is a well-typed substitution then* $\delta(t) \in SMP(M)$
4. *If* $t \in SMP(M)$ *and* $\mathsf{Ana}(t) = (K, T)$ *then* $K \subseteq SMP(M)$

The intention is that we can apply SMP to the message patterns $trms(\mathcal{S})$ of a protocol $\mathcal{S}$, and $SMP(trms(\mathcal{S}))$ is then an over-approximation of the messages that the intruder might ever learn from the honest agents of $\mathcal{S}$ (or send out to the honest agents) in any well-typed protocol run. The definition is generalized over an arbitrary set of terms, so that we can also apply SMP to messages occurring in a strand or a constraint. Consider, for instance, the set of sub-message patterns $SMP(trms(\mathcal{A}))$ built from the terms that occur in some well-formed constraint $\mathcal{A}$. The set then covers all message patterns of every message that might be sent over the network, and any pattern in a check made by an honest agent, for well-typed choices of the variables in the patterns.

Note that we also close the set of sub-message patterns under terms occurring during decomposition. (For proving a typing result for ordinary constraints one should prove that the constraints arising through constraint reductions never "fall out" of the set of sub-message patterns. Here one needs to make sure that the terms arising from decomposition are also captured by the sub-message patterns, since the keys usually end up in a reachable constraint in the constraint reduction system.) Since we assume that the terms obtained from a decomposition must be subterms of the original term, however, we already cover those terms in the second rule of Definition 8 and so we only include the keys used during decomposition in the fourth rule.

We will now require that all pairs t, t' of sub-message patterns that are not variables (i.e., are non-variable) can only be unified if their types match, and this will be our main condition of type-flaw resistance. This is a sufficient requirement to distinguish terms of different types and it therefore enables us to argue that ill-typed choices are unnecessary. In a nutshell, the typing result works as follows: with the condition of type-flaw resistance we ensure that the intruder cannot take a message generated by an honest agent (or a non-variable subterm of it) and use it in a different "context" of the protocol, i.e., a non-variable subterm of a different type. The constraint-based representation then allows one to argue that no attack relies on an ill-typed choice by the intruder: one can show that there is a sound, complete, and terminating reduction procedure for

(ordinary) intruder constraints that will instantiate variables only upon unification of two elements of SMP—and such a unifier is guaranteed to be well-typed for a type-flaw resistant protocol. All remaining uninstantiated variables can be instantiated arbitrarily by the intruder, in particular in a well-typed way. Thus one can conclude that there is a well-typed solution if there is one at all.

Definition 9 (Type-flaw resistance)**.** *First, let the* set operation tuples *of a constraint (or strand) $\mathcal{A}$ be defined as:*

$$setops(\mathcal{A}) = \{(t,s) \mid \mathsf{insert}(t,s) \text{ or } \mathsf{delete}(t,s) \text{ or } t \dot{\in} s \\ \text{or } (\forall \bar{x}.\ t \dot{\notin} s) \text{ for some } \bar{x} \text{ occurs in } \mathcal{A}\}$$

and extend this definition to protocols $\mathcal{S}$ as follows:

$$setops(\mathcal{S}) = \bigcup_{\ell \in \mathcal{S}} setops(\ell)$$

Then we define type-flaw resistance as follows:

1. *A set of terms M is* type-flaw resistant *iff for all $t, t' \in SMP(M) \setminus \mathcal{V}$ it holds that $\Gamma(t) = \Gamma(t')$ if t and t' are unifiable.*
2. *A strand (or constraint) $\mathcal{A}$ is* type-flaw resistant *iff $trms(\mathcal{A}) \cup setops(\mathcal{A})$ is type-flaw resistant, all bound variables of $\mathcal{A}$ have atomic type, and for any terms t, t' and variable sequences $\bar{x}$:*
   (a) *If $t \doteq t'$ occurs in $\mathcal{A}$ then $\Gamma(t) = \Gamma(t')$ if t and t' are unifiable.*
   (b) *If $\forall \bar{x}.\ t \not\doteq t'$, $\mathsf{insert}(t,t')$, $\mathsf{delete}(t,t')$, $t \dot{\in} t'$, or $\forall \bar{x}.\ t \dot{\notin} t'$ occurs in $\mathcal{A}$ then $\Gamma(fv(t) \cup fv(t')) \subseteq \mathfrak{T}_a$.*
   (c) *If $\mathsf{assert}(t)$ occurs in $\mathcal{A}$ then $t \notin \mathcal{V}$ and $\Gamma(fv(t)) \subseteq \mathfrak{T}_a$.*
   (d) *If $\mathsf{event}(t)$ occurs in $\mathcal{A}$ then $t \notin \mathcal{V}$.*
   (e) *If $\forall \bar{x}.\ \neg\mathsf{event}(t)$ occurs in $\mathcal{A}$ then $\Gamma(fv(t)) \subseteq \mathfrak{T}_a$.*
3. *A protocol $\mathcal{S}$ is* type-flaw resistant *iff the set $trms(\mathcal{S}) \cup setops(\mathcal{S})$ is type-flaw resistant and for all $\ell \in \mathcal{S}$ the strand ℓ is type-flaw resistant.*

The main type-flaw resistance condition is defined in Definition 9(1) and it states that matching pairs of messages that might occur in a protocol run must have the same type. For equality steps $t \doteq t'$ any solution $\mathcal{I}$ must be a unifier of t and t', and so they should have the same type. If $t \doteq t'$ is unsatisfiable (i.e., t and t' are not unifiable) then their types do not matter. Hence we can later prove that our reduction tr preserves type-flaw resistance, even if tr produces some unsatisfiable equality steps. For inequality steps $\forall \bar{x}.\ t \not\doteq t'$ we only need to require that the variables occurring in $\bar{x}$, t, and t' are atomic. For the remaining constraint steps note that when we translate a set operation such as $\mathsf{delete}(t,s)$ we construct steps of the form $(t,s) \doteq (t',s')$ and $(t,s) \not\doteq (t',s')$. Thus we must require all variables of t, t', s, and s' to be atomic, and if (t,s) and (t',s') are unifiable then they should have the same type. By requiring that the

set $trms(\mathcal{A}) \cup setops(\mathcal{A})$ is type-flaw resistant we have that the translated set operations must have the same type if they are unifiable. Similar conditions are needed for the event steps, but we can here relax the requirements slightly since their translations are simpler. Finally, a protocol is type-flaw resistant whenever its strands are, and we must additionally require here that $trms(\mathcal{S}) \cup setops(\mathcal{S})$ is type-flaw resistant because terms from different strands might be unifiable.

Note that if we allow for composed types for variables in inequalities then we can easily construct constraints which only have ill-typed solutions. For instance, consider the inequality $\forall x.\ y \neq f(x)$ where $\Gamma(y) = f(\Gamma(x))$. For any instance $f(c)$ of y where $\Gamma(f(c)) = \Gamma(y)$ there is an instance of x (namely c) that does not satisfy the inequality. Hence the constraint has no well-typed solution. However, there does exist ill-typed solutions; since we are working in the free algebra terms are equal if and only if they are syntactically equal, and hence any instance of y that is not of the form $f(c)$ for some c would be a solution to the inequality. [24] has no such restrictions on the type of universally quantified variables, and their typing result breaks because of this and other issues. We explain the other issues of [24] in Appendix B. Thus it seems that a typing result for stateful protocols necessarily requires a carefully restricted setting like our set-based approach.

As an example of type-flaw resistance we show that the keyserver protocol is type-flaw resistant. One approach to proving type-flaw resistance of a protocol $\mathcal{S}$ is to first find a set of strand steps M that subsumes the steps of $\mathcal{S}$ as well-typed instances. By proving type-flaw resistance of all steps in M, and of the set of terms occurring in M, we can conclude that $\mathcal{S}$ must be type-flaw resistant. For our example we can consider the following set, where $\Gamma(\{A, S, U\}) = \{\mathsf{agent}\}$ and $\Gamma(PK) = \mathsf{value}$:

$$
\begin{aligned}
M = \{&\mathsf{assert}(\mathsf{attack}), \mathsf{delete}(PK, \mathsf{valid}(U)),\\
&\forall A.\ PK \notin \mathsf{revoked}(A), \forall A.\ PK \dot{\notin} \mathsf{valid}(A),\\
&\mathsf{insert}(PK, \mathsf{valid}(U)), \mathsf{insert}(PK, \mathsf{ring}(U)),\\
&\mathsf{insert}(PK, \mathsf{revoked}(U)), PK \dot{\in} \mathsf{valid}(U), PK \dot{\in} \mathsf{ring}(U),\\
&\mathsf{receive}(\mathsf{inv}(PK)), \mathsf{receive}(\mathsf{sign}(\mathsf{inv}(PK), \langle U, PK \rangle)),\\
&\mathsf{send}(\mathsf{inv}(PK)), \mathsf{send}(PK), \mathsf{send}(\mathsf{sign}(\mathsf{inv}(PK), \langle U, PK \rangle))\}
\end{aligned}
$$

Hence all variables have atomic type and so the non-constant, non-variable sub-message patterns of M consist of the composed terms and subterms closed under well-typed variable renaming and well-typed instantiation of the variables with constants. It is easy to see that each pair of non-variable terms among these composed sub-message patterns have the same type if they are unifiable. Thus the total set of terms of the protocol—and in each strand—is type-flaw resistant.

What remains to be shown is that each strand step in M satisfy requirement 2(b) and 2(c) of Definition 9 (the remaining requirements are vacuously satisfied). The only event step occurring in M is $\mathsf{assert}(\mathsf{attack})$, and so 2(c) is satisfied. For the set operations occurring in M it is easy to see that the set terms are composed and only contains variables of atomic type, and that all elements PK of sets are of type value. Thus the final requirement, 2(b), is also satisfied.

In general, type-flaw resistance is in our opinion a reasonable property to require from protocols and their implementations: most importantly one should not have messages that encrypt raw data, like a nonce or a key, without any bit of information what the data means, because this opens the door for the intruder to reuse messages from honest agents that he cannot produce himself (and whose precise content he may not even know) in a different context. In fact, most concrete implementations satisfy this. Our result extends previous typing results in the scope of protocols that can be considered to stateful protocols; the type-flaw resistance requirement is thus also extended accordingly, however this is in some sense also conservative: all protocols that are type-flaw resistant according to the notion of [19] are also type-flaw resistant according to our Definition 9. In a nutshell, the additional requirements for set operations and events are simply to exclude that sets and events can be used as an "unchecked side-channel" where type-flaws attacks can creep in. The requirements on set operations are, in fact, only as strict as the requirements on inequalities and the tuples $(\cdot,\cdot)$ that arise in the translation tr. In particular, we support arbitrary types for set elements—the only restrictions being that the variables in set elements have atomic types and that unifiable set elements in the same set have the same type. Thus we support set elements of atomic types, composed types, and even non-homogeneous sets (i.e., sets containing elements of different types). In Appendix D we give further examples to illustrate that our notion works on real-world examples.

