2015 IEEE 28th Computer Security Foundations Symposium

Set-π: Set Membership π-calculus

Alessandro Bruni, Sebastian Mödersheim, Flemming Nielson, Hanne Riis Nielson
DTU Compute, Technical University of Denmark
Emails: {albr,samo,fnie,hrni}@dtu.dk

***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or the security of communication can depend on whether a particular key is registered to a server or it has been revoked.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-π, an extension of the Applied π-calculus that includes primitives for handling databases of objects, and propose a translation from Set-π 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.

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], [24], [15], [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:

$$A \to B \ : \ \{Msg, Counter\}_{Key}$$

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* [18] 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 π 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 [16] 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".

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-π calculus that extends the popular applied π calculus [1] 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 π, 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-π, that ask for attacker-derivable messages, their set membership status, and boolean combinations thereof. A specification is secure iff no query is satisfied in any reachable state. Note that we do not specify a particular attacker, but more generally prove that the protocol is secure in the presence of an *arbitrary* attacker A that can be specified as a Set-π calculus (without access to restricted names and sets).

DOI 10.1109/CSF.2015.20

The second contribution is a stateful abstraction for Set-π. 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-π 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-π specification is secure for the given queries and against an arbitrary Set-π-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-π. 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-π. Section VI presents our translation of Set-π 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 $\mathbb{S} = \{\mathsf{s_1}, \ldots, \mathsf{s_n}\}$ the sets declared in a specific Set-π 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

$$
\begin{aligned}
M, N &::= x \mid a^l[M_1, \ldots, M_n] \mid f(M_1, \ldots, M_n) \\
P, Q &::= 0 \mid !^k P \mid P_1 \,|\, P_2 \\
&\quad \mid \mathsf{out}(M, N);\ P \mid \mathsf{in}(M, x : T);\ P \mid \mathsf{new}^l\ x : a;\ P \\
&\quad \mid \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \\
&\quad \mid \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathsf{update}(b^+);\ P \\
&\quad \mid \mathsf{lock}(L);\ P \mid \mathsf{unlock}(L);\ P \\
b &::= b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid M \in s \\
b^+ &::= b_1^+; b_2^+ \mid M \in s \mid M \notin s \\
Sys &::= \mathsf{new}\ s : \mathsf{set}\ T;\ Sys \\
&\quad \mid \mathsf{reduc}\ \forall \vec{x} : \vec{T}\ .\ g(M_1, \ldots, M_n) \rightarrow M;\ Sys \,|\, P
\end{aligned}
$$

Fig. 1: The process calculus

primitives. For example:

$$\mathsf{reduc}\ \forall x_m : t_m, x_k : key\ .\ dec(enc(x_m, x_k), x_k) \rightarrow x_m;$$

models symmetric key encryption: for every message x_m, key x_k, if a process knows an encrypted message $enc(x_m, x_k)$ and the key x_k then it can obtain the message x_m. For any rewrite rule of the form $\mathsf{reduc}\ \forall \vec{x} : \vec{T}\ .\ g(M_1, \ldots, M_n) \rightarrow M;$ we require that $fv(M) \subseteq fv(M_1, \ldots, M_n) \subseteq \{\vec{x}\}$.

Processes P, Q are: the stuck process 0, replication – which is marked by a label k, parallel composition of two processes, output, typed input, restriction – marked by a label l – 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.

