I. INTRODUCTION

The automated verification of security protocols has been the subject of intensive study for about two decades now. This has resulted in methods and tools that are feasible for finding attacks or proving the absence of attacks for a large class of protocols. One of the most successful approaches is static analysis, as for instance used in the ProVerif tool [6], [8], [23], [13], [10]. The key idea of this approach is to avoid the exploration of the state space of a transition system, but rather compute an over-approximation of the set of messages that the intruder can ever learn. The abstraction is efficient because it avoids the common state-explosion of model checking and it does not require a limitation to finite state-spaces. While this works fine for many protocols, we get trivial “attacks” if a protocol relies on a notion of state that is not local to a single session. A simple example is the protocol:

\[ A \rightarrow B : \{ \text{Msg, Counter} \}_\text{Key} \]

where Key is a symmetric key known only to A and B, Msg is some payload message and Counter is the current value of a counter used for avoiding replay attacks: B accepts a message only if Counter is strictly greater than in the last accepted message from A. This protocol thus ensures injective agreement [19] on Msg, since B can be sure that A has sent Msg and it is not a replay, i.e., even if A chooses to transmit several times the same Payload Msg, B will not accept it more often than A sent it. There are of course several ways to model such a counter in the applied \( \pi \) calculus, the input language of ProVerif, however none is going to work in the abstraction due to its monotonicity: roughly speaking, whatever B accepts once, he will accept any number of times and we thus get trivial attacks. In fact, verifying injective agreement properties in ProVerif requires a dedicated mechanism [9].

The above message is taken from the CANAuth protocol [17] that is intended for the automotive industry and needs to work under strong limitations on bandwidth and time. Due to these constraints, standard mechanisms like challenge-response (B first sends a nonce, then A includes it in the message instead of the counter) are no option. But even without such bounds there are practical real-world examples that today’s abstraction approaches cannot support:

- Key update/revocation: after updating an old key with a fresh one, one does not accept messages encrypted with the old key anymore (at least after some grace period).
- Key tokens/hardware security modules: they maintain a set of keys of different status and attribute, and can be communicated with through an API. When changing the status of a key, an operation may no longer be possible with that key.
- Data bases: an online shop that maintains a database of orders along with their current status; a customer may cancel an order, but only as long as it has not entered the status “shipped”.

Abstract—Communication protocols often rely on stateful mechanisms to ensure certain security properties. For example, counters and timestamps can be used to ensure authentication, or the security of communication can depend on whether a particular key is registered to a server or it has been revoked. ProVerif, like other state of the art tools for protocol analysis, achieves good performance by converting a formal protocol specification into a set of Horn clauses, that represent a monotonically growing set of facts that a Dolev-Yao attacker can derive from the system. Since this set of facts is not state-dependent, the category of protocols of our interest cannot be precisely analysed by such tools, as they would report false attacks due to the over-approximation.

In this paper we present Set-\( \pi \), an extension of the Applied \( \pi \)-calculus that includes primitives for handling databases of objects, and propose a translation from Set-\( \pi \) into Horn clauses that employs the set-membership abstraction to capture the non-monotonicity of the state. Furthermore, we give a characterisation of authentication properties in terms of the set properties in the language, and prove the correctness of our approach. Finally we showcase our method with three examples, a simple authentication protocol based on counters, a key registration protocol, and a model of the Yubikey security device.
More generally, systems that have a notion of state (that is not local to a session) and that have a non-monotonic behavior—i.e. an action is possible until a certain change of state and that is disabled afterwards are incompatible with the abstraction of tools like ProVerif.

Contributions. In this paper we formally define the novel Set-$\pi$ calculus that extends the popular applied $\pi$ calculus by a notion of sets of messages. It allows us to declaratively specify how processes can store, lookup and manipulate information like sets of keys, orders, or simply counters as in the above example. (Note that this does not increase the expressive power of applied $\pi$, since one could also simulate sets using private channels.) The semantics gives rise to an infinite-state transition system since we can model unbounded processes that generate any number of fresh messages. We can define state-based queries for Set-$\pi$, that ask for attacker-derivable messages, their set membership status, and boolean combinations thereof. A specification is secure iff no query is derivable.

The second contribution is a stateful abstraction for Set-$\pi$. The idea is that the abstraction of a message incorporates the information to which sets it belongs, and we model how this set membership can change. In doing so, we integrate the essential part of the state information into an otherwise stateless abstraction. This fine balance allows us to combine the benefits of stateless abstraction—namely avoiding state explosion and bounds to finite state spaces—and at the same time support a large class of protocols that rely on some state aspects.

Formally, this abstraction is a translation from a Set-$\pi$ protocol specification and a set of queries into a set of first-order Horn clauses. Our third contribution is to prove a soundness result for this abstraction: every reachable state is abstractly represented by the Horn clauses. In particular, if the Horn clauses have no model, then the given Set-$\pi$ specification is secure for the given queries and against an arbitrary Set-$\pi$-attacker. For checking whether the Horn clauses have a model, we can use various automated tools like ProVerif.

Finally, we demonstrate the practical feasibility of our approach by two major case studies on the MaCAN and CANAuth which are candidates for the next generation automotive systems; we use in this paper only excerpts as illustrating examples, the full analysis is found in [12].

a) Plan of the Paper: The rest of the paper proceeds as follows: Section II introduces the language and presents, as a running example, a simplified version of CANAuth—a protocol for securing in-vehicle communication—a protocol that can be more precisely analysed with Set-$\pi$. Section III describes the type system. Section IV gives an instrumented semantics for the language. Section V presents two definitions for weak and strong authentication and provides a mechanised way to encode such properties in Set-$\pi$. Section VI presents our translation of Set-$\pi$ into Horn clauses and Section VII proves the correctness of this approach. In Section VIII we present the key registration and the Yubikey examples along with experimental results, and finally in Section IX we discuss related work.

II. SET-$\Pi$

The calculus is presented in Figure 1. The syntactic categories are terms $M,N$, processes $P,Q$, set-membership expressions $b$, set-membership transitions $b^+$, and system declarations $Sys$. Types in the language, denoted by $T$ and $T^{Sym}$, are introduced in the next section.

We mark with $\mathbb{P}$ the set of processes produced by the syntactic category $P$ and with $\mathbb{M}$ the set of terms produced by $M$. To avoid ambiguity, we mark with $\mathbb{S} = \{s_1, \ldots, s_n\}$ the sets declared in a specific Set-$\pi$ model, while we use $s, s', s_1, s_2$ and so on to denote any of the sets in $\mathbb{S}$, and $L$ denotes a collection of locked sets.

Terms are either variables, names or constructor applications. Names are annotated with a sequence of terms that record which copy of the process created them, by the use of session identifiers, and input terms to keep track of data dependencies. Constructors are generally accompanied by destructors defined as rewrite rules that describe cryptographic primitives. For example:

\begin{align*}
\text{reduce } & \forall x_m : t_m, x_k : \text{key} \cdot \text{dec(enc}(x_m, x_k), x_k) \rightarrow x_m; \\
\text{models symmetric key encryption: for every message } x_m, \text{ key } x_k, \text{ if a process knows an encrypted message } & \text{enc}(x_m, x_k) \text{ and the key } x_k \text{ then it can obtain the message } x_m. \text{ For any rewrite rule of the form reduce } & \forall \vec{x} : T^\pi \cdot g(M_1, \ldots, M_n) \rightarrow M; \text{ we require that } f_v(M) \subseteq f_v(M_1, \ldots, M_n) \subseteq \{\vec{x}\}. \\
\text{Processes } & P,Q \text{ are: the stuck process } 0, \text{ replication } \rightarrow \text{ which is marked by a label } k, \text{ parallel composition of two processes, output, typed input, restriction } \rightarrow \text{ marked by a label } l \text{ and destructor application. We require that processes are closed and that they are properly alpha-renamed. Note that the user does not specify annotated names and labels in the initial process, hence the grey color. Names are introduced by the semantic step for restriction, and unique labels are automatically inserted by the parser. }
\end{align*}
The distinguishing feature of our calculus is the ability to track values in databases: the membership test (if b then P else Q) allows us to check a membership condition b, while a set transition (update(b*); P) inserts and removes terms from sets, according to b*. Finally we use looks on sets to ensure linearity of set transitions: the construct lock(L) prevents all other processes to modify sets in L in the continuation, while unlock(L) releases the locks on L.

A system Sys is the context in which the process operates. A new set s containing elements of type T is declared with new s : set T, and the reduc construct specifies a rewrite rule.