Finally, we prove that reachable constraints $\mathcal{A}$, and their translations $tr(\mathcal{A})$, are type-flaw resistant whenever the initial protocol is:

Lemma 4 (Type-flaw resistance preservation)**.** *If $\mathcal{S}_0$ is a type-flaw resistant protocol and $(\mathcal{S}_0; 0) \overset{w}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A})$ then both $\mathcal{S}$ and $\mathcal{A}$ are type-flaw resistant. Moreover, if $\mathcal{A}' \in tr(\mathcal{A})$ then $\mathcal{A}'$ is also type-flaw resistant.*

5.3 The Typing Result

All that remains is to prove the actual typing result for stateful protocols. By using our reduction tr together with existing typing results on ordinary constraints we can prove the following:

Theorem 3 (Typing result on symbolic constraints)**.** *If $\mathcal{A}$ is well-formed, $\mathcal{I} \models \mathcal{A}$, and $\mathcal{A}$ is type-flaw resistant, then there exists a well-typed interpretation $\mathcal{I}_\tau$ such that $\mathcal{I}_\tau \models \mathcal{A}$.*

With this intermediate result we can prove our main theorem:

Theorem 4 (Typing result for stateful protocols)**.** *If $\mathcal{S}_0$ is a type-flaw resistant protocol, and $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \overset{w}{\Longrightarrow}^* (\mathcal{S}; M, D, E)$ where $w = (\sigma_1, \ell_1), \ldots, (\sigma_k, \ell_k)$ then there exists a state $(\mathcal{S}; M', D', E')$ such that $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \overset{w'}{\Longrightarrow}^* (\mathcal{S}; M', D', E')$ where $w' = (\sigma'_1, \ell_1), \ldots, (\sigma'_k, \ell_k)$ for some well-typed ground substitutions $\sigma'_1, \ldots, \sigma'_k$.*

6 Case Studies

In this section we discuss how our typing result is applicable in practice on several protocols, in particular that many protocols already satisfy the requirements of type-flaw resistance or require only minor changes to do so.

Due to lack of space we will only consider an extension of the keyserver example here. In Appendix D we also consider the examples from the AIF and AIF-ω tools (some of those examples have similarly been considered in SAPIC, in particular PKCS#11 and ASW, but in a way that violates the corresponding type flaw-resistance requirements).

6.1 Automatically Checking Type-Flaw Resistance

One crucial point of the typing result is that it is relatively easy to check, namely by statically looking at the format of messages rather than traversing the entire state space, and that this can also be done automatically as a static analysis of a user's specification before verification in a typed model.

Note that $SMP(M)$ is in general infinite, but it is sufficient to check the following finite representation SMP_0 for type-flaw resistance: starting with $SMP_0 = M$, we first ensure that for every message $t \in SMP_0$ that contains a variable x of a composed type $f(\tau_1, \ldots, \tau_n)$, we ensure that also $[x \mapsto f(x_1, \ldots, x_n)](t) \in SMP_0$ for some variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ that do not occur in t. (Even if some τ_i are themselves composed types, this can be done by adding finitely many messages, since all type expressions are finite terms.) Next, we close SMP_0 under subterms and key terms of Ana. Finally, let us ensure by well-typed α-renaming that all terms in SMP_0 have pairwise disjoint variables. Note that SMP_0 is a representation of $SMP(M)$ in the sense that every $SMP(M)$ term is a well-typed instance of an SMP_0 term. Now the condition that every pair $s, t \in SMP_0 \setminus \mathcal{V}$ with $\Gamma(s) \neq \Gamma(t)$ has no unifier, is equivalent to the type-flaw resistance of M. *Proof sketch:* Note that $SMP_0 \subseteq SMP(M)$, giving us one direction of the equivalence. For the other direction, suppose there are any $s, t \in SMP(M)$ such that $\Gamma(s) \neq \Gamma(t)$. Since SMP_0 represents $SMP(M)$, there exists terms $s_0, t_0 \in SMP_0$ and well-typed substitutions θ_1 and θ_2 such that $s = \theta_1(s_0)$ and $t = \theta_2(t_0)$. Hence also $\Gamma(s_0) \neq \Gamma(t_0)$, and so s_0 and t_0 are not unifiable by assumption. Thus s and t cannot be unified as well because s_0 and t_0 do not share variables.

6.2 Extension of the Keyserver Example

We will now illustrate by a small example how type-flaw problems can arise in practice, how type-flaw resistance is violated in such a case, and how the situation can be fixed. Suppose for the key server example, we augment the protocol with an exchange where a user can prove to be alive, formalized by having for each user u and each session $j \in \mathbb{N}$ the following transaction strand:

$$\mathsf{receive}(N_j).PK_{u,j} \doteq \mathsf{ring}(u).\mathsf{send}(\mathsf{sign}(\mathsf{inv}(PK_{u,j}), N_j))$$

where all N_j and $PK_{u,j}$ have atomic types. The idea is that anybody can send the user a challenge N_j, and u answers with a signature on it. In this blunt form it is obviously a bad idea, since an intruder can send an arbitrary term instead of N_j. Indeed the protocol now violates type-flaw resistance: $\mathsf{sign}(\mathsf{inv}(PK_{u,j}), N_j)$ has a unifier with the normal update message $\mathsf{sign}(\mathsf{inv}(PK_i), \langle U_i, NPK_i \rangle)$, while they have different types. The general recommendation is thus to use some form of tag to indicate what the messages should mean. In fact, many protocol standards already describe a concrete message format, e.g., in this case that nonces and public keys have certain byte lengths, or even fields that indicate the length, if it is not fixed; in contrast many protocol models model only abstractly the exchanged information as a tuples. It is thus recommended to model the concrete message formats by transparent functions, i.e., functions like pair that the intruder can compose and decompose, and check that the concrete formats of the protocol standard are disjoint so that a confusion is impossible. In this case we may have functions $\mathsf{update}(U, PK)$ that is used in the update message and functions $\mathsf{challenge}(N)$ and $\mathsf{response}(N)$ to model the challenge response protocol to have rather the following form:

$$\mathsf{receive}(\mathsf{challenge}(N_j)).$$
$$PK_{u,j} \dot{\in} \mathsf{ring}(u).\mathsf{send}(\mathsf{sign}(\mathsf{inv}(PK_{u,j}), \mathsf{response}(N_j)))$$

One may argue that the formatting of challenge message is irrelevant since it is in cleartext. We suggest, however, to use formatting information also here, since it is in fact good practice for implementations anyway and does not really hurt.

With the change we now have again type-flaw resistance and our typing result is applicable.

7 Connections to Other Formalisms

We have introduced the formalism of transaction strands to have a simple and mathematically pure formalism as a protocol model for our result without the disturbance of the many technical details of various protocol models. We want to illustrate now that our result can nonetheless be used in various protocol models, but we only sketch the main ideas.

We only consider AIF-ω here, but in Appendix E we also discuss Maude-NPA, Set-π, and process calculi in general.

Note that the core of our result is proved on symbolic constraints (intruder strands) of a symbolic transition system. Connecting another formalism with our typing result requires only two aspects. First, one needs to define the semantics for the formalism in terms of a symbolic transition system with constraints (including set operations, equalities, and inequalities). Second, one needs to transfer the notion of type-flaw resistance, so that a type-flaw resistant specification in the formalism will only produce type-flaw resistant constraints. We have done this for transaction strands with detailed proofs. Due to the vari-

ety of other formalisms and their technical details, we only sketch here and in Appendix E the ideas for the most common constructions.

Our transaction strands are in some sense a purified version of AIF-ω. In a nutshell, it describes protocols by a set of rewrite rules for a state transition system, where each state is a set of *facts* like $\mathsf{ik}(m)$ to denote that the intruder knows message m. It is thus also similar to other rewriting based languages like Maude-NPA or the AVANTSSAR ASLan.

One can translate each AIF-ω rule into transaction strands as follows. Every intruder knowledge fact $\mathsf{ik}(m)$ on the left hand side of a rule corresponds to receiving a message m, and on the right hand side to sending a message m. If the expression t in s occurs on the left-hand side, then the transaction strand must contain $t \dot{\in} s$; if the same expression does not occur on the right-hand side, then the transaction must include $\mathsf{delete}(t, s)$. If the expression t notin s occurs on the left-hand side, then the transaction must contain $\forall \bar{x}.\, t \dot{\notin} s$ where $\bar{x}$ are the variables that on the left-hand side only occur in notin expressions. Finally, if t in s occurs on the right-hand side but not on the left, then the transaction must include $\mathsf{insert}(t, s)$. All other facts of AIF-ω are persistent (i.e., once true, they remain true in all successor states), therefore we can model them as events in transaction strands, using $\mathsf{event}(f)$ for the left-hand side facts and $\mathsf{assert}(f)$ for right-hand side facts. Note that the order of all these actions in the transaction matters: first we should have all receiving messages, checking for events and set memberships, then modifying sets and sending the outgoing messages. Still one may wonder what happens in the following AIF-ω rule: x in $s.y$ in $s \Rightarrow x$ in s. If $x = y$ then this rule is contradictory, and the semantics of AIF-ω excludes such substitutions. For that reason, we also have to include the inequality $x \neq y$ to the transaction to exactly follow the AIF-ω semantics. In all remaining cases the inner order of the actions is actually irrelevant, but these subtle points where one of the motivations to introduce transaction strands. Finally, note that the rules from AIF-ω may have variables that represent any value from a countable set of constants, as well as the creation of fresh values. Since transaction strands do not have a mechanism for creating fresh values and free variables are not allowed, one must instantiate these variables appropriately, producing a countable set of transaction strands from finitely many rules.

With this translation from AIF-ω rules to transaction strands, we also directly obtain a semantics using symbolic constraints and actually immediately transfer the notion of type-flaw resistance from transaction strands with the obvious adaptations. However, type-flaw resistance will not be directly satisfied for typical AIF-ω specifications immediately, because they would contain rules for the intruder that contain untyped variables. While for honest agents, it is not a restriction to declare the *intended* type for each variable, the intruder deduction rules should be applicable to messages of *any* type. Thus, we have to make the reservation that the intruder deduction of an AIF-ω specification must be within the bounds of the intruder model we have used here, namely composition with public functions and decomposition according to an Ana theory. This is indeed possible for all the standard operators like symmetric and asymmetric encryption, signatures, hashes, and transparent functions like pair;

operators that require algebraic equations like xor are however not supported.

8 Conclusion

Over the past years, several typing results have emerged for security protocols, gradually extending the class of protocols that can be supported, in particular [18, 2, 1, 19]. A common idea for proving such typing results is to use a notion of symbolic constraints to represent executions (in particular attacks) and show that whenever there is a solution then there is a well-typed one. The requirement that the protocols have to fulfill for such a result is only that all messages of different intended type have sufficiently different structure to never be confused. This is fulfilled by many common protocols like a standard setup of TLS [19].

One relevant trend in protocol security is the support for stateful protocols, i.e., protocols in which participants can manipulate a global state that is shared among an unbounded number of sessions. This is for instance relevant to model security devices like key tokens or servers that maintain a database. There is only one typing result so far that supports stateful protocols, namely [24]. We point out several mistakes of this paper in Appendix B, showing that their results do not hold in this generality. A particular problem are variables of composed types in negative conditions, which illustrates that typing results for stateful systems are far more subtle than intuition suggests. Our main contribution of this paper is to establish the first precise typing result for a class of stateful protocols. Despite a meticulous formalization it is conceptually still quite simple, as it is based on a reduction to the existing typing results, in particular the formalization of [19].

Our typing result conservatively extends existing ones, i.e., for stateless protocols we do not require any further restrictions. The restrictions on set operations are similar to those on messages, but additionally, we have to limit here the use of variables of composed types (unless negative operations are not needed for a set). In fact, the condition of our typing result is satisfied by most examples distributed with the AIF-ω tool [25], and in the remaining cases a simple disambiguation of messages is sufficient.

Besides the trend towards the verification of more complex stateful protocols that this typing result focuses on, there are other crucial trends like the verification of privacy-type goals using equivalence properties, and typing results in this direction have been established [13]. A question for future research is thus if statefulness and equivalence proofs can be combined. Another closely related area are compositionality results that can often benefit from typing results, for instance in [1]. Establishing compositionality for stateful protocols is another interesting direction for future research.

References

[1] O. Almousa, S. Mödersheim, P. Modesti, and L. Viganò. Typing and compositionality for security protocols: A generalization to the geometric fragment. In *ESORICS 2015*, pages 209–229, 2015.

[2] M. Arapinis and M. Duflot. Bounding messages for free in security protocols - extension to various security properties. *Inf. Comput.*, 239:182–215, 2014.