The distinguishing feature of our calculus is the ability to track values in databases: the membership test ($\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q$) allows us to check a membership condition b, while a set transition ($\mathsf{update}(b^+);\ P$) inserts and removes terms from sets, according to b^+. Finally we use locks on sets to ensure linearity of set transitions: the construct $\mathsf{lock}(L)$ prevents all other processes to modify sets in L in the continuation, while $\mathsf{unlock}(L)$ 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 $\mathsf{new}\ s : \mathsf{set}\ T$, 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 $mktpl_n(M_1, \ldots, M_n)$ and n destructors $\mathsf{reduc}\ \forall \vec{x} : \vec{T}\ .\ proj_n^i(mktpl_n(x_1, \ldots, x_n)) \rightarrow x_i$;
- pattern matching on tuples for let bindings and inputs, which can be encoded using multiple let bindings with the destructors $proj_n^i$ and equality tests;
- $!^{\{s_1, \ldots, s_n\}} P$ 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 $M \notin s$ the expression $\neg(M \in s)$, and omitted else branches where not needed.

b) 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 $hmac(msg(c), k)$ such a signature. Bob receives the message and checks whether the counter c is already in the set $received$; if not, it accepts the message.

$$
\begin{aligned}
A \triangleq\ & \mathsf{new}^1\ c : cnt; \\
& \mathsf{let}\ m = msg(c)\ \mathsf{in} \\
& \mathsf{event}\ send(m); \\
& \mathsf{out}(ch, \langle m, hmac(m,k)\rangle);\ 0
\end{aligned}
$$

$$
\begin{aligned}
B \triangleq\ & \mathsf{in}(ch, \langle x_m, x_s\rangle : \langle msg(cnt), hmac(msg(cnt), key)\rangle); \\
& \mathsf{let}\ x_c = getcnt(x_m)\ \mathsf{in} \\
& \mathsf{let}\ \_ = checksign(x_m, x_s, k)\ \mathsf{in} \\
& \quad \mathsf{if}\ x_c \notin received\ \mathsf{then} \\
& \quad\quad \mathsf{update}(x_c \in received); \\
& \quad\quad \mathsf{event}\ accept(x_m);\ 0
\end{aligned}
$$

$$
\begin{aligned}
S \triangleq\ & \mathsf{new}\ received : \mathsf{set}\ cnt; \\
& \mathsf{reduc}\ \forall x : cnt\ .\ getcnt(msg(x)) \rightarrow x; \\
& \mathsf{reduc}\ \forall x : t, k : key\ .\ checksign(x, hmac(x,k), k) \rightarrow x; \\
& \mathsf{new}^2\ k : key; \\
& (!^3\, A \mid\ !^{4\,\{received\}} B)
\end{aligned}
$$

In order to express authentication we insert two events: *send* and *accept*. These are just syntactic sugar for set operations and, as we show in Section V, they can be translated into set operations.

III. Type system

The type system presented in Figure 2 is constructed to track the membership of values in sets. We denote by Sym the set of symbols for terms, destructors and sets that occur in a process; the category of types for Sym is T^{Sym}.

Data types T are either type variables, name types or constructors over types. Name types (a) are atomic types like key or cnt, type variables (t) are used to make destructors polymorphic, and constructor types define the shape

$$
\begin{array}{rll}
T ::= & t \mid a \mid f(T_1, \ldots, T_n) & \text{algebraic data types} \\
T^{Sym} ::= & T & \text{data types} \\
& \mid (T_1, \ldots, T_n) \rightarrow T & \text{destructor type} \\
& \mid \mathsf{set}\ T & \text{set type}
\end{array}
$$

Fig. 2: Type system

of a constructor. For example the term $pk(skey^l[])$ has type $pkey \triangleq pk(skey)$, and a possible type for the constructor $enc(x_m, x_k)$ could be $enc(pair(id, pkey), key)$, if we give the type $pair(id, pkey)$ to x_m and key to x_k.

Destructor types are of the form $(T_1, \ldots, T_n) \rightarrow T$ where we require $fv(T) \subseteq fv(T_1, \ldots, T_n)$. A destructor can therefore be applied to different types of data in the process, as long as the typing judgment instantiates a ground type when it is applied. For example the destructor: $\mathsf{reduc}\ \forall x_m : t, x_k : key\ .\ dec(enc(x_m, x_k), x_k) \rightarrow x_m$; has type $(enc(t, key), key) \rightarrow t$, while the instantiation $dec(enc(x_m, x_k), x_k)$ has type $(enc(pair(id, pkey), key)) \rightarrow pair(id, pkey)$ considering the previous type assignment.