As syntactic sugar we add the following features to Set-π:

- n-tuples \( \langle M_1, \ldots, M_n \rangle \), which can be encoded with a constructor \( \text{mktp}_{\text{n}}(M_1, \ldots, M_n) \) and n destructors \( \forall \vec{x} : T^n \cdot \text{proj}_i^n(\text{mktp}_{\text{n}}(x_1, \ldots, x_n)) \rightarrow x_i \);
- pattern matching on tuples for let bindings and inputs, which can be encoded using multiple let bindings with the destructors \( \text{proj}_i^n \) and equality tests;
- \( \forall \vec{s}_1, \ldots, \vec{s}_n \cdot P \) means the replication of P that locks sets \( s_1, \ldots, s_n \) before its execution; the semantics releases the locks when P reduces to 0;
- let variable assignments when the right hand side is not a defined destructor, denoting by \( M \not\in s \) the expression \( \forall (M \in s) \), and omitted else branches where not needed.

a) CANAuth example: As a running example we use CANAuth, a protocol that runs on top of resource limited CAN bus networks and—due to the real-time requirements of CAN bus networks—uses one way communication from source to destination, avoiding challenge-response patterns. The low level mechanism that is used to ensure freshness properties is the use of counters together with message authentication codes. Comparing a counter with the highest value previously received allows to ensure that a message cannot be replayed.

Here we model a simplified form of its message authentication procedure, that assumes that the two communicating parties, a sender Alice and a receiver Bob, have established a session key \( k \) and are both keeping track of their own local copy of a counter \( c \).

In order for Alice to send a message \( m \) to Bob, she signs \( m \) and her own counter \( c \) increased by one with the shared key \( k \). Here we denote with \( \text{hmac}(msg(c), k) \) such a signature. Bob receives the message and checks whether the counter \( c \) is already in the set \( \text{received} \); if not, it accepts the message.

\[
A \triangleq \text{new}^1 c : \text{cnt};
\]

let \( m = \text{msg}(c) \) in

\begin{align*}
\text{event} \ & \text{send}(m); \\
\text{out}(\text{ch}, (m, \text{hmac}(m, k))); \ & 0 \\
\end{align*}

\[
B \triangleq \text{in}(\text{ch}, (x_m, x_s) : \langle \text{msg}(\text{cnt}), \text{hmac}(\text{msg}(\text{cnt}), \text{key}) \rangle); \\
\text{let} \ & x_c = \text{getcnt}(x_m) \ 	ext{in} \\
\text{let} \ & x_c = \text{checksign}(x_m, x_s, k) \ 	ext{in} \\
\text{if} \ & x_c \not\in \text{received} \ 	ext{then} \\
& \text{update}(x_c \in \text{received}); \\
& \text{event} \ \text{accept}(x_m); \ & 0
\]
simply try to type the continuation under the same \( \Gamma \). The rules for input and restriction add to \( \Gamma \) the type of the new bound variables. The rules for if and update check that the membership test or the set transitions are well-formed (i.e. \( M \) is of type \( T \) when \( s \) is of type set \( T \)) and that the interested sets are locked. Finally the rule for \( \text{restriction NEW} \) maps the type of the new \( l \) to name instances \((x,y,...)\). The purpose of \( \text{restriction NEW} \) is to substitute any type \( \Gamma \) with a term that has only variables of atomic types and such that the term \( \Gamma \) constructed from type \( T \) using the function \( \text{pt}^l_\rho \). The purpose of \( \text{pt}^l_\rho \) is to substitute any type \( T \) with a term \( N' \) that is homomorphic to \( T \): that is, for every occurrence of a name \( a \) in the type \( T \) it produces a variable \( x_a,V \) that is syntactically different from all other variable occurrences, and every occurrence of a constructor type produces a constructor term of the same form.

The rule \( \text{PAR} \) splits the process into two parallel processes. 

In the semantic rules we assume \( \Gamma \) to contain the type definitions for sets, constructors and destructors, and the initial process to be well-typed according to \( \Gamma \).

The concrete semantics presented here is a synchronous semantics, which we choose for simplicity and in accordance with the previous related work on ProVerif \[9\].

The rule \( \text{NIL} \) removes the process \( 0 \) when it holds no locks. The rule \( \text{COM} \) matches an input and an output processes if the output has the type required by the input; note that the set \( V_1 \) of influencing terms for the input process is increased with the term \( N' \) constructed from type \( T \) using the function \( \text{pt}^l_\rho \). The purpose of \( \text{pt}^l_\rho \) is to substitute any type \( T \) with a term \( N' \) that is homomorphic to \( T \): that is, for every occurrence of a name \( a \) in the type \( T \) it produces a variable \( x_a,V \) that is syntactically different from all other variable occurrences, and every occurrence of a constructor type produces a constructor term of the same form.

The rule \( \text{PAR} \) splits the process into two parallel processes. Replication \( \text{REPL} \) is annotated with \( k \in \mathbb{N} \) and produces a fresh copy of \( P \), adding \( x_a \) to \( V \) and the substitution \( \{ k/x_k \} \) to the environment \( \rho \); the replication process is annotated with the index \( k + 1 \) after the transition.

The rule for restriction \( \text{NEW} \) maps \( x \) to \( a[V] \) in the continuation of the process, where \( l \) is a unique label for the process \( \text{new}^l x : a ; P \). Here we extend the terms of Figure \[7\] to annotate names with a list of variables \( V \) under square brackets.

The rules for let reduce the process to \( P_1 \) where \( x \) is substituted with the result of the rewrite rule in case of success, and to \( P_2 \) otherwise.

To that end, we define the relation \( \rightarrow \rho \) as follows. Let \( s \) be a term that has only variables of atomic types and such that

---

**IV. SEMANTICS**

We define in Figure \[4\] an instrumented operational semantics for the language. We have transitions of the form \( \rho,S,P \rightarrow \rho',S',P' \) where:

- \( \rho : \mathbf{Var} \rightarrow \mathbb{M} \), an environment mapping process variables \((x,y,...)\) to name instances \((a'[\ldots])\),
- \( S \subseteq \mathcal{S} \times \mathbb{M} \) records the set-membership states,
- \( \mathcal{P} \subseteq \mathcal{P} \times \mathcal{S} \times \mathcal{S} \times \mathbb{M} \) is a multiset of concurrent processes, which are represented as triplets \((P,L,V)\) where \( P \) is a process, \( L \) is the set of locks held by \( P \), and \( V \) is a list of terms that influenced the process (either session identifiers or inputs).

A configuration \( \rho,S,P \) represents the parallel execution of all processes in \( \mathcal{P} \):

\[
\left| (P,L,V) \in \mathcal{P} \right| \rho(P_i)
\]
Lemma 1 (Type substitution). Let \( P \) be a process, \( \Gamma \) and \( \Gamma' \) two type environments, \( M \) a term and \( T \) a type. If \( x \notin \text{Dom}(\Gamma') \) and \( \Gamma[x \rightarrow T]\Gamma' \vdash P \) and \( \Gamma' \vdash \frac{M}{x} \) then \( L, \Gamma, \Gamma' \vdash P\{\frac{M}{x}\} \).

Proof sketch. The proof is carried out by induction on the shape of \( P \), and its sub-terms and boolean conditions.

In particular, when \( x \) is encountered in \( P \), we know that:

\[
\Gamma[x \rightarrow T]\Gamma'(x) = T
\]

is applied for the proof of \( L, \Gamma[x \rightarrow T]\Gamma' \vdash P \). Since \( x\{\frac{M}{x}\} = M \), our statement directly follows from the hypothesis.

Lemma 2 (Environment extension). Let \( \rho \) and \( \rho' \) be two environments such that \( \text{Dom}(\rho) \subseteq \text{Dom}(\rho') \) and for all \( x \) in \( \text{Dom}(\rho) \) we have \( \rho(x) = \rho'(x) \), let \( P \) be a process, \( \Gamma \) a type environment. If \( L, \Gamma[\rho] \vdash P \), then \( L, \Gamma[\rho'] \vdash P \).

Proof. By induction on the shape of \( P \).
a) Attacker processes: Because sets represent private information of the protocol, our attacker model does not have access to sets. We define it as follows:

**Definition 1 (Attacker process).** An attacker process is a well-typed process that shares a channel \( x_{eh} \) with the honest protocol, and cannot perform set operations (if, update, lock and unlock constructs are excluded).

V. AUTHENTICATION IN SET-\( \pi \)

A general definition of authentication goals for security protocols that has become standard in formal verification are Lowe’s notions of non-injective and injective agreement [19]. These notions are hard to combine with an abstract interpretation approach as they inherently include a form of negation that is incompatible with over-approximation. For this reason, ProVerif has a special notion of events that are handled in a special way by its resolution procedure. We now show that in our Set-\( \pi \) calculus we can directly express both non-injective and injective agreement using sets, and we can thus define events practically as syntactic sugar.