[3] M. Arapinis, E. Ritter, and M. Ryan. Statverif: Verification of stateful processes. In *CSF 2011*, pages 33–47. IEEE, 2011.

[4] 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*, pages 267–282, 2012.

[5] A. Armando and L. Compagna. Sat-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008.

[6] D. A. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *Int. J. Inf. Sec.*, 4(3):181–208, 2005.

[7] G. Bella. *Formal Correctness of Security Protocols - With 62 Figures and 4 Tables*. Information Security and Cryptography. Springer, 2007.

[8] G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET purchase protocols. *J. Autom. Reasoning*, 36(1-2):5–37, 2006.

[9] B. Blanchet and A. Podelski. Verification of cryptographic protocols: tagging enforces termination. *Theor. Comput. Sci.*, 333(1-2):67–90, 2005.

[10] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing pkcs#11 security tokens. In *CCS 2010*, pages 260–269, 2010.

[11] A. D. Brucker and S. Mödersheim. Integrating automated and interactive protocol verification. In *FAST 2009*, 2009.

[12] A. Bruni, S. Mödersheim, F. Nielson, and H. R. Nielson. Set-pi: Set membership p-calculus. In *CSF 2015*, pages 185–198, 2015.

[13] R. Chrétien, V. Cortier, and S. Delaune. Typing messages for free in security protocols: The case of equivalence properties. In P. Baldan and D. Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2014.

[14] V. Cortier and S. Delaune. Safely composing security protocols. *Formal Methods in System Design*, 34(1):1–36, 2009.

[15] C. Cremers and S. Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012.

[16] S. B. Fröschle and G. Steel. Analysing pkcs#11 key management apis with unbounded fresh data. In *ARSPA-WITS 2009*, pages 92–106, 2009.

[17] J. D. Guttman. State and progress in strand spaces: Proving fair exchange. *J. Autom. Reasoning*, 48(2):159–195, 2012.

[18] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[19] A. V. Hess and S. Mödersheim. Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL. In *CSF 2017*, 2017.

[20] S. Kremer and R. Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

[21] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *CAV 2013*, 2013.

[22] J. K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *CCS 2001*, 2001.

[23] S. Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *CCS 2010*, 2010.

[24] S. Mödersheim. Deciding Security for a Fragment of ASLan. In *ESORICS*, pages 127–144. Springer, 2012.

[25] S. Mödersheim and A. Bruni. Aif-ω: Set-based protocol abstraction with countable families. In *POST 2016*, 2016.

[26] S. Mödersheim and P. Modesti. Verifying sevecom using set-based abstraction. In *IWCMC 2011*, pages 1164–1169, 2011.

[27] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[28] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, 1999.

[29] M. Rusinowitch and M. Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *Theor. Comput. Sci.*, 299, 2003.

[30] SEVECOM project. Deliverable 2.1-App. A Baseline Security Specification, 2009. `https://www.sevecom.eu/Deliverables/Sevecom_Deliverable_D2.1-App.A_v1.2.pdf`.

[31] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1):191–230, 1999.

A Proofs

This section contains the proofs of our technical results. Note that many of the theorems and lemmas are more general versions of the ones found in the paper. For the proofs we make use the following definition of *substitution composition*: Given substitutions δ and σ the composition $\delta \cdot \sigma$ is defined as the substitution $\lambda x.\ \sigma(\delta(x))$.

A.1 Constraint Semantics

Lemma 1. *Given a ground set of terms M, a ground database mapping D, a ground set E of asserted events, an interpretation $\mathcal{I}$, and symbolic constraints $\mathcal{A}$ and $\mathcal{A}'$, the following holds:*

$$\mathcal{I} \models_{M,D,E} \mathcal{A}.\mathcal{A}' \text{ if and only if } (\mathcal{I} \models_{M,D,E} \mathcal{A} \text{ and } \mathcal{I} \models_{M \cup ik(\mathcal{I}(\mathcal{A})), db(\mathsf{insert}(D).\mathcal{I}(\mathcal{A})), E \cup ev(\mathcal{I}(\mathcal{A}))} \mathcal{A}')$$

Proof. Each direction of the biconditional follows easily by an induction on the leftmost constraint $\mathcal{A}$. □ □

Lemma 2 *Given a ground state $(\mathcal{S}; M, D, E)$, a transaction strand $\ell \in \mathcal{S}$ (condition C1), and a ground substitution σ with domain $fv(\ell)$ (condition C2), then the conditions C3 to C9 hold if and only if $\sigma \models_{M,D,E} dual(\ell)$.*

Proof. Let $\ell = \mathsf{receive}(T).S.\mathsf{send}(T')$ where S does not contain send and $\mathsf{receive}$ steps and observe that $dual(\ell) = \mathsf{send}(T).dual(S).\mathsf{receive}(T')$. Condition C3 then corresponds to $\sigma \models_{M,D,E} \mathsf{send}(T)$, condition C4 to C9 to $\sigma \models_{M,D,E} dual(S)$, and the remaining part $\mathsf{receive}(T')$ is irrelevant as it is satisfied for any M, D, E, and σ. Thus C3 to C9 hold if and only if $\sigma \models_{M,D,E} dual(\ell)$. □ □

A.2 Transition Systems

Lemma 3 (Well-formedness of reachable symbolic constraints). *If $\mathcal{S}_0$ is a well-formed protocol and $(\mathcal{S}_0; 0) \overset{w}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A})$ then $\mathcal{A}$ is a well-formed symbolic constraint and $\mathcal{S}$ is a well-formed protocol.*

Proof. By induction on reachability. The base case (i.e., the symmetric case) is trivial. For the inductive case assume that $(\mathcal{S}_0; 0) \overset{w'}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A}) \overset{\ell}{\Longrightarrow}^{\bullet} (\mathcal{S} \setminus \{\ell\}; \mathcal{A}.\mathcal{A}')$ where $\mathcal{A}$ and $\mathcal{S}$ are well-formed by the induction hypothesis. Let $\ell = \mathsf{receive}(T).S.\mathsf{send}(T')$, where S does not contain further $\mathsf{receive}$ and send steps, then $\mathcal{A}' = dual(\ell) = \mathsf{send}(T).S.\mathsf{receive}(T')$ by definition. Since $\mathcal{S} \setminus \{\ell\} \subseteq \mathcal{S}$ and $\mathcal{S}$ is well-formed we have that $\mathcal{S}$ must also be well-formed. Since all strands in $\mathcal{S}_0$ are variable-disjoint we also have that $fv(\mathcal{A}) \cap fv(\mathcal{A}') = \emptyset$. Hence $\mathcal{A}.\mathcal{A}'$

is well-formed if $\mathcal{A}$ is well-formed and $\mathcal{A}'$ is well-formed. The prefix $\mathcal{A}$ is well-formed by the induction hypothesis, and since $\ell \in \mathcal{S}$ we know that $dual(\ell) = \mathcal{A}'$ is well-formed as well. Thus we can conclude the case. □ □

Theorem 1 (Equivalence of transition systems). *Let $\mathcal{S}_0$ be a protocol and let $\{\ell_1, \ldots, \ell_k\} \subseteq \mathcal{S}_0$. Then:*

1. *If $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \stackrel{w}{\Longrightarrow}^* (\mathcal{S}; M, D, E)$ where $w = (\sigma_1, \ell_1), \ldots, (\sigma_k, \ell_k)$ then*
   (a) $(\mathcal{S}_0; 0) \stackrel{w'}{\Longrightarrow}^{\bullet *} (\mathcal{S}; dual(\ell_1).\cdots.dual(\ell_k))$ *where* $w' = \ell_1, \ldots, \ell_k$,
   (b) $\sigma_1 \cdot \ldots \cdot \sigma_k \models dual(\ell_1).\cdots.dual(\ell_k)$,
   (c) $M = ik((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k)))$,
   (d) $D = db((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k)))$, *and*
   (e) $E = ev((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k)))$.
2. *If $(\mathcal{S}_0; 0) \stackrel{w}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A})$ and $\mathcal{I} \models \mathcal{A}$ where $w = \ell_1, \ldots, \ell_k$, $dom(\mathcal{I}) = fv(\mathcal{A})$, and $\mathcal{I}$ is ground, then there exists substitutions $\sigma_1, \ldots, \sigma_k$ such that*
   (a) $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \stackrel{w'}{\Longrightarrow}^* (\mathcal{S}; ik(\mathcal{I}(\mathcal{A})), db(\mathcal{I}(\mathcal{A})), ev(\mathcal{I}(\mathcal{A})))$ *where* $w' = (\sigma_1, \ell_1), \ldots, (\sigma_k, \ell_k)$,
   (b) $\mathcal{A} = dual(\ell_1).\cdots.dual(\ell_k)$,
   (c) $dom(\sigma_i) = fv(\ell_i)$ *for all* $i \in \{1, \ldots, k\}$, *and*
   (d) $\mathcal{I} = \sigma_1 \cdot \ldots \cdot \sigma_k$

Proof. 1. We prove the first implication by an induction on reachability.

The base case (i.e. the symmetric case) follows easily from the assumptions.

So, in the inductive case, assume that

$$\begin{array}{rcl} (\mathcal{S}_0; \emptyset, \emptyset, \emptyset) & \stackrel{w}{\Longrightarrow}^* & (\mathcal{S}; M, D, E) \\ & \stackrel{w_{k+1}}{\Longrightarrow} & (\mathcal{S} \setminus \{\ell_{k+1}\}; M \cup \sigma_{k+1}(T'), \\ & & \quad db(\mathsf{insert}(D).\sigma_{k+1}(S)), \\ & & \quad E \cup ev(\sigma_{k+1}(S))) \end{array}$$

where $\ell_{k+1} = \mathsf{receive}(T).S.\mathsf{send}(T') \in \mathcal{S}$ and $w = (\sigma_1, \ell_1), \ldots, (\sigma_k, \ell_k)$ and $w_{k+1} = (\sigma_{k+1}, \ell_{k+1})$. We furthermore assume the induction hypothesis:

(H1) $(\mathcal{S}_0; 0) \stackrel{w'}{\Longrightarrow}^{\bullet *} (\mathcal{S}; dual(\ell_1).\cdots.dual(\ell_k))$ where $w' = \ell_1, \ldots, \ell_k$,
(H2) $\sigma_1 \cdot \ldots \cdot \sigma_k \models dual(\ell_1).\cdots.dual(\ell_k)$,
(H3) $M = ik((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k)))$,
(H4) $D = db((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k)))$, and
(H5) $E = ev((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k)))$.