Set types specify the type of terms contained in sets. For example a set of type $\mathsf{set}\ pk(skey)$ contains public keys.

The typing rules (Figure 3) enforce the correct typing of processes. Γ is the type environment, a map from identifiers of terms, destructors and sets to their type.

The typing rules for terms check whether the environment contains the right types for variables, and build types accordingly for the constructors. The rules for systems create the type environment required for typing destructors and set operations in processes. The rule for destructors applies the substitution $\sigma = \{^{T_i}/_{x_i}\}$ to the terms $M_1, \ldots, M_n, M$ in order to obtain the type of the destructor, while the rule for sets simply adds the type to the environment.

Processes are typed recursively on their syntactic form. The rules for the stuck process, parallel, replication and output simply try to type the continuation under the same Γ. The rules for input and restriction add to Γ 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 $\mathsf{set}\ T$) and that the interested sets are locked. Finally the rule for let infers the type of the result of the destructor application given the types of $M_1, \ldots, M_n$, by finding a type substitution that allows typing all arguments of the destructor and by applying such substitution to the result type.

We allow destructor definitions to contain type variables, while we require processes and sets to have only terms of ground types. In the Section IV we introduce a formal semantics for the language, together with the necessary subject reduction results for the type system.

IV. Semantics

We define in Figure 4 an instrumented operational semantics for the language. We have transitions of the form $\rho, S, \mathcal{P} \rightarrow \rho', S'\mathcal{P}'$ where:

Terms:

$$\frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \frac{}{\Gamma \vdash a[M_1, \ldots, M_n] : a} \qquad \frac{\Gamma \vdash M_1 : T_1 \quad \cdots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash f(M_1, \ldots, M_n) : f(T_1, \ldots, T_n)}$$

Conditions:

$$\frac{L, \Gamma \vdash M : T}{L, \Gamma \vdash M \in s}\ \Gamma(s) = \mathsf{set}\ T, s \in L \qquad \frac{L, \Gamma \vdash b_1 \quad L, \Gamma \vdash b_2}{L, \Gamma \vdash b_1 \wedge b_2} \qquad \frac{L, \Gamma \vdash b_1 \quad L, \Gamma \vdash b_2}{L, \Gamma \vdash b_1 \vee b_2} \qquad \frac{L, \Gamma \vdash b_1^+ \quad L, \Gamma \vdash b_2^+}{L, \Gamma \vdash b_1^+;\ b_2^+} \qquad \frac{L, \Gamma \vdash b}{L, \Gamma \vdash \neg b}$$

Systems:

$$\frac{\Gamma[g \mapsto (\sigma M_1, \ldots, \sigma M_n) \to \sigma M] \vdash Sys}{\Gamma \vdash \mathsf{reduce}\ \forall \vec{x} : \vec{T}\ .\ g(M_1, \ldots, M_n)\ \to\ M;\ Sys}\ \sigma = \{^{T_i}/_{x_i}\} \qquad \frac{\Gamma[s \mapsto \mathsf{set}\ T] \vdash Sys}{\Gamma \vdash \mathsf{new}\ s : \mathsf{set}\ T;\ Sys} \qquad \frac{\emptyset, \Gamma \vdash P}{\Gamma \vdash P}$$