The definition of authentication is based on events \( e_1(M) \) and \( e_2(M) \) where message \( M \) typically contains the (claimed) sender and (intended) receiver name, as well as the data that the participants want to agree on; the event \( e_1(M) \) is issued by the sender typically at the beginning of the session and \( e_2(M) \) by the receiver at the end of the session, but the precise content and placement can be chosen by the modeler. One can then define non-injective agreement as follows: whenever an event \( e_2(M) \) happens for a message \( M \), then previously the event \( e_1(M) \) must have happened. Thus, it is an attack if somebody accepts a message that has not been sent that way. The injective agreement additionally requires that if \( e_2(M) \) has occurred \( n \) times, then \( e_1(M) \) must have previously occurred at least \( n \) times (i.e., there is an injective mapping from \( e_2 \) events to \( e_1 \) events). Roughly speaking, it is an attack if a message is accepted more often than it was actually sent. It is hard to automatically verify injective agreement in this context, the same way that is incompatible with over-approximation. For this reason, we define it as follows:

\[
\rho, S, \mathcal{P} \cup \{(\text{event } e_1(M); P, L, V)\} \rightarrow \rho, S, \mathcal{P} \cup \{(P, L, V)\}
\]

With this extension of the language, we can reason about non-injective and injective agreement properties according to Lowe’s definitions. We then encode processes in the extended calculus with events into processes in standard Set-\( \pi \) and show how our encoding simulates the events.

**Definition 2 (Non-injective agreement).** There is a non-injective agreement between event \( e_1(M) \) and event \( e_2(M) \) if and only if, for every possible trace \( \rho, S_0, \mathcal{P}_0 \rightarrow \rho, S_1, \mathcal{P}_1 \rightarrow \cdots \rightarrow \rho, S_n, \mathcal{P}_n \) produced by the protocol, if \( \rho_i, S_i, \mathcal{P}_i \rightarrow e_1(M) \rightarrow \rho_{i+1}, S_{i+1}, \mathcal{P}_{i+1} \) occurs in the trace, then also \( \rho_j, S_j, \mathcal{P}_j \rightarrow e_2(M) \rightarrow \rho_{j+1}, S_{j+1}, \mathcal{P}_{j+1} \) occurs, for some \( j < i \).

We construct a transformation from the extended language with events into the language without events, then prove the equivalence between Definition 2 in the original process (extended with events) and a set-property of the transformed process.

The transformation is as follows:

\[
\text{new } e : \text{event}(T); \text{Sys} \rightarrow \text{new } e : \text{set } T; \text{Sys}
\]

Every event declaration becomes a set declaration in the translated process (assuming that the names for sets and events are disjoint). Whenever an event \( e(M) \) occurs, where \( M \) is of type \( T \), we substitute it with the process that locks \( e \), inserts \( M \) in the set \( e \), unlocks \( e \) and continues with \( P \); we also add a set declaration for \( e \) in its scope. Furthermore, to gain precision in the analysis we merge a set transition followed by an event into a single operation. That is, if we have a process \( \text{update}(b^+); e(M); P \), we transform it into the process \( \text{lock}(e); \text{update}(b^+; M \in e); \text{unlock}(e); P \).

Note that this transformation is sound, although two semantic steps are merged into one: for the purpose of finding violations to an agreement property where \( e(M) \) should happen before \( e'(M) \), if there is a trace where event \( e'(M) \) happens between \( \text{update}(b^+) \) and event \( e(M) \), then there is also a trace where event \( e'(M) \) happens after the set operation. Given a process \( P \) we denote its event-free encoding as \( \text{agree}(P) \).

**Theorem 2.** Let \( P \) be an extended process with events. If there is no reachable state \( S \) from \( P' = \text{agree}(P) \) that satisfies the expression \( M \in e_2 \land M \notin e_1 \), then there is no non-injective agreement between \( e_1(M) \) and \( e_2(M) \) in \( P \).

**Proof sketch.** To prove the correctness of our transformation we construct a simulation relation between \( P \) and \( P' \), were the semantic step of an event is simulated by our construction. The full proof is found in the appendix. □

**Definition 3 (Injective agreement).** There is an injective agreement between event \( e_1(M) \) and event \( e_2(M) \) if and only if, for every possible trace \( \rho, S_0, \mathcal{P}_0 \rightarrow \rho, S_1, \mathcal{P}_1 \rightarrow \cdots \rightarrow \rho, S_n, \mathcal{P}_n \) produced by the protocol, if \( \rho_i, S_i, \mathcal{P}_i \rightarrow e_1(M) \rightarrow \rho_{i+1}, S_{i+1}, \mathcal{P}_{i+1} \) occurs in the trace, then also \( \rho_j, S_j, \mathcal{P}_j \rightarrow e_2(M) \rightarrow \rho_{j+1}, S_{j+1}, \mathcal{P}_{j+1} \) occurs, for some \( j < i \); furthermore, there does not exist \( k > i \) such that \( \rho_k, S_k, \mathcal{P}_k \rightarrow e_1(M) \rightarrow \rho_{k+1}, S_{k+1}, \mathcal{P}_{k+1} \).
For proving injective agreement properties the transformation becomes:

\[
\text{new } e : \text{event}(T); \text{Sys} \rightarrow \text{new } e : \text{set } T; \\
\text{event } e(M); P \rightarrow \text{lock}(e, \text{twice}-e); \\
\text{if } M \notin e \text{ then } \text{update}(M \in e); \\
\text{unlock}(e, \text{twice}-e); P \\
\text{else } \text{update}(M \in \text{twice}-e); \\
\text{unlock}(e, \text{twice}-e); P
\]

Every event declaration becomes a pair of set declarations for \( e \) and \( \text{twice}-e \) in the translated process (assuming that the names for sets and events are disjoint). Whenever an event \( e(M); P \) occurs, where \( M \) is of type \( T \), we substitute it with the process that locks \( e \) and \( \text{twice}-e \), and performs \( \text{update}(M \in e); P \) when \( M \) has not yet been inserted in \( e \); when it is already present in \( e \) it performs \( \text{update}(M \in \text{twice}-e); P \), and in both cases unlocks \( e \) and \( \text{twice}-e \); finally we add set declarations for \( e \) and \( \text{twice}-e \) in the scope. Similarly to the non-injective case, we merge a set operation with the event that follows. Given a process \( P \) we denote its event-free encoding as \( \text{inj-agree}(P) \).

\textbf{Theorem 3.} Let \( P \) be an extended process with events. If no reachable state \( S \) from \( P' = \text{inj-agree}(P) \) satisfies the expression \((M \in e_2 \land M \notin e_1) \lor (M \in \text{twice}-e_2)\), then the injective agreement between \( e_1(M) \) and \( e_2(M) \) holds in \( P \).

\textbf{Proof sketch.} To prove the correctness of our transformation we construct a simulation relation between \( P \) and \( P' \), where the semantic step of an event is simulated by our construction. The full proof is found in the appendix.

Relating back to our example, Figure 5 shows in green the desired transitions and in red the undesired ones. Our model satisfies non-injective agreement if no message is being accepted without being previously sent by the honest principal. It satisfies injective agreement if no message is accepted twice.

\section{VI. Translation}

The translation takes a process in Set-\( \pi \) and produces a set of Horn clauses that are then solved by a saturation based resolution engine, like ProVerif or SPASS.

At the end of the section we show how the translation is carried out for our CANAuth example. We now present the general concepts of the translation at an intuitive level, which we then refine with details later in the section. The translation produces clauses with predicates of the form \( \text{msg}(M, N) \) to denote that the system has produced an output of \( N \) on channel \( M \), predicates of the form \( \text{att}(M) \) to denote that the attacker process knows \( M \), predicates of the form \( \text{name}(a) \) to denote that a new name is produced by the protocol, and clauses that conclude transfer(\( \cdot, \cdot \)) to denote set-transitions.

The body of a Horn clause represents the inputs that are required to reach a specific point in the process, while the head of the clause represents the output that is generated. For example:

\[
\text{in}(ch, x : a); \text{in}(ch, y : b); \text{out}(ch, f(x, y))
\]

produces the clause:

\[
\text{msg}(ch, x) \land \text{msg}(ch, y) \Rightarrow \text{msg}(ch, f(x, y))
\]

Names and variables in the predicates are annotated with a special constructor \( \text{val} \) that defines their current membership class. For example if we have three sets in our system \( s_1, s_2, s_3 \), the term \( \text{val}(a, 1, 0, x_{s_3, a}) \) represents a name \( a \) in the process algebra in a state where \( a \) is in \( s_1 \), it is not in \( s_2 \) and its membership to the set \( s_3 \) is not constrained, as denoted by the variable \( x_{s_3, a} \). By doing so two clauses can be unified only if the terms are in consistent states. For example \( \text{val}(a, 1, 0, x_{s_3, a}) \) unifies with \( \text{val}(a, x_{s_1, a}, 0, 1) \) but not with \( \text{val}(a, x_{s_1, a}, 1, 1) \), because the first term represents a name \( a \) that is not in \( s_2 \), while the third term represents \( a \) in a state where it belongs to \( s_2 \).