In the remaining proof for this case we will use the abbreviations $\mathcal{I} = \sigma_1 \cdot \ldots \cdot \sigma_{k+1}$ and $\mathcal{A} = dual(\ell_1).\cdots.dual(\ell_{k+1})$. We will now prove each of the five parts of the thesis for this case:

- From (H1) and $\ell_{k+1} \in \mathcal{S}$ we can apply $\stackrel{\ell_{k+1}}{\Longrightarrow}{}^{\bullet}$ and immediately conclude part (a) of the thesis, namely $(\mathcal{S}_0; 0) \stackrel{w', \ell_{k+1}}{\Longrightarrow}{}^{\bullet *} \quad (\mathcal{S} \backslash \{\ell_{k+1}\}; \mathcal{A})$.
- From the assumption and Lemma 2 we know that $\sigma_{k+1} \models_{M,D,E} dual(\ell_{k+1})$ and since all strands of a protocol must be pairwise variable-disjoint we furthermore know that $(\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_{k+1})) = dual(\ell_{k+1})$. Hence $\mathcal{I} \models_{M,D,E} dual(\ell_{k+1})$ because $dom(\sigma_i) \cap dom(\sigma_j) = \emptyset$ for all $i, j \in \{1, \ldots, k+1\}, i \neq j$ since $dom(\sigma_i) = fv(\ell_i)$ for all $i \in \{1, \ldots, k+1\}$. Likewise, we get $\mathcal{I} \models dual(\ell_1).\cdots.dual(\ell_k)$ from (H2). All together we can then conclude part (b), namely $\mathcal{I} \models \mathcal{A}$.
- Note that $dual(\mathsf{receive}(T).S.\mathsf{send}(T')) = \mathsf{send}(T).dual(S).\mathsf{receive}(T')$ and so $ik(dual(\ell_{k+1})) = T'$. Together with the variable disjointedness of the strands and (H3) we have:

$$\begin{aligned}
&M \cup \sigma_{k+1}(T') \\
&= ik((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k))) \cup \sigma_{k+1}(T') \\
&= ik(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k))) \cup \mathcal{I}(T') \\
&= ik(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k))) \cup ik(\mathcal{I}(dual(\ell_{k+1}))) \\
&= ik(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k).dual(\ell_{k+1})))
\end{aligned}$$

  which proves the third part of the case.
- Note that $dual(S) = S$ and therefore $db(\sigma_{k+1}(S)) = db(\sigma_{k+1}(dual(\ell_{k+1})))$. Together with the variable disjointedness and (H4) we then have:

$$\begin{aligned}
&db(\mathsf{insert}(D).\sigma_{k+1}(S)) \\
&= db(\mathsf{insert}(D).\mathcal{I}(dual(\ell_{k+1}))) \\
&= db(\mathsf{insert}(db(\mathcal{I}(dual(\ell_1).\cdots. \\
&\qquad\qquad dual(\ell_k)))).\mathcal{I}(dual(\ell_{k+1}))) \\
&= db(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k).dual(\ell_{k+1})))
\end{aligned}$$

  which proves the fourth part of the case.
- The fifth and final part of the case is proven similarly to the third part and using (H5):

$$\begin{aligned}
&E \cup \sigma_{k+1}(ev(S)) \\
&= ev((\sigma_1 \cdot \ldots \cdot \sigma_k)(dual(\ell_1).\cdots.dual(\ell_k))) \\
&\quad \cup \sigma_{k+1}(ev(S)) \\
&= ev(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k))) \cup \mathcal{I}(ev(S)) \\
&= ev(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k))) \\
&\quad \cup ev(\mathcal{I}(dual(\ell_{k+1}))) \\
&= ev(\mathcal{I}(dual(\ell_1).\cdots.dual(\ell_k).dual(\ell_{k+1})))
\end{aligned}$$

Thus we have proven all parts of the thesis for the inductive case.

2. We also prove the second implication by an induction on reachability. Again, the base case is trivial.

   For the inductive case we assume that

$$\begin{array}{rll}(\mathcal{S}_0; 0) & \stackrel{w}{\Longrightarrow}{}^{\bullet *} & (\mathcal{S}; \mathcal{A}) \\ & \stackrel{\ell_{k+1}}{\Longrightarrow}{}^{\bullet} & (\mathcal{S} \setminus \{\ell_{k+1}\}; \mathcal{A}.\mathit{dual}(\ell_{k+1}))\end{array}$$

   and $\mathcal{I} \models \mathcal{A}.\mathit{dual}(\ell_{k+1})$ where $w = \ell_1, \ldots, \ell_k$, $\mathit{dom}(\mathcal{I}) = \mathit{fv}(\mathcal{A}.\mathit{dual}(\ell_{k+1}))$, and $\mathcal{I}$ is ground. Now obtain the unique σ_{k+1} and $\mathcal{I}'$ such that $\mathit{dom}(\sigma_{k+1}) = \mathit{fv}(\mathit{dual}(\ell_{k+1}))$, $\mathit{dom}(\mathcal{I}') = \mathit{fv}(\mathcal{A})$, and $\mathcal{I} = \mathcal{I}' \cdot \sigma_{k+1}$. This is always possible since the domain of $\mathcal{I}$ is exactly the free variables of $\mathcal{A}.\mathit{dual}(\ell_{k+1})$ and $\mathcal{I}$ is ground and since all strands of $\mathcal{S}_0$ are pairwise-variable disjoint. Hence we have that $\mathcal{I}' \models \mathcal{A}$ and $\sigma_{k+1} \models_{\mathit{ik}(\mathcal{I}'(\mathcal{A})), \mathit{db}(\mathcal{I}'(\mathcal{A})), \mathit{ev}(\mathcal{I}'(\mathcal{A}))} \mathit{dual}(\ell_{k+1})$, the former of which is the premise to the induction hypotheses for this case, and we can therefore apply the induction hypotheses to get

   (H1) $(\mathcal{S}_0; \emptyset, \emptyset) \stackrel{w'}{\Longrightarrow}{}^{*} (\mathcal{S}; \mathit{ik}(\mathcal{I}'(\mathcal{A})), \mathit{db}(\mathcal{I}'(\mathcal{A})), \mathit{ev}(\mathcal{I}'(\mathcal{A})))$
   where $w' = (\sigma_1, \ell_1), \ldots, (\sigma_k, \ell_k)$,

   (H2) $\mathcal{A} = \mathit{dual}(\ell_1).\cdots.\mathit{dual}(\ell_k)$,

   (H3) $\mathit{dom}(\sigma_i) = \mathit{fv}(\ell_i)$ for all $i \in \{1, \ldots, k\}$, and

   (H4) $\mathcal{I}' = \sigma_1 \cdot \ldots \cdot \sigma_k$

   This basically lets us immediately prove the second, third, and fourth part of the thesis for this case, namely

   - $\mathcal{A}.\mathit{dual}(\ell_{k+1}) = \mathit{dual}(\ell_1).\cdots.\mathit{dual}(\ell_k).\mathit{dual}(\ell_{k+1})$,
   - $\mathit{dom}(\sigma_{k+1}) = \mathit{fv}(\ell_i)$ for all $i \in \{1, \ldots, k+1\}$, and
   - $\mathcal{I} = \sigma_1 \cdot \ldots \cdot \sigma_k \cdot \sigma_{k+1}$

   All that remains to be proven is

$$(\mathcal{S}_0; \emptyset, \emptyset) \stackrel{w''}{\Longrightarrow}{}^{*} (\mathcal{S} \setminus \{\ell_{k+1}\}; \mathit{ik}(\mathcal{I}(\mathcal{A}.\mathit{dual}(\ell_{k+1}))), \mathit{db}(\mathcal{I}(\mathcal{A}.\mathit{dual}(\ell_{k+1}))), \mathit{ev}(\mathcal{I}(\mathcal{A}.\mathit{dual}(\ell_{k+1}))))$$

   where $w'' = w', (\sigma_{k+1}, \ell_{k+1})$.

   From the first induction hypothesis (H1), Lemma 2, and $\sigma_{k+1} \models_{\mathit{ik}(\mathcal{I}'(\mathcal{A})), \mathit{db}(\mathcal{I}'(\mathcal{A})), \mathit{ev}(\mathcal{I}'(\mathcal{A}))} \mathit{dual}(\ell_{k+1})$ we can apply $\stackrel{(\sigma_{k+1}, \ell_{k+1})}{\Longrightarrow}$ to get

$$(\mathcal{S}_0; \emptyset, \emptyset) \stackrel{w''}{\Longrightarrow}{}^{*} (\mathcal{S} \setminus \{\ell_{k+1}\}; \mathit{ik}(\mathcal{I}'(\mathcal{A})) \cup \sigma_{k+1}(T'), \mathit{db}(\mathsf{insert}(\mathit{db}(\mathcal{I}'(\mathcal{A}))).\sigma_{k+1}(S)), \mathit{ev}(\mathcal{I}'(\mathcal{A})) \cup \mathit{ev}(\sigma_{k+1}(S)))$$

   where $\ell_{k+1} = \mathsf{receive}(T).S.\mathsf{send}(T')$. Hence we only need to prove that

- $ik(\mathcal{I}'(\mathcal{A})) \cup \sigma_{k+1}(T') = ik(\mathcal{I}(\mathcal{A}.dual(\ell_{k+1})))$,
- $db(\mathsf{insert}(db(\mathcal{I}'(\mathcal{A}))).\sigma_{k+1}(S)) = db(\mathcal{I}(\mathcal{A}.dual(\ell_{k+1})))$, and
- $ev(\mathcal{I}'(\mathcal{A})) \cup \sigma_{k+1}(S) = ev(\mathcal{I}(\mathcal{A}.dual(\ell_{k+1})))$ and

and this is proven similarly to how we proved the third to fifth part of the previous case. Thus we have proven all four conjuncts—(a), (b), (c), and (d)—of the conclusion for this inductive case.

□ □

A.3 Constraint Reduction

We prove Theorem 2 in two steps. First, we prove the second part of the theorem, namely that the translation preserves well-formedness. Secondly, we prove the equivalence, i.e., the first part of the theorem.

Theorem 2(2) (Well-formedness of translation). *Let X be a set of variables, D be a database mapping, and E be a finite set of events such that $fv(D) \cup fv(E) \subseteq X$. If $\mathcal{A}$ is well-formed w.r.t. the variables X and $\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ then $\mathcal{A}'$ is well-formed w.r.t. X.*

Proof. We prove the statement by an induction on $tr_{D,E}(\mathcal{A})$:

- Case $tr_{D,E}(0)$: Trivially true by definition of well-formedness.
- Cases $tr_{D,E}(\mathsf{send}(t).\mathcal{A})$, $tr_{D,E}(\mathsf{receive}(t).\mathcal{A})$, $tr_{D,E}(t \doteq t'.\mathcal{A})$, and $tr_{D,E}((\forall \bar{x}.\, t \not\doteq t').\mathcal{A})$: Follows easily from the induction hypotheses.
- Case $tr_{D,E}(\mathsf{insert}(t,s).\mathcal{A})$: From the premises we have that $fv(t) \cup fv(s) \subseteq X$ because $\mathsf{insert}(t,s).\mathcal{A}$ is well-formed w.r.t. X. Hence $\mathcal{A}$ is well-formed w.r.t. X and $fv(D \cup \{(t,s)\}) = fv(D) \subseteq X$ as well. Thus we can apply the induction hypothesis to $\mathcal{A}' \in tr_{D\cup\{(t,s)\},E}(\mathcal{A})$ and conclude the case.
- Case $tr_{D,E}(\mathsf{delete}(t,s).\mathcal{A})$: Note that $fv(D \setminus \{d_1,\ldots,d_i\}])) \subseteq fv(D) \subseteq X$. The remaining part of this case now follows from a similar argument to the one given in the previous case.
- Case $tr_{D,E}(t \mathrel{\dot\in} s.\mathcal{A})$: By the premises we have that $\mathcal{A}$ is well-formed w.r.t. $X \cup fv(t) \cup fv(s)$. From the induction hypothesis we then have that $\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ is also well-formed w.r.t. $X \cup fv(t) \cup fv(s)$. Thus $(t,s) \doteq d.\mathcal{A}'$ is well-formed w.r.t. X.
- Case $tr_{D,E}((\forall \bar{y}.\, t \mathrel{\dot\notin} s).\mathcal{A})$: The constraints $\forall \bar{y}.\, t \mathrel{\dot\notin} s$ and $(\forall \bar{y}.\, (t,s) \not\doteq d_1).\cdots.(\forall \bar{y}.\, (t,s) \not\doteq d_n)$ have the same well-formedness requirement. Thus the case follows straightforwardly from the induction hypothesis.
- Case $tr_{D,E}(\mathsf{event}(t).\mathcal{A})$: This case is similar to the $tr_{D,E}(t \mathrel{\dot\in} s.\mathcal{A})$ case.

- Case $tr_{D,E}((\forall \bar{y}.\ \neg \mathsf{event}(t)).\mathcal{A})$: This case is similar to the $tr_{D,E}((\forall \bar{y}.\ t \notin s).\mathcal{A})$ case.