Processes:

$$\frac{}{\emptyset, \Gamma \vdash 0} \qquad \frac{\emptyset, \Gamma \vdash P_1 \quad \emptyset, \Gamma \vdash P_2}{\emptyset, \Gamma \vdash P_1 \,|\, P_2} \qquad \frac{\emptyset, \Gamma \vdash P}{\emptyset, \Gamma \vdash !^l\ P} \qquad \frac{L, \Gamma \vdash P}{L, \Gamma \vdash \mathsf{out}(M, N);\ P} \qquad \frac{L, \Gamma[x \mapsto T] \vdash P}{L, \Gamma \vdash \mathsf{in}(M, x : T);\ P}$$

$$\frac{L, \Gamma[x \mapsto a] \vdash P}{L, \Gamma \vdash \mathsf{new}^l\ x : a;\ P} \qquad \frac{L, \Gamma \vdash b \quad L, \Gamma \vdash P \quad L, \Gamma \vdash Q}{L, \Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q} \qquad \frac{L, \Gamma \vdash b \quad L, \Gamma \vdash P}{L, \Gamma \vdash \mathsf{update}(b^+);\ P} \qquad \frac{L \cup L', \Gamma \vdash P}{L, \Gamma \vdash \mathsf{lock}(L');\ P}\ L' \cap L = \emptyset$$

$$\frac{\Gamma \vdash M_i : \sigma T_i \quad L, \Gamma[x \mapsto \sigma T] \vdash P \quad L, \Gamma \vdash Q}{L, \Gamma \vdash \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q}\ \Gamma(g) = (T_1, \ldots, T_n) \to T \qquad \frac{L \setminus L', \Gamma \vdash P}{L, \Gamma \vdash \mathsf{unlock}(L');\ P}\ L' \subseteq L$$

Fig. 3: Typing rules for terms, rewrite rules, processes and boolean expressions.

- $\rho : \mathsf{Var} \to \mathbb{M}$, an environment mapping process variables $(x, y, \ldots)$ to name instances $(a^l[\ldots])$,
- $S \subseteq \mathbb{S} \times \mathbb{M}$ records the set-membership states,
- $\mathcal{P} \subseteq \mathbb{P} \times \wp(\mathbb{S}) \times \wp(\mathbb{M})$ 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 $\rho, \mathcal{S}, \mathcal{P}$ represents the parallel execution of all processes in $\mathcal{P}$:

$$|_{(P_i, L_i, V_i) \in \mathcal{P}} \rho(P_i)$$

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

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 NIL removes the process 0 when it holds no locks. The rule 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 pt_x^V. The purpose of pt_x^V 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 PAR splits the process into two parallel processes. Replication REPL is annotated with $k \in \mathbb{N}$ and produces a fresh copy of P, adding x_k to V and the substitution $\{^k/_{x_k}\}$ to the environment ρ; the replication process is annotated with the index $k + 1$ after the transition.

The rule for restriction NEW maps x to $a^l[V]$ in the continuation of the process, where l is a unique label for the process $\mathsf{new}^l\ x : a;\ P$. Here we extend the terms of Figure 1 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 $\to_\rho$ as follows. Let s be a term that has only variables of atomic types and such that $\rho(s)$ is ground. Then $s \to_\rho t$ holds iff for some reduction rule $\mathsf{reduce}\ \forall \vec{x} : \vec{T}\ .\ l\ \to\ r$, there is a σ such that:

- σ is the most general unifier of l and s; w.l.o.g. we can assume that $fv(Img(\sigma)) \subseteq Dom(\rho)$;
- $t = \sigma(r)$. (Note that $\rho(t)$ is ground.)

Otherwise (if no such σ exists), we write $s \not\to_\rho$.

The rules for $\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ execute P_1 in case the set-membership state S satisfies the boolean formula b, and P_2 otherwise. The rule for update updates the current state according to the expression b^+. Finally, lock and unlock respectively acquire and release the locks on the sets in L' for the current process.

Having presented the semantic for Set-π, we need to prove that our typing judgments are preserved over the transition relation. Hence we introduce Lemma 1 to then prove subject reduction (Theorem 1).