Now we look at how the translation is constructed. We use a special function \( \alpha \), which we call the set-abstraction, to record whether a particular term belongs to some sets or not, and introduce the rules of Figure 6 to transform clauses, predicates and terms into annotated ones. The set-abstraction is a function of type:

\[
\alpha : (S \times M) \rightarrow (\{0, 1\} \cup \{x_{s, M} \mid s \in S, M \in M\})
\]

where we require:

\[
\alpha(s, M) \notin \{0, 1\} \implies \alpha(s, M) = x_{s, M}
\]

It takes a process set \( s \) and a term \( M \), and returns either the constant 1, to enforce that \( M \) is in \( s \), the constant 0,
\[ \text{restrict}(\alpha, M \in s) = \begin{cases} 1 & \text{if } M' = M \land s' = s \\ \alpha(s', M') & \text{otherwise} \end{cases} \]

where \( \alpha'(s', M') = \begin{cases} 1 & \text{if } M' = M \land s' = s \\ \alpha(s', M') & \text{otherwise} \end{cases} \)

\[ \text{restrict}(\alpha, a \notin s) = \begin{cases} 0 & \text{if } M' = M \land s' = s \\ \alpha(s', M') & \text{otherwise} \end{cases} \]

where \( \alpha'(s', M') = \begin{cases} 0 & \text{if } M' = M \land s' = s \\ \alpha(s', M') & \text{otherwise} \end{cases} \)

\[ \text{restrict}(\alpha, b_1 \land b_2) = \text{restrict}(\alpha', b_2) \mid \alpha' \in \text{restrict}(\alpha, b_1) \]

\[ \text{restrict}(\alpha, b_1 \lor b_2) = \text{restrict}(\alpha', b_1) \cup \text{restrict}(\alpha', b_2) \]

\[ \text{restrict}(\alpha, -b_1 \land b_2) = \text{restrict}(\alpha, -b_1) \lor (-b_2) \]

\[ \text{restrict}(\alpha, -b_1 \lor b_2) = \text{restrict}(\alpha, -b_1) \land (-b_2) \]

\[ \text{restrict}(\alpha, -b) = \text{restrict}(\alpha, b) \]

\[ \text{zero}(\alpha, a) = \alpha' \]

where \( \alpha'(s, M) = \begin{cases} 0 & \text{if } a \text{ occurs in } M \\ \alpha(s, M) & \text{otherwise} \end{cases} \)

\[ \text{relax}(\alpha, L) = \alpha' \]

where \( \alpha'(s, M) = \begin{cases} \alpha(s, M) & \text{if } s \in L \\ x_{s,M} & \text{otherwise} \end{cases} \)

Fig. 7. Functions for updating \( \alpha \)

to enforce that \( M \) is not in \( s \), or the variable \( x_{s,M} \), to allow one of the two choices to be picked consistently across the hypotheses.

The function \( \llbracket p \rrbracket_\alpha \) of Figure 6 recursively applies the set-abstraction to the clauses. When it encounters an annotated name \( a'\llbracket V \rrbracket \) in the protocol, it produces a constructor \( \text{val}(\ldots) \) where the first parameter is the name itself—with no annotations in case of attacker names—and the remaining parameters represent the membership of \( a'\llbracket V \rrbracket \) to the sets \( s_1, \ldots, s_n \); similar clauses are generated for occurring variables. For the purpose of making the analysis feasible, as a well-formedness condition we require set types (of the form set \( T \)) to contain only name types and monadic constructors over name types. For example, set \( \text{Seed} \), set \( \text{pk}(\text{Seed}) \) and set \( \text{sk}(\text{Seed}) \) are acceptable set types, while \( \text{key}(\text{Seed}, \text{Nonce}) \) is not a monadic constructor, hence set \( \text{key}(\text{Seed}, \text{Nonce}) \) is not an acceptable set type.

The function \( \llbracket P \rrbracket H V L \alpha \) of Figure 8 takes a process \( P \), a set of hypothesis predicates \( H \), that intuitively represent the set of messages required to reach \( P \), a list of influencing terms for the process \( V \), a set of locks \( L \) held by the process, and the set-abstraction \( \alpha \), and produces a set of clauses representing the protocol behaviour.

Lastly the set of functions \( \text{restrict}, \text{zero} \) and \( \text{relax} \) modify the set-abstraction for various constructs of Set-\( \pi \). The function \( \text{restrict} \) takes a set-abstraction \( \alpha \) and a boolean formula \( b \) and produces the set of all consistent abstractions that satisfy \( b \), while \( \text{zero} \) inserts the constant 0 for fresh names, and \( \text{relax} \) introduces variables in the image of \( \alpha \) for unlocked sets.

Having introduced the auxiliary functions for manipulating the set-abstraction, we now come back to explaining the translation process. The function \( \text{relax}(\alpha, L) \) is applied at each step of the translation, as it inserts variables in the image of \( \alpha \) for all sets that are not locked, as they may be changed by other processes.

The translation for 0 produces an empty set of clauses. Replication \( !P \) translates \( P \) with the introduction of a new session variable \( x_1 \) in the list of influencing variables \( V \). Parallel composition \( P_1 \parallel P_2 \) is translated as the union of the clauses generated by both processes.

Input \( in(M, x : T) \) adds the predicate \( \text{msg}(M, N') \) as an hypothesis in \( H \), where \( N' \) is a copy of \( T \) where every occurrence of a name type is replaced with a unique variable using \( \text{pt}_x(T) \); the substitution \( \{N'/x\} \) is then applied on the continuation. Output \( out(M, N) \) produces a clause with head \( \llbracket \text{msg}(M, N) \rrbracket \alpha \) and with hypotheses \( \llbracket H \rrbracket \alpha \). The rule for new \( x : a \) introduces a restricted name: the value class of \( a'\llbracket V \rrbracket \) is set to 0 for every set, the predicate name \( \alpha'\llbracket V \rrbracket \) is introduced both in the hypotheses for analysing the continuation and as a fact that follows the current set of hypotheses \( H \). This ensures that all the set-abstraction variables occurring in the head of a clause are closed under the hypotheses.

The rule for let \( x = g(M_1, \ldots, M_n) \) in \( P_1 \) else \( P_2 \) looks for a substitution \( \sigma \) that successfully unifies a definition of the rewrite rule for the destructor \( g \) with the actual parameters \( M_1, \ldots, M_n \), and then finds a substitution \( \theta \) that unifies the terms in the set-abstraction \( \alpha \) accordingly to the unification on the process algebra terms; if both substitutions are found then \( \sigma(P_1) \) is analysed where \( x \) is substituted with the result of the reduction. The rule also includes the clauses generated for \( P_2 \) on the updated state \( \alpha' \), with no restriction, which is a standard over-approximation found in similar works, e.g. [7].

The rule for if \( b \) then \( P_1 \) else \( P_2 \) translates \( P_1 \) with all the set-abstractions that satisfy the formula \( \neg b \). The rule for lock(s); translates the continuation by first introducing \( s \) in the locked sets \( L \), and applying \( \text{relax} \) to take into account state changes from other processes before the lock takes place. Similarly, the rule for unlock(s); translates the continuation by removing \( s \) from the set \( L \), and applying \( \text{relax} \). The rule for update(\( b^+ \));, for every name and variable occurring in \( b^+ \) that we denote by \( M \), creates a clause of the form \( \llbracket H \rrbracket_{\alpha'} \Rightarrow \llbracket \text{transfer}(\llbracket M \rrbracket_{\alpha'}, \llbracket M \rrbracket_{\alpha' \cup \{(s, M)\}} \rrbracket) \rrbracket \), in order to mark that whenever \( M \) appears in a predicate on state \( \alpha' \), we will also have the same predicate on state \( \alpha' \cup \{(s, M)\} \), then proceeds to translate the continuation.

a) Clauses representing the attacker: We add the following set of clauses to represent a Dolev-Yao attacker. The attacker can eavesdrop messages form known channels:

\[ \text{msg}(x_{ch}, x_{msg}) \land \text{att}(x_{ch}) \Rightarrow \text{att}(x_{msg}) \]

The attacker can insert known messages into channels:

\[ \text{att}(x_{ch}) \land \text{att}(x_{msg}) \Rightarrow \text{msg}(x_{ch}, x_{msg}) \]