□ □

Theorem 2(1) (Semantic equivalence of constraints and their translation). *Assume $fv(D) \cup fv(E)$ to be disjoint from the bound variables of $\mathcal{A}$. Then $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}$ iff there exists $\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ such that $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$.*

Proof. Consider the following two statements which together are equivalent to the original biconditional:

1. If $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}$ then there exists $\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ such that $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$.
2. If $\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$ then $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}$.

We prove the first of these implications by an induction on $\mathcal{A}$:

- Case 0: Trivially true.
- Cases $\mathsf{send}(t).\mathcal{A}$, $\mathsf{receive}(t).\mathcal{A}$, $t \doteq t'.\mathcal{A}$, and $(\forall \bar{x}.\ t \neq t').\mathcal{A}$: Follows straightforwardly from the induction hypotheses.
- Case $\mathsf{insert}(t,s).\mathcal{A}$: So $\mathcal{I} \models_{M,\mathcal{I}(D)\cup\{\mathcal{I}((t,s))\},\mathcal{I}(E)} \mathcal{A}$. Since $\mathcal{I}(D)\cup\{\mathcal{I}((t,s))\} = \mathcal{I}(D \cup \{(t,s)\})$ we can apply the induction hypothesis to obtain $\mathcal{A}'$ where $\mathcal{A}' \in tr_{D\cup\{(t,s)\},E}(\mathcal{A})$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$. Thus $\mathcal{A}' \in tr_{D,E}(\mathsf{insert}(t,s).\mathcal{A})$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$.
- Case $\mathsf{delete}(t,s).\mathcal{A}$: Hence $\mathcal{I} \models_{M,\mathcal{I}(D)\setminus\{\mathcal{I}((t,s))\},\mathcal{I}(E)} \mathcal{A}$. Now partition D into the sets $\{d_1,\ldots,d_i\}$ and $\{d_{i+1},\ldots,d_n\}$ for some $0 \leq i \leq n$ such that $\mathcal{I}(\{(t,s)\}) = \mathcal{I}(\{d_1,\ldots,d_i\})$ and $\mathcal{I}((t,s)) \notin \mathcal{I}(\{d_{i+1},\ldots,d_n\})$. Then

$$\begin{aligned}&\mathcal{I}(D) \setminus \mathcal{I}(\{(t,s)\})\\ &\quad= \mathcal{I}(D) \setminus \mathcal{I}(\{d_1,\ldots,d_i\})\\ &\quad= \mathcal{I}(\{d_{i+1},\ldots,d_n\})\\ &\quad= \mathcal{I}(D \setminus \{d_1,\ldots,d_i\})\end{aligned}$$

  We can then apply the induction hypothesis to obtain $\mathcal{A}' \in tr_{D\setminus\{d_1,\ldots,d_i\},E}(\mathcal{A})$ such that $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$. Now let $\mathcal{B} = (t,s) \doteq d_1.\cdots.(t,s) \doteq d_i.(t,s) \neq d_{i+1}.\cdots.(t,s) \neq d_n$. Then we have that $\mathcal{B}.\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{B}.\mathcal{A}'$ which concludes the case.
- Case $t \dot{\in} s.\mathcal{A}$: Hence, by the premises of this case, $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}$ where $\mathcal{I}((t,s)) \in \mathcal{I}(D)$. So by the induction hypothesis we can obtain $\mathcal{A}'$ such that $\mathcal{A}' \in tr_{D,E}(\mathcal{A})$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$. Because $\mathcal{I}((t,s)) \in \mathcal{I}(D)$ it must be the case that there exists some $d \in D$ such that $\mathcal{I}((t,s)) = \mathcal{I}(d)$. Thus we can conclude $(t,s) \doteq d.\mathcal{A}' \in tr_{D,E}(t \dot{\in} s.\mathcal{A})$ for such a $d \in D$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} (t,s) \doteq d.\mathcal{A}'$.

- Case $(\forall \bar{x}.\ t \dot{\notin} s).\mathcal{A}$: Hence $\mathcal{I}(\delta((t,s))) \notin \mathcal{I}(D)$ for all ground substitutions δ with domain $\bar{x}$. Hence $\mathcal{I}(\delta((t,s))) \neq \mathcal{I}(d)$ for all $d \in D$ and all δ with domain $\bar{x}$. Since $\bar{x} \cap fv(D) = \emptyset$ we have that $\delta(D) = D$. Hence $\mathcal{I}(\delta((t,s))) \neq \mathcal{I}(\delta(d))$ for all $d \in D$ and all δ with domain $\bar{x}$. Therefore $\mathcal{I} \models_{M,\emptyset,\emptyset} (\forall \bar{x}.\ (t,s) \not\doteq d_1).\cdots.(\forall \bar{x}.\ (t,s) \not\doteq d_n)$, where $D = \{d_1, \ldots, d_n\}$, by definition of the constraint semantics. Thus the case follows by the induction hypothesis.

- The cases $\mathsf{assert}(t).\mathcal{A}$, $\mathsf{event}(t).\mathcal{A}$, and $(\forall \bar{x}.\ \neg\mathsf{event}(t)).\mathcal{A}$ are proven similarly to the cases $\mathsf{insert}(t,s).\mathcal{A}$, $t \dot{\in} s.\mathcal{A}$, and $(\forall \bar{x}.\ t \dot{\notin} s).\mathcal{A}$ respectively.

The other implication is also proven by an induction on $\mathcal{A}$:

- Case 0: Trivially true.

- Cases $\mathsf{send}(t).\mathcal{A}$, $\mathsf{receive}(t).\mathcal{A}$, $t \doteq t'.\mathcal{A}$, and $(\forall \bar{x}.\ t \not\doteq t').\mathcal{A}$: Follows straightforwardly from the induction hypotheses.

- Case $\mathsf{insert}(t,s).\mathcal{A}$: Hence $\mathcal{A}' \in tr_{D \cup \{(t,s)\},E}(\mathcal{A})$. Since we already have that $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$ from the premises we can now apply the induction hypothesis to get that $\mathcal{I} \models_{M,\mathcal{I}(D \cup \{(t,s)\}),\mathcal{I}(E)} \mathcal{A}$. Since also $\mathcal{I}(D \cup \{(t,s)\}) = \mathcal{I}(D) \cup \{\mathcal{I}((t,s))\}$ it follows that $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathsf{insert}(t,s).\mathcal{A}$.

- Case $\mathsf{delete}(t,s).\mathcal{A}$: Hence $\mathcal{A}' = \mathcal{B}.\mathcal{A}''$ for some $d_1, \ldots, d_n$, i, n, $\mathcal{B}$, and $\mathcal{A}''$ where

  - $0 \leq i \leq n$,
  - $D = \{d_1, \ldots, d_i \ldots, d_n\}$,
  - $\mathcal{B} = (t,s) \doteq d_1.\cdots.(t,s) \doteq d_i.(t,s) \not\doteq d_{i+1}.\cdots.(t,s) \not\doteq d_n$, and
  - $\mathcal{A}'' \in tr_{D \setminus \{d_1,\ldots,d_i\},E}(\mathcal{A})$.

  We also have that $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}''$ and $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{B}$ because $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}'$. Note that $\mathcal{I}(D \setminus \{d_1, \ldots, d_i\}) = \mathcal{I}(D) \setminus \{\mathcal{I}((t,s))\}$ because $\mathcal{I}((s,t)) \notin \mathcal{I}(\{d_{i+1}, \ldots, d_n\})$ and $\{\mathcal{I}((t,s))\} = \mathcal{I}(\{d_1, \ldots, d_i\})$. Thus, we can apply the induction hypothesis and conclude that $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathsf{delete}(t,s).\mathcal{A}$.

- Case $t \dot{\in} s.\mathcal{A}$: Hence $\mathcal{A}' = (t,s) \doteq d.\mathcal{A}''$ for some $d \in D$ and $\mathcal{A}'' \in tr_{D,E}(\mathcal{A})$, and together with the premises we then have that $\mathcal{I}((t,s)) = \mathcal{I}(d)$ and $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}''$. We can now apply the induction hypothesis to get $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}$. Since $d \in D$ and $\mathcal{I}((t,s)) = \mathcal{I}(d)$ we also have that $\mathcal{I}((t,s))$ is in $\mathcal{I}(D)$. Thus we can conclude $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} t \dot{\in} s.\mathcal{A}$.

- Case $(\forall \bar{x}.\ t \dot{\notin} s).\mathcal{A}$: Hence $\mathcal{A}' = (\forall \bar{x}.\ (t,s) \not\doteq d_1).\cdots.(\forall \bar{x}.\ (t,s) \not\doteq d_n).\mathcal{A}''$ for some $\mathcal{A}'' \in tr_{D,E}(\mathcal{A})$ and $D = \{d_1, \ldots, d_n\}$. Hence, using the premises, we know that $\mathcal{I}(\delta((t,s))) \neq \mathcal{I}(\delta(d_i))$ for all $i \in \{1, \ldots, n\}$ and all ground δ with domain $\bar{x}$. Hence $\mathcal{I}(\delta((t,s))) \notin \mathcal{I}(D)$ for all ground δ with domain $\bar{x}$ because $\bar{x}$ and the free variables of D are disjoint. Since we also have $\mathcal{I} \models_{M,\emptyset,\emptyset} \mathcal{A}''$ from the premises we can apply the induction hypothesis to get $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} \mathcal{A}$, and thus we can conclude $\mathcal{I} \models_{M,\mathcal{I}(D),\mathcal{I}(E)} (\forall \bar{x}.\ t \dot{\notin} s)\mathcal{A}$.

- The cases $\mathsf{assert}(t).\mathcal{A}$, $\mathsf{event}(t).\mathcal{A}$, and $(\forall \bar{x}.\ \neg\mathsf{event}(t)).\mathcal{A}$ are proven similarly to the cases $\mathsf{insert}(t,s).\mathcal{A}$, $t \dot{\in} s.\mathcal{A}$, and $(\forall \bar{x}.\ t \dot{\notin} s).\mathcal{A}$ respectively.

□ □

A.4 The Typing Result

For our typing result we use the results of [19]. The authors have already extended their typing result to support equalities $t \doteq t'$ and inequalities $t \not\doteq t'$ in strands and constraints, and to support the Ana theories that we use in this paper. Their formalization is available at:

`http://www2.compute.dtu.dk/~samo/typing-soundness/` (*)

Thus we get from Theorem 4 of (*) the following result (note that their theorem is on the level of protocol transition systems, i.e., on constraints reachable in a symbolic protocol transition system $\Longrightarrow^{\bullet}$, but that this can easily be used to prove a result on constraints $\mathcal{A}$ since $(\{dual(\mathcal{A})\}; 0) \Longrightarrow^{\bullet *} (\emptyset; \mathcal{A})$):

Theorem 5 (Typing result on ordinary symbolic constraints)**.** *If $\mathcal{A}$ is well-formed and ordinary, $\mathcal{I} \models \mathcal{A}$, and $\mathcal{A}$ is type-flaw resistant, then there exists a well-typed interpretation $\mathcal{I}_\tau$ such that $\mathcal{I}_\tau \models \mathcal{A}$.*

The remaining section contains the proof of our typing results and related lemmas.

Lemma 4 (Type-flaw resistance preservation).