For every \( n \)-ary constructor \( f \) occurring in the protocol we produce a clause:

\[ \text{att}(x_1) \land \cdots \land \text{att}(x_n) \Rightarrow \text{att}(f(x_1, \ldots, x_n)) \]
be the set of clauses produced by the translation $C$ (for any context predicate: roughly speaking, when the fixedpoint contains introduce clauses that express the meaning of the as well as the public channel shared with the honest protocol: $[\{H \Rightarrow \text{msg}(M, N)\}]_\alpha$

\[ \text{new} \ x \ a; \ P \ H V L \alpha = [P\{a[V]/\}](H \land \text{name}(a[V]))V L \alpha' \cup \{H \Rightarrow \text{name}(a[V])\}_\alpha' \]

where \( \alpha' = \text{zero}(\alpha, [a[V]]) \)

\[ \text{let } x = g(M_1, \ldots, M_n) \text{ in } P_1 \text{ else } P_2 \ H V L \alpha = \]

\[ \{[[P_1]] \sigma(H) \sigma(V) \Rightarrow \alpha'' \mid \text{reduc } \forall \tilde{x} : \tilde{T}, \ g(M'_1, \ldots, M'_n) \rightarrow M' \} \text{ in the scope of let,} \]

\( \sigma \) is an m.g.u. that satisfies \( M_1 = M'_1 \land \ldots \land M_n = M'_n \land x = M' \),

\( \theta \) is an m.g.u. that satisfies \( \forall s, N_1, N_2, \sigma(N_1) = \sigma(N_2) \Rightarrow \alpha'(s, N_1) \equiv \alpha'(s, N_2) \),

and \( \alpha'' \) satisfies \( \forall N, \alpha'(s, \sigma(N)) = \theta(\alpha'(s, N)) \)

\[ \cup \{P_2\} H V L \alpha' \]

where \( \alpha' = \text{relax}(\alpha, L) \)

\[ \text{if } b \text{ then } P_1 \text{ else } P_2 \ H V L \alpha = \{[[P_1]] \text{restrict}(\text{relax}(\alpha, L), b) \} \cup \]

\[ \{[[P_2]] \text{restrict}(\text{relax}(\alpha, L), \neg b) \} \]

\[ \text{lock}(L'); \ P \ H V L \alpha = [P] H V L (L \cup L')(\text{relax}(\alpha, L)) \]

\[ \text{unlock}(L'); \ P \ H V L \alpha = [P] H V L (L \cup L')(\text{relax}(\alpha, L)) \]

\[ \text{update}(b^+); \ P \ H V L \alpha = \{[H]_\alpha \Rightarrow \text{transfer}(\{M\}_\alpha, \{M\}_\alpha') \mid M \in f_v(b^+) \cup f_n(b^+) \} \cup [P] H V L \alpha'' \]

where \( \alpha' = \text{relax}(\alpha, L) \) and \( \alpha'' = \text{update}(\alpha, b^+) \) and \( \alpha''' = \text{relax}(\alpha'', L) \)

Fig. 8. Translation rules for processes into Horn clauses

For all destructors of the form \( g(M_1, \ldots, M_n) \rightarrow M \) we produce a clause:

\[ \text{att}(M_1) \land \ldots \land \text{att}(M_n) \Rightarrow \text{att}(M) \]

Finally, the attacker knows a name for each name type \( a \) in the initial state \( S_0 \):

\[ \Rightarrow \text{att}(a^T[\]) \} S_0; \Rightarrow \text{name}(a^T[\}) S_0 \]

as well as the public channel shared with the honest protocol:

\[ \Rightarrow \text{att}\{ch_{\text{in}}[\} S_0 \]

\( b) \) Clauses representing the set-transitions: We now introduce clauses that express the meaning of the transfer(\{\cdot\}) predicate: roughly speaking, when the fixedpoint contains transfer(\( M, M' \)), then for every fact \( C[M] \) also \( C[M'] \) holds (for any context \( C[\cdot] \)). Since there are infinitely many contexts \( C[\cdot] \), we cannot directly write this as Horn clauses, but it actually suffices to restrict ourselves to contexts that occur on the right-hand side of a Horn clause. More precisely, let \( Cl \) be the set of clauses produced by the translation \( [P_0]000\alpha_0 \). Let \( M_\alpha, M'_\alpha \) be two terms of type \( a \), and consider every clause \( H \Rightarrow C[M_\alpha] \in Cl \) for some context \( C[\cdot] \) that is not a transfer predicate. Finally let \( \alpha \) and \( \alpha' \) be two set-abstractions such that \( H' \Rightarrow \text{transfer}(\{M'_\alpha\} \alpha, \{M'_\alpha\} \alpha') \in Cl \), and let \( \sigma = \text{msg}(M_\alpha, M'_\alpha) \). For each such case we add the following Horn clause:

\[ \{C[M'_\alpha]\}_\alpha \sigma \land \text{transfer}(\{M'_\alpha\} \alpha, \{M'_\alpha\} \alpha') \Rightarrow \{C[M'_\alpha]\}_\alpha' \sigma \]

This set of clauses transfers messages and attacker knowledge between states. Hence if a predicate is derivable in the saturation in state \( \alpha \) and there is a transition from \( \alpha \) to \( \alpha' \), then the predicate will be derivable in state \( \alpha' \).

Intuitively this set of rules suffices for the translation because only the honest protocol produces state transitions; everything that the attacker can derive in a state, it can also derive in the successor state. Therefore it is only necessary to transfer the conclusions for the protocol. Lemma 5 establishes the correctness of this approach.

d) Translation of CANAuth into Horn clauses: To show how the translation is applied to produce Horn clauses from the original description, we have taken an excerpt from our running example, namely the receiving process, translated the events into set transitions, and label each point of the program:

\[ P_1 \triangleq \{r,a,at\} \text{in}_{1s}(ch, \{x_m, x_s\} : \{\text{msg}(cnt), \text{hmac}(\text{msg}(cnt), key)\} ; 1s) \]

let \( _\_ = \text{eq}(x_s, \text{hmac}(x_m, k)) \) in \( 14 \)

let \( x_c = \text{getcnt}(x_m) \) in \( 15 \)

if \( x_c \notin r \) then \(16 \)

if \( x_m \notin a \) then \(17 \text{update}(x_c \in r; x_m \in a); 18 \)

else \(19 \text{update}(x_c \in r; x_m \in at); 1\alpha \)

Figure 9 shows the clauses that are generated, together with the recursive calls of \( [\cdot] \) that are required to produce
them. We use here the notation like \((H_1 = \emptyset)\) to indicate the development of the parameters \(H, V, L\) and \(\alpha\) over the recursive calls of \([\cdot]\), and \(P^l\) to denote the subprogram of \(P\) at label \(l\).

VII. Correctness

In this section we want to establish the correctness of our translation with respect to the semantics of Section [IV]. We use the inference system of Figure [IV] to express our correctness results. Intuitively, this set of rules relates the instrumented semantics to the Horn clauses generated by the translation, namely that the fixed-point \(\mathcal{F}_{P_b}\) covers all possible behaviours of a process, when started in the given configuration \((\rho, L, V, S)\) and in any environment that cannot change sets in \(L\).

Let \(S\) and \(S'\) again be states of the sets (i.e., \(S(s)\) yields the elements that are members of set \(s\) in state \(S\)); we can view a state as a special case of an abstraction \(\alpha\) that has no variables (i.e., indetermined set memberships) and we can thus write \([^\cdot]_S\) accordingly. We will now show: if the semantic relation induces a reachable state \(S\) at which output \(N\) on channel \(M\) is produced, then the Horn clauses generated for the protocol entails the ground fact \([^\text{msg}(M, N)]_S\). This ensures that whatever behaviour is present in the semantics is also captured by the translation.

We denote by \(C_{P_b}\) the set of clauses produced by the translation, including the fixed clauses, and by \(\mathcal{F}_{P_b}\) the set of ground facts derivable from \(C_{P_b}\). First we introduce the order relation \(\sqsubseteq_L\) on the facts \(\mathcal{F}_{P_b}\) derivable from the initial protocol.

**Definition 4** (Order relation \(\sqsubseteq_L\)). The order relation \(S_1 \sqsubseteq_L S_2\) between states \(S_1\) and \(S_2\) holds iff:

(i) \(\forall s_j \in L \cdot S_1(s_j) = S_2(s_j)\);
(ii) \(\forall p(M_1, \ldots, M_k) \cdot [p(M_1, \ldots, M_k)]_{S_1} \in \mathcal{F}_{P_b} \Rightarrow [p(M_1, \ldots, M_k)]_{S_2} \in \mathcal{F}_{P_b}\).

Intuitively the \(\sqsubseteq_L\) relation captures the causal relation of the semantic rules, as condition (ii) requires all predicates of the form msg, name and att to be transferred from state \(S_1\) to state \(S_2\). Furthermore condition (i) imposes that the locked sets \(L\) are not modified between the two states. The most general of such relations is \(\sqsubseteq_0\), as it allows any set to be modified.

Next we formalise the definition for set-abstraction \(\alpha\) that was introduced in Section [VI].

**Definition 5** (Set-abstraction). The mapping \(\alpha\) abstracts \(S\) under the environment \(\rho\) iff for every set \(s\), term \(M\), either \(\alpha(s, M) = 1\) and \(\rho(M) \in S(s)\), or \(\alpha(s, M) = 0\) and \(\rho(M) \notin S(s)\), or \(\alpha(s, M) = x_{s,M}\).

A set abstraction \(\alpha\) abstracts a state \(S\) if every pair \((s, M)\) that maps to a variable in \(\alpha\) is mapped to a variable that is unique in the image (this is ensured syntactically by the use of \(x_{s,M}\)), and whenever \(\alpha(s, M)\) maps to the constants 1 and 0 then \(\rho(M) \in s\) and \(\rho(M) \notin s\), respectively, in the state \(S\).

The following lemmata establish the relation between the operations used in the translation and the order relation \(\sqsubseteq_L\). The interested reader can find the full proofs in the extended version of this article.

**Lemma 3** (relax preserves the set-abstraction over \(\sqsubseteq_L\)). Let \(S, S'\) be two states such that \(S \sqsubseteq_L S'\), and assume \(\alpha\) abstracts \(S\) under \(\rho\). Then \(\alpha' = \text{relax}(\alpha, L)\) abstracts \(S'\) under \(\rho\).

Since relax inserts unique variables in \(\alpha'\) for all sets that are not locked, and for all sets \(s\) that are locked \(\alpha(s) = \alpha'(s)\) and \(S(s) = S'(s)\) holds by condition (i) of the order relation, then \(\alpha\) satisfies the properties of Definition [V].

**Lemma 4** (restrict preserves the set-abstraction). Let \(\alpha\) be a set abstraction, \(\rho\) an environment, \(S\) a state and \(A = \text{restrict}(\alpha, b)\). If \(\rho, S \models b\) and \(\alpha\) abstracts \(S\), then there exists an \(\alpha' \in A\) such that \(\alpha'\) abstracts \(S\). Restrict produces a set of set-abstractions each representing a possible way of satisfying the formula \(b\). Lemma 4 establishes that if \(\alpha\) abstracts \(S\) then at least one of these restrictions on \(\alpha\) satisfies the abstraction of \(S\).

**Lemma 5** (transfer preserves \(\sqsubseteq_L\)). Let \(S\) be a set-membership state, and \(S' = S \cup \{(s_1, M_1), \ldots, (s_j, M_j)\} \setminus \{(s_{j+1}, M_{j+1}), \ldots, (s_n, M_n)\}\). If for all \(M \in \{M_1, \ldots, M_n\}\) we have transfer\((\{M\}, S)\) \(\in \mathcal{F}_{P_b}\) then for any set of locks \(L\) such that \(\{s_1, \ldots, s_n\} \cap L = \emptyset\) we have \(S \sqsubseteq_L S'\).

Lemma 5 establishes that transfer predicates actually capture the state transitions, hence following the definition of the order \(\sqsubseteq_L\) the set of predicates derivable in the updated state is larger than that derivable in the original state.

Next we type the attacker process \(A\) and the honest protocol \(P_0\), under the initial environment \(\rho_0 = [x_{ch} \mapsto ch^0]\).

**Lemma 6** (Typability of \(A\)). Let \(A\) be an attacker process, then \(\rho_0, \emptyset, \emptyset, S_0 \models A\).

**Proof sketch.** Let \(B\) be a subprocess of \(A\), \(\rho\) an environment, \(S\) a state, \(V\) a list of terms. We prove that if:

(i) \(\rho(B)\) is a closed process, \(\rho(V)\) is ground,
(ii) \(S_0 \sqsubseteq_0 S\), and
(iii) for every maximal subterm \(M\) of \(B\) closed under \(\rho\), we have \(\rho(\text{att}(M)) \in \mathcal{F}_{P_b}\),

then:

\(\rho, V, \emptyset, S \models B\).

Proof by induction over the depth of \(B\). In particular, we have that (i) \(f v(A) = x_{ch}\), hence \(\rho_0(A)\) is closed; (ii) \(S_0 \sqsubseteq_0 S_0\) by reflexivity; and (iii) the only maximal subterm of \(A\) that is bound by \(\rho_0\) is \(x_{ch}\), and by construction of the translation \(\rho_0(\text{att}(x_{ch}))\) \(\in \mathcal{F}_{P_b}\). Hence the attacker process types.

**Lemma 7** (Typability of \(P_0\)). \(\rho_0, \emptyset, \emptyset, S_0 \models P_0\).

**Proof sketch.** Let \(Q\) be a process. We prove that, given a list of terms \(V\), a set of locks \(L\), a state \(S\), a set-abstraction \(\alpha\), an environment \(\rho\), if:

(i) \(\rho(Q)\) is a closed process, \(\rho(V)\) and \(\rho(H)\) are ground,
(ii) \(\alpha\) abstracts \(S\) under \(\rho\),
(iii) \(C_{P_b} \supseteq [Q] H V L \alpha\),
(iv) for every predicate \(p\) in \(H\), we have that \(\rho(p) \in \mathcal{F}_{P_b}\)

Then \(\rho, V, L, S \models Q\).
\[ [B][0][0][0][\alpha_0] = [P^{x_1}] (H_1 = \emptyset) (V_2 = [x_{\text{rel}}]) (L_1 = \emptyset) (\alpha_1 = \text{relax}(\alpha_0, \emptyset)) = [P^{x_2}] (H_2 = \emptyset) (V_2 = V_1) (L_2 = \{r, a, at\}) (\alpha_2 = \text{relax}(\alpha_1, \emptyset)) = [P^{x_3}] (\alpha_3 = \text{relax}(\alpha_2, L_2)) = [P^{x_4}] (\alpha_4 = \theta(\text{relax}(\alpha_3, L_3))) = [P^{x_5}] (\alpha_5 = \text{restrict}(\text{relax}(\alpha_4, L_4), x_{\text{cnt}, 1} \notin r)) = [P^{x_6}] (\alpha_6 = \text{restrict}(\text{relax}(\alpha_5, L_5), msg(x_{\text{cnt}, 1}) \notin a)) \]

\[ [P^{x_7}] (H_7 = H_6) (V_7 = V_6) (L_7 = L_6) (\alpha_7 = \text{restrict}(\text{relax}(\alpha_6, L_6), msg(x_{\text{cnt}, 1}) \in a)) \]

\[ [P^{x_8}] (H_8 = H_7) (V_8 = V_7) (L_8 = L_7) (\alpha_8 = \theta(\text{relax}(\alpha_7, L_8))) \]

\[ [P^{x_9}] (H_9 = H_8) (V_9 = V_8) (L_9 = L_8) (\alpha_9 = \text{restrict}(\text{relax}(\alpha_8, L_8), msg(x_{\text{cnt}, 1}) \notin a)) \]

\[ [P^{x_{10}}] (H_{10} = H_9) (V_{10} = V_9) (L_{10} = L_9) (\alpha_{10} = \text{restrict}(\text{relax}(\alpha_9, L_9), msg(x_{\text{cnt}, 1}) \in a)) \]

**Fig. 9. Horn clauses generated by the translation**

The proof is carried out by induction on the structure of the process \( Q \).

In particular, (i) \( \rho_0 \) closes \( P_0 \) by construction, \( \rho_0(\emptyset) \) is trivially ground, (ii) \( \alpha_0 \) abstracts \( S_0 \) under \( \rho_0 \) by construction, (iii) \( C_{\rho_0} \supseteq [P_0][0][0][0][0][\alpha_0] \) by definition of the translation, (iv) holds vacuously. Therefore the conditions (i-iiv) are satisfied and hence \( \rho_0, \emptyset, \emptyset, S_0 \models P_0 \).

**Theorem 4** (Subject reduction). If \( \rho, S, \mathcal{P} \rightarrow \rho', S', \mathcal{P}' \) and for all \( (P, L, V) \in \mathcal{P} \) we have \( \rho, V, L, S \models P \) then for all \( (P', L', V') \in \mathcal{P}' \) we have \( \rho', V', L', S' \models P' \).

The proof is a case-by-case analysis on the semantic rules for the language.

**Theorem 5** (Correctness of the analysis). Let \( S\{\cdot\} \) be the system context, let \( P_0 \) be the protocol, let \( T \) be the set of types used by \( P_0 \), let \( A \) be any attacker process using only types in \( T \), and \( \rho_0 = [x_{\text{ch}} \mapsto ch[0]] \).

If the typing \( \langle P_0 \mid A, \emptyset, \emptyset \rangle \) holds, and if \( \rho_0, S_0, \mathcal{P}_0 = \langle (P_0 \mid A, \emptyset, \emptyset) \rightarrow \rho_n, S_n, \mathcal{P}_n = \mathcal{P}_n \uplus \{(\text{out}(M, N); P', L, V)\} \rangle \) then \( \langle \rho_n(M, N) \rangle \models S_n \in \mathcal{F}_{P_0} \).

**Proof.** By Lemma 7 we know that \( \rho_0, \emptyset, \emptyset, S_0 \models P_0 \); by Lemma 3 we know that \( \rho_0, \emptyset, \emptyset, S_0 \models A \), hence all processes in \( \mathcal{P}_0 \) type in the initial state \( S_0 \).