1. *If $\mathcal{A}$ is type-flaw resistant, well-formed, and $\mathcal{A}' \in tr(\mathcal{A})$, then $\mathcal{A}'$ is type-flaw resistant.*
2. *If $\mathcal{S}_0$ is a type-flaw resistant protocol and $(\mathcal{S}_0; 0) \overset{w}{\Longrightarrow}^{\bullet *} (\mathcal{S}; \mathcal{A})$ then both $\mathcal{S}$ and $\mathcal{A}$ are type-flaw resistant.*

Proof. 1. We first prove that $trms(\mathcal{A}') \cup setops(\mathcal{A}')$ is type-flaw resistant. Note that $trms(\mathcal{A}') \setminus trms(\mathcal{A}) \subseteq setops(\mathcal{A})$, and that $setops(\mathcal{A}') = \emptyset$. Hence $trms(\mathcal{A}') \cup setops(\mathcal{A}') \subseteq trms(\mathcal{A}) \cup setops(\mathcal{A})$. Since $trms(\mathcal{A}) \cup setops(\mathcal{A})$ is by assumption type-flaw resistant it follows that any subset is also type-flaw resistant. Thus $trms(\mathcal{A}') \cup setops(\mathcal{A}')$ is type-flaw resistant.

The reduced constraint $\mathcal{A}'$ does not contain set operations, and the remaining constraint steps of $\mathcal{A}'$ either originates from $\mathcal{A}$ or is constructed during the translation $tr(\mathcal{A})$. Thus it is sufficient in the remaining proof to only consider those steps which are created during the translation $tr(\mathcal{A})$, and we do so by a case analysis:

- During translation of a set operation $\mathsf{delete}(t,s)$, $t \dot{\in} s$, or $\forall \bar{x}.\ t \dot{\notin} s$ the translation adds new steps of the form $(t,s) \doteq (t',s')$ and $\forall \bar{x}.\ (t,s) \not\doteq (t',s')$ where $\mathsf{insert}(t',s')$ occurred somewhere in $\mathcal{A}$. We know by

type-flaw resistance of $\mathcal{A}$ that $\bar{x} \cup fv(t) \cup fv(t') \cup fv(s) \cup fv(s')$ all have atomic type. As we argued earlier, if (t, s) and (t', s') are unifiable then they have the same type. Hence all the new equality steps $(t, s) \doteq (t', s')$ are type-flaw resistant. Since all variables occurring in the inequality steps $\forall \bar{x}.\ (t, s) \not\doteq (t', s')$ are of atomic type we also have that these steps are type-flaw resistant.

- During translation of the event step $\mathsf{event}(e)$ the translation adds new a step of the form $e \doteq e'$ where $\mathsf{assert}(e')$ occurs in $\mathcal{A}$. We know by type-flaw resistance of $\mathcal{A}$ that $e, e' \in SMP(\mathcal{A}) \setminus \mathcal{V}$ and so they must have the same type if they are unifiable. Thus the new equality steps are type-flaw resistant.
- During translation of the event step $\forall \bar{x}.\ \neg\mathsf{event}(e)$ the translation adds a new step of the form $\forall \bar{x}.\ e \not\doteq e'$ where $\mathsf{assert}(e')$ occurs in $\mathcal{A}$. We know by type-flaw resistance of $\mathcal{A}$ that all variables $\bar{x} \cup fv(e) \cup fv(e')$ are of atomic type and so the new inequality steps are type-flaw resistant.

Thus $\mathcal{A}'$ is type-flaw resistant.

2. This follows by the fact that $\mathcal{A} = dual(\ell_1). \cdots . dual(\ell_k)$ for some $\ell_1, \ldots, \ell_k \in \mathcal{S}_0$, and since each ℓ_i are type-flaw resistant (so each $dual(\ell_i)$ is type-flaw resistant), $trms(\mathcal{S}_0) \cup setops(\mathcal{S}_0)$ is type-flaw resistant and $trms(\mathcal{A}) \cup setops(\mathcal{A}) \subseteq trms(\mathcal{S}_0) \cup setops(\mathcal{S}_0)$ (so $trms(\mathcal{A}) \cup setops(\mathcal{A})$ is type-flaw resistant), we can conclude that $\mathcal{A}$ is type-flaw resistant. □

□

Theorem 3 (Typing result on symbolic constraints). *If $\mathcal{A}$ is well-formed, $\mathcal{I} \models \mathcal{A}$, and $\mathcal{A}$ is type-flaw resistant, then there exists a well-typed interpretation $\mathcal{I}_\tau$ such that $\mathcal{I}_\tau \models \mathcal{A}$.*

Proof. From Theorem 2, Lemma 4(1), and the assumptions we can obtain a type-flaw resistant ordinary constraint $\mathcal{A}'$ such that $\mathcal{A}' \in tr(\mathcal{A})$ and $\mathcal{I} \models \mathcal{A}$. Hence, we can obtain a well-typed interpretation $\mathcal{I}_\tau$ such that $\mathcal{I}_\tau \models \mathcal{A}'$ by Theorem 5, and by then applying Theorem 2 again we can conclude the proof. □ □

Theorem 4 (Typing result for stateful protocols). *If $\mathcal{S}_0$ is a type-flaw resistant protocol, and $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \overset{w}{\Longrightarrow}^* (\mathcal{S}; M, D, E)$ where $w = (\sigma_1, \ell_1), \ldots, (\sigma_k, \ell_k)$ then there exists a state $(\mathcal{S}; M', D', E')$ such that $(\mathcal{S}_0; \emptyset, \emptyset, \emptyset) \overset{w'}{\Longrightarrow}^* (\mathcal{S}; M', D', E')$ where $w' = (\sigma'_1, \ell_1), \ldots, (\sigma'_k, \ell_k)$ for some well-typed ground substitutions $\sigma'_1, \ldots, \sigma'_k$.*

Proof. By first applying Theorem 1(1) and Lemma 4(2), then Theorem 3, and finally Theorem 1(2) we obtain the desired result. □ □

B A Mistake in a Related Work

Typing for stateful systems has also been considered in [24]. We show however that their result is incorrect with a counter-example. [24] allows a quite general specification of the intruder by a set of Horn clauses. There are restrictions of the form of these Horn clauses [24, Sec. 2.1]: each clause expresses either that the intruder can generate new terms by applying a function symbol to known terms (this corresponds to public function symbols in our work) or how the intruder can analyze terms, somewhat corresponding to our specification of Ana. For that, the requirement on the term obtained by the analysis is only that it must be a proper subterm of the term being analyzed. This allows for instance for the following Horn clause (where the predicate ik represents that a message is known by the intruder):

$$\mathsf{ik}(f(g(x)) \to \mathsf{ik}(x)$$

Suppose now f is a public function and the intruder knows $g(s)$ for a secret s. Then he can with the above rule apply f to $g(s)$ to obtain s. Such a step is however not covered by the constraint reduction procedure in [24], since analysis steps can only be applied to terms that the intruder directly knows, not ones he has to first compose. Now this leads to a counter-example for the typing result if we assume that f is not a public symbol, but there is an honest strand $\mathsf{receive}(x).\mathsf{send}(f(x))$ with variable x an atomic type, say, nonce. If the intruder knows $g(s)$ and s is a secret, then there is an ill-typed attack with $x = g(s)$, but no well-typed attack. Thus, the result of [24] does not hold for this generality of intruder models.

There is a second problem that there is no restriction on the type of universally quantified variables in [24]. Indeed composed-typed variables can also break the typing result as we have shown before. For instance, consider the inequality $\forall x.\ y \neq f(x)$ where $\Gamma(y) = f(\Gamma(x))$. For any instance $f(c)$ of y there is an instance of x (namely c) that does not satisfy the inequality. Hence the constraint has no well-typed solution. However, there does exist ill-typed solutions; since we are working in the free algebra terms are equal iff they are syntactically equal, and hence any instance of x that are not of the form $f(c)$ for some c would be a solution to the inequality. The following ASLan [24]

specification has exactly this issue:

Declarations:
$\mathsf{ik} : \mathsf{pred}\ (\mathsf{untyped})$
$X, n : \mathsf{nonce}$
$Y : f(\mathsf{nonce})$
$\mathsf{attack} : \mathsf{pred}\ ()$

Initial state:
$\mathsf{ik}(n)$

Transition rules:
$\mathsf{ik}(Y).\ \neg\exists X : Y = f(X) \Rightarrow \mathsf{attack}()$

Horn clauses:
$\forall X : \mathsf{ik}(X) \rightarrow \mathsf{ik}(f(X))$

Here the attack predicate cannot be derived if Y is instantiated with a well-typed instance in the transition rule. Thus it seems that a typing result for stateful protocols necessarily requires a carefully restricted setting like our set-based approach.

C Automatically Checking Type-Flaw Resistance

One crucial point of the typing result is that it is relatively easy to check, namely by statically looking at the format of messages rather than traversing the entire state space, and that this can also be done automatically as a static analysis of a user's specification before verification in a typed model.

Note $SMP(M)$ is in general infinite, but it is sufficient to check the following finite representation SMP_0 for type-flaw resistance: starting with $SMP_0 = M$, we first ensure that for every message $t \in SMP_0$ that contains a variable x of a composed type $f(\tau_1, \ldots, \tau_n)$, we ensure that also $t[x \mapsto f(x_1, \ldots, x_n)] \in SMP_0$ for some variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ that do not occur in t. (Even if some τ_i are themselves composed types, this can be done by adding finitely many messages, since all type expressions are finite terms.) Next, we close SMP_0 under subterms. Finally, let us ensure by well-typed α-renaming that all terms in SMP_0 have pairwise disjoint variables. Note that SMP_0 is a representation of $SMP(M)$ in the sense that every $SMP(M)$ term is a well-typed instance of an SMP_0 term. Now the condition that every pair $s, t \in SMP_0 \setminus \mathcal{V}$ with $\Gamma(s) \neq \Gamma(t)$ has no unifier, is equivalent to the type-flaw resistance of M. Proof sketch: note that $SMP_0 \subseteq SMP(M)$, giving us one direction of the equivalence. For the other direction, suppose there are any $s, t \in SMP(M)$ and σ such that $\Gamma(s) \neq \Gamma(t)$ and $\sigma(s) = \sigma(t)$. Since SMP_0 represents $SMP(M)$, there exists terms $s_0, t_0 \in SMP_0$ and well-typed substitutions θ_1 and θ_2 such that $s = \theta_1(s_0)$ and $t = \theta_2(t_0)$. Hence also $\Gamma(s_0) \neq \Gamma(t_0)$, and so s_0 and t_0 are not unifiable by assumption. Thus s and t cannot be unified as well because s_0 and t_0 do not share variables.

D Type-Flaw Resistance Examples

We want to discuss how our typing result is applicable in practice on several protocols, in particular that many protocols already satisfy the requirements of type-flaw resistance or require only minor changes to do so.

As for examples on stateless verification, we consider the examples from the AIF and AIF-ω tools, since this is the closest match to our formalization, as discussed in Section E.1. Note that some of the examples have similarly been considered in SAPIC, in particular PKCS#11, but in a way that violates the corresponding type flaw-resistance requirements, namely that the types of keys and values be atomic as discussed in Section E.2. To satisfy our requirements would not require a change of the protocol itself, but of the way its storage is modeled, e.g., repartitioning maps or use sets in the first place. Thus we discuss these examples rather on their AIF-ω models where the requirements are met or easily achieved.