Let \( \rho_0, S_0, \mathcal{P}_0 \rightarrow \rho_1, S_1, \mathcal{P}_1 \rightarrow \cdots \rightarrow \rho_n, S_n, \mathcal{P}_n. \) By inductively applying Theorem 4 on the length of the trace \( n \) we can conclude that all processes in \( \mathcal{P}_n \) type in the \( S_n \).

In particular, the process \( \text{out}(M, N); \mathcal{P}' \) in state \( S_n \) and hence \( \langle \rho_n(M, N) \rangle \models S_n \in \mathcal{F}_{P_0} \).

**Corollary 1** (Checking queries). If \( \rho_0, S_0, \mathcal{P}_0 = \{(P_0 \mid A, \emptyset, \emptyset) \rightarrow^* \rho_j, S_j, \mathcal{P}_j = \mathcal{P}_j \uplus \{(\text{out}(M, N); P', L, V)\} \rightarrow^* \rho_n, S_n, \mathcal{P}_n \) and \( \rho_n, S_n \models b \) then there exists \( \alpha_0 \in \text{restrict}(\alpha_0, b) \) where \( \theta = \text{mgu}(\alpha_n, \alpha_b) \) and \( \langle \rho_n(M, N) \rangle \models \alpha_0 \).

**Proof.** Follows from Theorem 5 and because \( S_j \preceq S_n \).

Therefore we can express any query of the form:

\[ \text{msg}(M, N) \] where \( b \) is a boolean expression ranging over names and monadic constructors in \( M \) and \( N \). Queries of this form are general enough to model secrecy from the attacker’s perspective (assuming that the channel \( M \) is public), as well as the authentication properties discussed in Section VII.

**VIII. EXPERIMENTAL EVALUATION**

We implemented our analysis into a prototype tool written in Haskell, that translates processes specified in Set-\( \pi \) and uses ProVerif as a back-end resolution engine for Horn clauses. The tool is available for download at [11].

Figure [11] shows the results for our examples: the running example on CANAuth, a flawed version of MaCAN [12], a key registration protocol, and an implementation of the Yubikey protocol modeled after [13].

We recorded the running times for our test suite on an Intel Core i3-7100U at 2.40GHz with 8 GB of RAM running Linux.

They are comparable to similar ProVerif models in applied-\( \pi \), which shows that there is little overhead induced by our specific translation. In the next two subsections we present the key registration protocol and the Yubikey example.

**A. Key Registration (with Attack)**

Here we present a key-registration protocol where an honest principal \( A \) registers its current pair of asymmetric keys
In turn

\[ A \Rightarrow \text{sk} \]

\[ \text{pk} \]

which encodes the new public key \( pk' \) with the old secret key \( sk \). \( S \) will be able to decrypt \( A \)'s message with the old public key \( pk \), move \( pk \) from the set of valid keys to the database of revoked keys and send back an acknowledgment to \( A \).

\[ S \Rightarrow A : \text{penc}(pk', (\text{confirm})) \]

In turn \( A \) will be able to decrypt this message with \( sk' \) and remove the old \( sk \) from its key-ring.

\[ A \triangleq \text{in}(kdb_a, sk_a : S\text{Key}); \]

if \( sk_a \in \text{ring}_a \) then

\[ \text{new} s'_a : \text{Seed}; \]

\[ \text{out}(ch, \text{senc}(sk_a, (\text{new key}, a, pk(s'_a)))); \]

\[ \text{out}(kdb_a, sk_a) ; \]

\[ \text{in}(ch, x_c : \text{senc}(PKey, x_t)); \]

let \( x_r = \text{pdec}(sk(s'_a), x_c) \) in

if \( x_r = \text{confirm} \) then

\[ \text{update}(sk_a \notin \text{ring}_a); \]

\[ \text{out}(ch, sk_a); 0 \]

\[ S \triangleq \text{in}(ch, x_a : \text{senc}(S\text{Key}, (x_t, x_{t'}, PKey))); \]

\[ \text{in}(kdb_a, pk_a : PKey); \]

let \((=\text{new}, a, pk'_a) = \text{pdec}(pk_a, x_s)\) in

if \( pk_a \in \text{valid}_a \land pk'_a \notin \text{valid}_a \land pk'_a \notin \text{revoked}_a \) then

\[ \text{update}(pk_a \in \text{revoked}_a); pk_a \notin \text{valid}_a; pk'_a \notin \text{valid}_a); \]

\[ \text{out}(ch, \text{penc}(pk'_a, \text{confirm})); \]

\[ \text{out}(kdb_a, pk'_a); 0 \]
\( Sys \triangleq \text{reduc } \forall x : \text{Seed}, m : t . \)
\[
\text{pdec}(pk(x), senc(sk(x), m)) \rightarrow m;
\]
\( \text{reduc } \forall x : \text{Seed}, m : t . \)
\[
\text{sdec}(sk(x), penc(pk(x), m)) \rightarrow m;
\]
\( \text{reduc } \forall x : \text{Seed}. \text{keypair}(sk(x), pk(x)) \rightarrow \text{true}; \)
new valid_a : set sk(Seed);
new revoked_a : set pk(Seed);
new kdb_a : SKey; new kdb_s : SKey; new s_a : Seed;
update(sk(s_a) \in \text{ring}_a; pk(s_a) \in \text{valid}_a);
out(kdb_a, sk(s_a));
out(kdb_s, pk(s_a));
\{ \{ \text{ring}, \text{valid}, \text{revoked} \} \} S)

Once a new key is established and the client receives
confirmation from the server that the secret key \( sk \) has been
revoked, \( sk \) can be revealed to the attacker. An attacker
succeeds in breaking the protocol when she discovers a secret
key that is still registered to the server.

We express the property with the following query:
\[
\text{att}(s) \text{ where } pk(s) \text{ in } \text{valid}_a
\]
with this query we find the attack that was originally discov-
ered by Cheval et al. [14].

**B. Zeb’s Key Registration in SetPi**

Here we present an adaptation of Zeb’s Key Registration
protocol from [21] with two clients, \( a \) and \( b \), and one server
\( s \), each running the protocol. The \textit{Init} process models the
out-of-band key registration, the \textit{Client} process models the
client requesting a new key, and the \textit{Server} process models the
server-side handling of the request. All processes are
parameterized on the agent names and sets on which they
operate:

\textit{Init}(a, ring, valid) \triangleq
lock(ring, valid);
new sk_1 : skey;
let pk_1 : pkey = pk(sk_1) in
update(sk_1 in ring \land \text{pk}_1 \text{ in valid});
out(d, sk_1);)
out(ch, pk_1);)
unlock(ring, valid);.

\textit{Client}(a, ring, revoked) \triangleq
lock(ring);
new sk_2 : skey;
in(d, sk_1 : skey);
if sk_1 in ring then
let pk_2 : pkey = pk(sk_2) in
update((sk_2 in ring) \land \neg sk_1 in ring);
out(ch, sign(sk_1, (a, pk_2)));)
unlock(ring);
lock(revoked);
let pk_1 : pkey = pk(sk_1) in
if pk_1 in revoked then
out(ch, sk_1);)
unlock(revoked);.

\textit{Server}(a, valid, revoked) \triangleq
lock(valid, revoked);
in(ch, pk_1 : pkey);
if pk_1 in valid then
in(ch, x_msg : sign(skey, (agent, pkey)));
let \( = a, x, pk_2 : pkey = \text{checks}(pk_1, x, msg) \) in
if \neg x \text{in} \text{revoked} then
update(\neg \text{pk}_1 \text{in} \text{valid}) \land \text{pk}_1 \text{in} \text{revoked}
\land \text{pk}_2 \text{in} \text{valid});
unlock(valid, revoked);.

\( Sys \triangleq \text{reduc } \forall x : \text{SKey}, y : t.
\)
\[
\text{checks}(pk(x), sign(x, y)) = y;
\]
\( \text{reduc } \forall x : \text{SKey}, y : t.
\)
\[
\text{getmsg}(sign(x, y)) = y;
\]
new ring_a : set SKey; new ring_s : set SKey;
new valid_as : set pk(SKey);
new valid_bs : set pk(SKey);
new revoked_as : set pk(SKey);
new revoked_bs : set pk(SKey);
new a : agent; new b : agent;
new d : channel;
\{ \text{Init}(a, ring_a, valid_as) \} |
\{ \text{Init}(b, ring_b, valid_bs) \} |
\{ \text{Client}(a, ring_a, revoked_as) \} |
\{ \text{Client}(b, ring_b, revoked_bs) \} |
\{ \text{Server}(a, valid_as, revoked_as) \} |
\{ \text{Server}(b, valid_bs, revoked_bs) \} |

As in the previous model, we express the security property
with the following query:

\[
\text{att}(s) \text{ where } pk(s) \text{ in } \text{valid}
\]
for \( \text{valid} \in \{ \text{valid}_as, \text{valid}_bs \} \).