D.1 Extension of the Keyserver Example

First, let us illustrate by a small example how type-flaw problems can arise in practice, how type-flaw resistance is violated in such a case, and how the situation can be fixed. Suppose for the key server example, we augment the protocol with an exchange where a user can prove to be alive, formalized by having for each user u and each session $j \in \mathbb{N}$ the following transaction strand:

$$\mathsf{receive}(N_j).PK_{u,j} \mathrel{\dot\in} \mathsf{ring}(u).\mathsf{send}(\mathsf{sign}(\mathsf{inv}(PK_{u,j}), N_j))$$

where all N_j and $PK_{u,j}$ have atomic types. The idea is that anybody can send the user a challenge N_j, and u answers with a signature on it. In this blunt form it is obviously a bad idea, since an intruder can send an arbitrary term instead of N_j. Indeed the protocol now violates type-flaw resistance: $\mathsf{sign}(\mathsf{inv}(PK_{u,j}), N_j)$ has a unifier with the normal update message $\mathsf{sign}(\mathsf{inv}(PK_i), \langle U_i, NPK_i\rangle)$, while they have different types. The general recommendation is thus to use some form of tag to indicate what the messages should mean. In fact, many protocol standards already describe a concrete message format, e.g., in this case that nonces and public keys have certain byte lengths, or even fields that indicate the length, if it is not fixed; in contrast many protocol models only abstractly the exchanged information as a tuples. It is thus recommended to model the concrete message formats by transparent functions, i.e., functions like pair that the intruder can compose and decompose, and check that the concrete formats of the protocol standard are disjoint so that a confusion is impossible. In this case we may have functions $\mathsf{update}(U, PK)$ that is used in the update message and functions $\mathsf{challenge}(N)$ and $\mathsf{response}(N)$ to model the challenge response protocol to have rather the following form:

$$\begin{aligned}&\mathsf{receive}(\mathsf{challenge}(N_j)).\\ &PK_{u,j} \mathrel{\dot\in} \mathsf{ring}(u).\mathsf{send}(\mathsf{sign}(\mathsf{inv}(PK_{u,j}), \mathsf{response}(N_j)))\end{aligned}$$

One may argue that the formatting of challenge message is irrelevant since it is in cleartext, we suggest, however, to use formatting information also here, since it is in fact good practice for implementations anyway and does not really hurt.

With the change we now have again type-flaw resistance and our typing result is applicable.

D.2 Secure Vehicle Communication

A set of examples for AIF is a model of the secure vehicle communication of the SEVECOM project [30, 26]. These define a setup for hardware security modules in cars that store a number of keys that can only be used via a number of API commands. A main concern is the so-called root key update. Here we have the following message patterns of incoming and outgoing messages, where the variables K, $K1$, and $K2$ are of type value: K, $\mathsf{sign}(\mathsf{inv}(K), K)$, and $\mathsf{sign}(\mathsf{inv}(K2), \mathsf{pair}(K1, K))$, where we have omitted some message patterns that can be obtained by well-typed substitutions. The corresponding set of sub-message patterns

$$SMP(\{K, \mathsf{sign}(\mathsf{inv}(K), K), \mathsf{sign}(\mathsf{inv}(K2), \mathsf{pair}(K1, K))\})$$

is the closure of the message patterns under subterms, term decomposition, and well-typed instantiation.

This is not directly type-flaw resistant: if we first consider a well-typed renaming of the first signature, say, $\mathsf{sign}(\mathsf{inv}(K3), K3)$ then there is a unifier with the other signature $\mathsf{sign}(\mathsf{inv}(K2), \mathsf{pair}(K1, K))$, namely identifying $K3$, $K2$, and $\mathsf{pair}(K1, K)$. Indeed the two signature messages here have different type and meaning (the first means key revocation, the second means key update), while they have nothing signaling in the signed text which message it is. Indeed, if we look at the standard [30], it requires that the revocation and the update signature contain specific text namely "REVOKE ROOT PUBLIC KEY" and "LOAD ROOT PUBLIC KEY". This had not been modeled in [26]. Again, we recommend to use here transparent functions $\mathsf{revoke}/1$ and $\mathsf{update}/2$, to model the format of the revoke and update messages, respectively, and for which the intruder can directly extract the arguments, i.e., $\mathsf{Ana}(\mathsf{revoke}(K)) = (\emptyset, \{K\})$ and $\mathsf{Ana}(\mathsf{update}(K1, K2)) = (\emptyset, \{K1, K2\})$. Then we have as SMP the following set closed under closure under subterms and well-typed substitutions (in this case study closing under term decomposition is unnecessary as it is subsumed by closing under subterms):

$$\{K, \mathsf{sign}(\mathsf{inv}(K), \mathsf{revoke}(K)), \mathsf{sign}(\mathsf{inv}(K2), \mathsf{update}(K1, K))\}$$

and indeed now type-flaw resistance is satisfied.

D.3 PKCS#11

The examples of AIF-ω contains a number of specifications of PKCS#11 based APIs following [16, 10]. Again the model of a crypto-device is here by a number

of transactions that consist of a command and arguments to the device, which performs some checks, possibly generates some encryptions and makes some notes and sends an output as a result. The question is if an intruder can obtain something by combining several API calls in a way that had not been anticipated. Again, the AIF-ω model of these calls is based on sets for describing the different flags associated to a key (e.g., whether it is key that can be extracted). The specification is again that all elements of the sets are declared to have type value, thus it only remains to check that the messages input and output to the device fulfill the type-flaw resistance. Since the commands themselves are not encrypted, the AIF models do not model opcodes and the like and just present the bare arguments (e.g. key-handles, encrypted messages etc.) to the device. We then obtain the following kind of messages:

$$\mathsf{bind}(N, K, K), \quad \mathsf{h}(N, K), \quad K, \quad \mathsf{senc}(M, K)$$

Here K, N, and M are of type value, and we have omitted some terms that are redundant under well-typed substitution again. Also this is type-flaw resistant, however there is an interesting point. As long as only the intruder is interacting with the token interface, the type-flaw resistance is guaranteeing that he has no gain from using ill-typed messages. However, when we consider extensions of these examples (e.g., a richer API or a network with other tokens or honest parties), then also more complex messages M in the symmetric encryption may be produced (or received by honest agents) and the type-flaw resistance breaks. It therefore seems like a good idea to not have raw encryptions of a key (like in $\mathsf{senc}(M, K)$) but to insert some more information into the encrypted message, like a format as in the SEVECOM example above. Indeed this solves some of the attacks that arise in the use of the API already, when we use different such formats (or tags) for keys of different intended use (e.g. wrap-unwrap attacks).

D.4 ASW

The fair contract signing protocol ASW is another example of a protocol that necessarily requires a global state. With AIF shipped a formalization of ASW that abbreviates some protocol messages drastically, for instance the function $msg1(A, B, contract(A, B), h(NA))$ to abbreviate a message from A to B that is actually signed with the private key of A, and intruder rules that allow the intruder to compose such a $msg1$-message if A is dishonest (and always decompose it). To use it with our typing result and the Ana functions, we need to use a more standard model, explicitly denoting the signature function, i.e., for $msg1$ we rather have $\mathsf{sign}(\mathsf{inv}(pk(A)), m1(A, B, contract(A, B), h(NA)))$ where $m1$ is a transparent function to model the concrete format of the message content.

Note that when this message is received by B, it has the form

$$\mathsf{sign}(\mathsf{inv}(pk(A)), m1(A, B, contract(A, B), HNA))$$

with a variable HNA of the composed type $h(nonce)$, since B cannot check at this point that this is indeed a hash of a nonce as it should be. The entire point of this fair exchange is in fact that the nonces are revealed only later.

The second message has the form $\mathsf{sign}(\mathsf{inv}(pk(B)), m2(B, t, h(NB)))$ where $t = \mathsf{sign}(\mathsf{inv}(pk(A)), m1(A, B, contract(A, B), HNA)$, i.e., the t is a message of the first form; note that here the variables A and B must all agree in these forms since this is part of what the participants check. When A receives this message, it has the form $\mathsf{sign}(\mathsf{inv}(pk(B)), m2(B, t', HNB))$ with a composed-type variable HNB since A similarly cannot check that this is really a hash of a nonce. In contrast t' has the form $\mathsf{sign}(\mathsf{inv}(pk(A)), m1(A, B, contract(A, B), h(NA)))$ since here the nonce NA has been created by A herself earlier.

Messages three and four of the protocol are simply the nonces NA and NB; even though we suggest not to have such raw data sent around (and rather wrap it in another transparent format), this is not a problem with type-flaw resistance.

Note that part of the verification we have now the equations $HNA = h(NA)$ and $HNB = h(NB)$ since after receiving the nonce from each other, the agents should check out with the respective HNA and HNB received earlier. Note also that if there was a continuation for the case that such a check fails, it could not be handled by our typing result, because that would imply composed-typed variables in inequalities.

The most interesting part of ASW is the communication with a server in case the above four-step contract signing goes wrong, i.e., if one of the agents does not receive an answer anymore, in particular if B has received message three from A and thus has a valid contract, and dishonestly refuses to reveal the final message four to A, so A does not have a contract. The protocol assumes that both agents have resilient channels to a trusted third party, i.e., they eventually get an answer. If A did not receive an answer to her message one, she can send an abort message to the server of the form $\mathsf{sign}(inv(pk(A)), abortReq(t))$ where t is the first message she had sent. If A or B at a later point in the protocol (i.e., after at least sending/receiving message two) do not obtain an answer, they can ask for a resolve, which is of the form $\mathsf{sign}(\mathsf{inv}(pk(X)), resolveReg(t1, t2))$ where $t1$ and $t2$ are the first two messages of the protocol and X is the agent A or B asking for the resolve. The server should now look in his database of contracts, and if the contract does not occur in the database yet, grant the abort or resolve request, by the messages $\mathsf{sign}(\mathsf{inv}(pk(s)), abort(t))$ or $\mathsf{sign}(\mathsf{inv}(pk(s)), resolve(t1, t2))$, respectively, where $\mathsf{inv}(pk(s))$ is the private key of the server. The result is of course also stored in the database, and this entry will be the reply to any agent who asks for an abort or resolve of that contract.

The AIF model has here several limitations: since resilient channels cannot be modeled directly, it models the interaction between users and servers as atomic transitions. The assumption of the real protocol is a bit weaker: an intruder cannot entirely block a request or the response, but he may be able to delay it, for instance observe a request and send a different request that arrives earlier at the server. Also the messages exchanged are not modeled, but only the effects on the users and servers database. We have thus here checked type-flaw resistance both for the restricted model that comes with AIF and for an extended model that includes all necessary steps and possible interleavings.

The database of the server is actually modeled as a family of sets $scondb(A, B, Status)$

for each agent A, B and $Status$ is either valid or aborted. However, instead of the contract, it stores only the nonce NA. This is due to AIF's limitations to sets of constants. It is sufficient to make a working model of ASW, since NA is sufficient to identify the concrete exchange.

In fact, satisfaction of the type-flaw resistance is easy to see, since every function symbol except sign is applied in all messages to terms of the same types and the message being signed is never directly a variable. Similar, for the sets, the contents have all type nonce, and the set terms have the form $family(A, B, Status)$ where A and B are agents and $Status$ ranges over a set of possible status messages.

E Connections to Other Formalisms

We have introduced the formalism of transaction strands to have a simple and mathematically pure formalism as a protocol model for our result without the disturbance of the many technical details of various protocol models. We want to illustrate now that our result can nonetheless be used in various protocol models, but we only sketch the main ideas and discuss also limitations of our typing results.

Note that the core of our result is proved on symbolic constraints (intruder strands) of a symbolic transition system. Connecting another formalism with our typing result requires only two aspects. First, one needs to define the semantics for the formalism in terms of a symbolic transition system with constraints (including set operations, equalities, and inequalities). Second, one needs to transfer the notion of type-flaw resistance, so that a type-flaw resistant specification in the formalism will only produce type-flaw resistant constraints. We have done this for transaction strands with detailed proofs. Due to the variety of other formalisms and their technical details, we only sketch in the following the ideas for the most common constructions.

E.1 AIF-ω and Rewriting

Our transaction strands are in some sense a purified version of AIF-ω. In a nutshell, it describes protocols by a set of rewrite rules for a state transition system, where each state is a set of *facts* like $\mathsf{ik}(m)$ to denote that the intruder knows message m. It is thus also similar to other rewriting based languages like Maude-NPA or the AVANTSSAR ASLan.