**C. Yubikey**

Yubikey is a small USB token used to authenticate to
supported online services. It works by maintaining a pair of
a secret identity (shared with the server) and a public identity
(shared publicly), and by sending to the Yubikey server its own
public identity, together with the one time password encrypted
with the current value of a counter using a shared key \( k \).
Here we model a simplified version of the Yubikey protocol, where we are interested in the injective agreement between the client Yubikey \((YK)\) and the server \((Srv)\). The process \(BP\) represents the process activated by pressing the Yubikey button, which authenticates the user to the server. We define a public channel \(ch\), and a private channel \(ch\_server\) that is only used to securely exchange the secret identity and shared key to the server.

The Yubikey process \(YK\) creates a new fresh key \(k\), its own public and secret identities \((x_{pid} \text{ and } x_{sid})\), stores them securely to the server, then reveals its public identity and starts the \(BP\) process.

The button press \((BP)\) process initiates the authentication procedure, increasing the counter (this is encoded in our calculus by the creation of a fresh value), producing the nonces \(x_{nonce}\) and \(x_{tpr}\), and sending the encrypted message. An event \(yk\_press\) is inserted to denote that the button has been pressed.

The server on the other end receives the login request from the Yubikey, retrieves its secret identity and key \(k\) from its own channel, pattern matching on the Yubikey’s public identity to find the right tuple, decrypts the message with the retrieved key \(k\), and finally if the counter has not been used, it issues a \(yk\_login\) event to conclude the protocol.

\[\begin{align*}
 Sys & \triangleq \text{new } yk\_press : \text{event}(cnt); \\
 & \quad \text{new } yk\_login : \text{event}(cnt); \\
 & \quad \text{new } used : \text{set } cnt; \\
 & \quad \text{reduc } \forall x : t ; k : \text{key} . \ sdec(senc(x, k), k) \rightarrow x; \\
 & \quad \text{new } ch : \text{channel}; \\
 & \quad \text{new } ch\_server : \text{channel}; \\
 & \quad (\lnot YK | \lnot Srv)
\end{align*}\]

\[\begin{align*}
 BP & \triangleq \text{new } x_c : \text{cnt}; \\
 & \quad \text{new } x_{nonce} : \text{nonce}; \\
 & \quad \text{new } x_{tpr} : \text{nonce}; \\
 & \quad \text{event } yk\_press(x_c); \\
 & \quad \text{out}(ch, \langle x_{pid}, x_{nonce}, senc(x_{sid}, x_{c}, x_{tpr}), k \rangle); \\
 & \quad 0
\end{align*}\]

\[\begin{align*}
 YK & \triangleq \text{new } k : \text{key}; \\
 & \quad \text{new } x_{pid} : \text{pid}; \\
 & \quad \text{new } x_{sid} : \text{sid}; \\
 & \quad \text{out}(ch\_server, \langle x_{pid}, x_{sid}, k \rangle); \\
 & \quad \text{out}(ch, x_{pid}); \\
 & \quad \lnot BP
\end{align*}\]

\[\begin{align*}
 Srv & \triangleq \text{in}(ch, \langle x_{pid}, x_{nonce}, x_{enc} \rangle : \\
 & \quad (\text{pid, nonce, senc((sid, cnt, nonce, key))}); \\
 & \quad \text{in}(ch\_server, \langle x_{pid}, x_{sid}, x_{tpr} \rangle : (\text{pid, sid, key})); \\
 & \quad \text{let } \langle x_{sid}, x_{cnt}, x_{tpr} \rangle = sdec(x_{enc}, x_k) \text{ in} \\
 & \quad \text{lock}(used); \\
 & \quad \text{if } x_{cnt} \notin used \text{ then} \\
 & \quad \text{update}(x_{cnt} \in used); \\
 & \quad \text{event } yk\_login(x_{cnt});
\end{align*}\]

Here we find an injective agreement between the events \(yk\_press\) and \(yk\_login\). Although this example shows only one Yubikey and Server pair, it can be extended by including multiple copies of the client and server processes, and copies of the respective sets and events to prove the injective agreement with a finite number of participants.

**IX. Conclusions and Related Work**

The Set-\(\pi\) calculus and its set-based abstraction method provide an important step to overcome a serious limitation in current automated protocol verification: the limited support verifying protocols that use state. When a change in state can lead to non-monotonicity (things that were possible before the change are not possible after it) then the standard abstraction and resolution approach leads to false positives. Our solution is to enter just the “right amount” of state information into the abstraction of messages: enough to represent the non-monotonic aspects we want to model, but only so much that we do not destroy the benefit of the stateless abstraction approach in the first place. This work has been inspired by several works that go in a similar direction and we discuss here how they relate to us.

The closest works are two articles that similar to this work add state information into abstraction-based approaches. The AIF framework \([21]\) first presented the idea of encoding set memberships into the state abstraction. AIF is based on the low-level AVISPA Intermediate Format \([24]\) and thus does not have the declarative of a process calculus. For instance, one has to explicitly specify the attacker and cannot derive it from the calculus. Further, AIF uses the “raw” set membership abstraction, while in our abstraction approach we do integrate the context in which messages have been created which gives a finer abstraction. Also Set-\(\pi\) uses locks on sets, while AIF does not have this notion (and the lock exists only as per the scope of each AIF transition rule).

The second similar stateful abstraction approach is StatVerif \([2]\) which also provides an extension of the applied \(\pi\) calculus. While we use state information in the abstraction of messages, StatVerif encodes state information as an additional argument in the generated predicates of the Horn clauses. The state transition that this approach supports are in some sense like “breaking glass”: we can make at some point a global change which cannot be reverted (to avoid cycles in the state transition graph). We believe that Set-\(\pi\) and StatVerif have some complementary strengths as there are examples that cannot be directly expressed in the other. While StatVerif can express that a set of messages makes a state transition at the same time, our abstraction looses this relation between messages. On the other hand, we can flexibly have messages change their set membership independent of each other, and they can return to any previous state. An argument for the expressiveness of Set-\(\pi\) is that we have a systematic way to formalize agreement properties using sets.

There are several model-checking approaches that can deal with stateful protocols, namely the AVISPA/AVANTSSAR platform \([3]\). Note that here one needs to bound the number of
steps of honest agents which is often fine for finding attacks, but gives limited guarantees for verification. In fact, [15] studies APIs of key tokens using SATMC [4] of AVISPA, and considers abstractions of data similar to our set abstraction. This can in some cases lead to finitely many reachable abstracted states so the analysis despite depth bound is complete. The AVANTSSAR platform also includes the novel specification language ASLan that besides sets also supports the formulation of Horn-clause policies that are freshly evaluated in every state. For certain fragments we can obtain effective model-checking approaches, but again at the price of bounding the number of steps of honest agents [22].

Another verification approach that supports the verification of stateful protocols is the Tamarin prover [20]. Instead of abstraction techniques, it uses backward search and lemmata that allow to cut of search to cope with the infinite state spaces. Even for quite simple stateful protocols, such lemmata have to be supplied by the user to achieve termination. This is demonstrated by the work of [18] that presents another extension of the applied π-calculus where processes can manipulate a global key map and defines a suitable encoding into Tamarin rules. The benefit of Tamarin and related tools is a great amount of flexibility in formalising relationships between data that cannot be captured by a particular abstraction and resolution approach. However it comes at the price of loosing automation, i.e., that the user has to supply insight into the problem by proving auxiliary lemmata.

Other works have proposed a type-based approach to the verification of stateful protocols. Bugliesi et al. [13] propose a type system with resource-aware authorisation policies based on a variant of affine logic with replication. The type system is constructed on a variant of the Applied π-calculus, and can express access revocation to resources once they are consumed. Swamy et al. [23] propose a variant of ML with value-dependent types called F* for proving security properties in stateful protocol implementations. It has been used to prove the security of a full implementation of the TLS protocol [5], and can encode the type system of [13]. Like in the case of Tamarin, these approaches are very expressive but not fully automated and often require the user to supply additional lemmata. The most interesting current developments are thus to identify fragments of the problem that can be covered for instance using SMT solvers.

We believe that there is potential for further refining our analysis, in particular it seems possible to integrate ideas from the StatVerif approach, possibly even from the Tamarin-based approach into the set-based abstraction to further enlarge the class of protocols we can support.

ACKNOWLEDGMENTS

The work presented in this paper was partially supported by the EU ARTEMIS Project no. 295354 “SESAMO” (sesamo-project.eu) and by the EU FP7 Projects no. 318424, “FutureID: Shaping the Future of Electronic Identity” (futureid.eu). The authors would like to thank Roberto Vigo for valuable discussion and the anonymous reviewers for helpful comments.

REFERENCES