One can translate each AIF-ω rule into transaction strands as follows. Every intruder knowledge fact $\mathsf{ik}(m)$ on the left hand side of a rule corresponds to receiving a message m, and on the right hand side to sending a message m. If the expression t in s occurs on the left-hand side, then the transaction strand must contain $t \dot{\in} s$; if the same expression does not occur on the right-hand side, then the transaction must include $\mathsf{delete}(t, s)$. If the expression t notin s occurs on the left-hand side, then the transaction must contain $\forall \bar{x}.\, t \dot{\notin} s$ where x are the variables that on the left-hand side only occur in notin expressions. Finally, if

t in s occurs on the right-hand side but not on the left, then the transaction must include $\mathsf{insert}(s, t)$. All other facts of AIF-ω are persistent (i.e., once true, they remain true in all successor states), therefore we can model them as events in transaction strands, using $\mathsf{event}(f)$ for the left-hand side facts and $\mathsf{assert}(f)$ for right-hand side facts. Note that the order of all these actions in the transaction matters: first we should have all receiving messages, checking for events and set memberships, then modifying sets and sending the outgoing messages. Still one may wonder what happens in the following AIF-ω rule: x in $s.y$ in $s \Rightarrow x$ in s. If $x = y$ then this rule is contradictory, and the semantics of AIF-ω excludes such substitutions. For that reason, we also have to include the inequality $x \neq y$ to the transaction to exactly follow the AIF-ω semantics. In all remaining cases, then the inner order of the actions is actually irrelevant then, but these subtle points where one of the motivations to introduce transaction strands. Finally, note that the rules from AIF-ω may have variables that represent any value from a countable set of constants, as well as the creation of fresh values. Since transaction strands do not have a mechanism for creating fresh values and free variables are not allowed, one must instantiate these variables appropriately, producing a countable set of transaction strands from finitely many rules.

With this translation from AIF-ω rules to transaction strands, we also directly obtain a semantics using symbolic constraints and actually immediately transfer the notion of type-flaw resistance from transaction strands with the obvious adaptations. However, type-flaw resistance will not be directly satisfied for typical AIF-ω specifications immediately, because they would contain rules for the intruder that contain untyped variables. While for honest agents, it is not a restriction to declare the *intended* type for each variable, the intruder deduction rules should be applicable to messages of *any* type. Thus, we have to make the reservation that the intruder deduction of an AIF-ω specification must be within the bounds of the intruder model we have used here, namely composition with public functions and decomposition according to an Ana theory. This is indeed possible for all the standard operators like symmetric and asymmetric encryption, signatures, hashes, and transparent functions like pair; operators that require algebraic equations like xor are however not supported. We come back to this when discussing process calculi and reduction rules below.

Finally, note that other rewriting based formalisms like Maude-NPA (or the closely related linear logic rules) are not based on sets, but usually multi-sets of facts, and they are not persistent, i.e., facts can be removed by transitions, which cannot directly be modeled by our notion of events in transaction strands. There is however a way to encode this using sets: for each fact where we want to encode non-persistent behavior, we introduce a corresponding event with one more argument. For this argument we use a fresh constant whenever a fact is introduced by a transition and the argument becomes member of a special set active. Whenever the fact shall be removed, we simply remove the corresponding constant from the set active. This allows model both the multi-set aspect as well as the non-persistent aspect.

E.2 Set-π and Process Calculi

Process calculi are a very popular way of specifying protocols. While they can immediately describe stateful systems (due to Turing completeness), this is usually not at a level that directly works with existing verification methods so well. Therefore several extensions have been proposed, namely Set-π for set operations similar to AIF-ω, and Sapic for adding a notion of maps. One gap to the rewriting formalisms above is that process calculi do not have the notion of an atomic transaction. Therefore both AIF-ω and Sapic rely on the use of locks, i.e., in order to read and write on a set or (an element of) a map, one has to first lock it, and no other process can get a lock on the same item before it is unlocked. It is possible to give a translation to transaction strands, modeling explicitly the locks by an additional set that stores which of the other sets are locked. However, it is a bit more convenient to directly give a semantics as a symbolic transition system, i.e., producing symbolic constraints in each execution.

However, before we can do that, there is another obstacle to overcome: it is convenient to model in process calculi decryption and checking of messages explicitly by a let construct and reduction rules. For instance if the public function crypt represents asymmetric encryption and inv the private function that maps from public to private keys, for decryption one would introduce a new operator dcrypt and have the reduction rule $\mathsf{dcrypt}(\mathsf{crypt}(x,y),\mathsf{inv}(x)) \to y$. Then receiving and decrypting a message for instance would be $\mathsf{in}(u).\mathsf{let}\ v = \mathsf{dcrypt}(u,\mathsf{inv}(k))\ \mathsf{in}\ P\ \mathsf{else}\ Q$ Thus process P is executed if the received message u is indeed encrypted with k (and binding v to the content of that message), otherwise Q is executed. Note that the destructor dcrypt does not occur as part of "normal" messages.

Our typing result can only support such destructors, if we can express such decryption operations using an Ana theory, in the example we would have $\mathsf{Ana}(\mathsf{crypt}(x,y)) = (\{\mathsf{inv}(x)\},\{y\})$ and we would translate the above example process into $\mathsf{in}(u).\ \mathsf{if}\ (\mathsf{crypt}(k,?v) \doteq u)\ \mathsf{then}\ P\ \mathsf{else}\ Q$. Note that here we have actually made an extension of Set-π, namely adding the concept of equalities from transaction strands to the if construct, including that newly introduced variables on the left-hand side are binding, here v, and we mark this by a question mark as is standard. This is formally defined by the symbolic semantics below.

Besides destructors, process calculi also commonly use reduction rules for checks on messages, e.g., $\mathsf{verify}(\mathsf{sign}(\mathsf{inv}(x),y),x) \to \mathsf{true}$ that can be used to verify a signature, for instance: $\mathsf{in}(u).\ \mathsf{let}\ true = \mathsf{verify}(u,k)\ \mathsf{in}\ P\ \mathsf{else}\ Q$. For this, we do not need to have a corresponding line in Ana, rather we can model this directly by an equality: $\mathsf{in}(u).\ \mathsf{if}\ \mathsf{sign}(\mathsf{inv}(k),?z) \doteq u\ \mathsf{then}\ P\ \mathsf{else}\ Q$. With this, all the standard operators can be supported, except those that require algebraic equations like xor.

If we now assume Set-π without let but instead with equations in if, we can define its semantics as a symbolic transition system as follows (using notation and labels similar to the original ground semantics):

$$
\begin{array}{ll}
\text{NIL}: & \mathcal{P} \uplus \{(0, \emptyset)\}, \mathcal{A} \to \mathcal{P}, \mathcal{A} \\
\text{COM}_1: & \mathcal{P} \uplus \{(\mathsf{in}(x).P_1, L_1)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P_1, L_1)\}, \mathcal{A}.\mathsf{send}(x) \\
\text{COM}_2: & \mathcal{P} \uplus \{(\mathsf{out}(N).P_2, L_2)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P_2, L_2)\}, \mathcal{A}.\mathsf{receive}(N) \\
\text{PAR}: & \mathcal{P} \uplus \{(P_1 \mid P_2, \emptyset)\}, \mathcal{A} \to \mathcal{P} \uplus \{(P_1, \emptyset), (P_2, \emptyset)\}, \mathcal{A} \\
\text{REPL}: & \mathcal{P} \uplus \{(!P, \emptyset)\}, \mathcal{A} \to \mathcal{P} \uplus \{(\alpha(P) \mid !P, \emptyset)\} \\
\text{NEW}: & \mathcal{P} \uplus \{(\mathsf{new}\ x.P, L)\}, \mathcal{A} \to \mathcal{P} \uplus \{(P[x \mapsto c], L)\}, \mathcal{A}\} \\
& \text{for some fresh name } c \\
\text{IF1}: & \mathcal{P} \uplus \{(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, L)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P_1, L)\}, \mathcal{A}.tr(b) \\
\text{IF2}: & \mathcal{P} \uplus \{(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, L)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P_2, L)\}, \mathcal{A}.tr(\neg b) \\
\text{SET}_+: & \mathcal{P} \uplus \{(\mathsf{insert}(t, s).P, L)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P, L)\}, \mathcal{A}.\mathsf{insert}(t, s) \\
\text{SET}_-: & \mathcal{P} \uplus \{(\mathsf{delete}(t, s).P, L)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P, L)\}, \mathcal{A}.\mathsf{delete}(t, s) \\
\text{LCK}: & \mathcal{P} \uplus \{(\mathsf{lock}(l).P, L)\}, \mathcal{A} \to \\
& \mathcal{P} \uplus \{(P, \{l\} \cup L)\}, \mathcal{A}.l \neq l_1. \cdots .l \neq l_n \\
& \text{where } \{l_1, \ldots, l_n\} = L \cup \bigcup_{(P', L') \in \mathcal{P}} L' \\
\text{ULCK}: & \mathcal{P} \uplus \{(\mathsf{unlock}(l).P, \{l\} \uplus L)\}, \mathcal{A} \to \mathcal{P} \uplus \{(P, L)\}, \mathcal{A}
\end{array}
$$

where $\alpha(P)$ is a fresh renaming of all variables in P that are bound by an in statement, and where the translation $tr(b)$ of a condition b is defined as follows. Recall that we had used the notion of binding occurrences also in equations (and logically this can also be done in set membership checks) and marked the respective occurrence by a question mark, like $?x = ...$ Let in the following $\bar{x}$ be the set of variables of a condition that are marked with the question mark:

$$
\begin{array}{lcl}
tr(s \doteq t) & = & s \doteq t \\
tr(s \neq t) & = & \forall \bar{x}.s \neq t \\
& & \text{where } \bar{x} \text{ are the variables in } s \text{ marked with } ? \\
tr(t \mathrel{\dot\in} s) & = & t \mathrel{\dot\in} s \\
tr(t \mathrel{\dot\notin} s) & = & \forall \bar{x}.t \mathrel{\dot\notin} s
\end{array}
$$

Note that the locking is not checked upon set operations, as this is done statically in set-π. Since in general sets can have terms with variables , we have formulate the check as inequalities in the LCK rule.

In order to check type-flaw resistance, one now needs to consider the translation from let-statements into equations (which can be done transparent to the user) and then the type-flaw resistance property is almost as before, only we need to consider each condition positive and its negation (unless the else case is empty). Note that sometimes this may lead to violations of type-flaw resistance when we have variables of composed types, since they are not allowed in inequalities. This is only a problem if two issues arise at the same time though: (1) the else branch is not empty and (2) the structure of the message is not entirely

discernible to that agent (e.g. if the result of a decryption is an encryption that the agent does not have the key to). One of the two issues alone can be handled, however.

E.3 SAPIC

Finally, let us consider the Sapic tool that is also a process calculus, but instead of sets has a global map, i.e., one can insert key-value pairs into the map (where inserting multiple times with the same key is overwriting), delete pairs, and query what value is associated to a key.

For a restricted setting, we can indeed express this map with sets, namely if we can split the map into finitely many partitions where each key and value are of some atomic type. For instance, in the PKCS examples, the value type are actually tuples, but the second part ranges over finitely many values and thus one could represent this maps as a finite collection of maps with atomic value type.

The idea is of course to model map $m = [k_1 \mapsto v_1, \ldots, k_n \mapsto v_n]$ by a family of sets $m(\cdot)$ such that $v_1 \in m(k_1), \ldots, v_n \in m(k_n)$. Initially, all maps should contain one distinguished symbol $\perp$ to represent that for that key no value is in the map. Then to insert the tuple (k, v) translates to the set operations $x \dot{\in} m(k).\mathsf{delete}(x, m(k)).\mathsf{insert}(v, m(k))$. To delete key k from the map is then like inserting $(k, \perp)$. Querying for key k is checking $x \dot{\in} m(k)$ and $x \not\equiv \perp$.