We define the mapping $\widehat{a^l[V]} = a$ that recovers the type from an instrumented name, and its extension to environments $\widehat{\rho}(x) = \widehat{\rho(x)}$.

$$\rho, S, \mathcal{P} \uplus \{(0, \emptyset, V)\} \to \rho, S, \mathcal{P} \quad \text{NIL}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{in}(M, x:T);\ P_1, L_1, V_1), (\mathsf{out}(M, N);\ P_2, L_2, V_2)\} \to \quad \text{COM}$$
$$mgu(N', N) \circ \rho, S, \mathcal{P} \uplus \{(P_1\{^{N'}/_x\}, L_1, N' :: V_1), (P_2, L_2, V_2)\} \quad \text{where } \Gamma \vdash N : T \text{ and } N' = pt_x^{ri(V_1)}(T)$$

$$\rho, S, \mathcal{P} \uplus \{(P_1 \mid P_2, \emptyset, V)\} \to \rho, S, \mathcal{P} \uplus \{(P_1, \emptyset, V), (P_2, \emptyset, V)\} \quad \text{PAR}$$

$$\rho, S, \mathcal{P} \uplus \{(!^k\, P, \emptyset, V)\} \to \rho\{^k/_{x_k}\}, S, \mathcal{P} \uplus \{(P, \emptyset, x_k :: V), (!^{k+1}\, P, \emptyset, V)\} \quad \text{REPL}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{new}^l\ x : a;\ P, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P\{^{a^l[V]}/_x\}, L, V)\} \quad \text{NEW}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P_1\{^M/_x\}, L, V)\} \text{ if } g(M_1, \ldots, M_n) \to_\rho M \quad \text{LET1}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P_2, L, V)\} \text{ if } g(M_1, \ldots, M_n) \not\to_\rho \quad \text{LET2}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P_1, L, V)\} \quad \text{if } \rho, S \models b \quad \text{IF1}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P_2, L, V)\} \quad \text{if } \rho, S \not\models b \quad \text{IF2}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{update}(b^+);\ P, L, V)\} \to \rho, update(S, \rho(b^+)), \mathcal{P} \uplus \{(P, L, V)\} \quad \text{SET}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{lock}(L');\ P, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P, L \cup L', V)\} \quad \text{if } \forall\ (P'', L'', V'') \in \mathcal{P}\ .\ L' \cap L'' = \emptyset \quad \text{LCK}$$

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{unlock}(L');\ P, L, V)\} \to \rho, S, \mathcal{P} \uplus \{(P, L \setminus L', V)\} \quad \text{if } L' \subseteq L \quad \text{ULCK}$$

$\rho, S \models b_1 \wedge b_2$ iff $\rho, S \models b_1$ and $\rho, S \models b_2$

$\rho, S \models b_1 \vee b_2$ iff $\rho, S \models b_1$ or $\rho, S \models b_2$

$\rho, S \models \neg b$ iff $\rho, S \not\models b$

$\rho, S \models M \in s_i$ iff $\rho(M) \in S(s_i)$

$pt_x^V(a) = x_{a,V}$

$pt_x^V(f(T_1, \ldots, T_n)) = f(pt_x^{1::V}(T_1), \ldots, pt_x^{n::V}(T_n))$

$ri(V)$ denotes the set of variables x_k in V produced by replication.

$update(S, M \in s) = S \cup \{(s, M)\}$

$update(S, M \notin s) = S \setminus \{(s, M)\}$

$update(S, b_1^+; b_2^+) = update(update(S, b_1^+), b_2^+)$

Fig. 4: Semantics for the process algebra

Lemma 1 (Type substitution)**.** *Let P be a process, Γ and Γ' two type environments, M a term and T a type. If $x \notin Dom(\Gamma')$ and $L, \Gamma[x \mapsto T]\Gamma' \vdash P$ and $\Gamma\Gamma' \vdash M : T$ then $L, \Gamma\Gamma' \vdash P\{^M/_x\}$.*

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:

$$\frac{(\Gamma[x \mapsto T]\Gamma')(x) = T}{\Gamma[x \mapsto T]\Gamma' \vdash x : T}$$

is applied for the proof of $L, \Gamma[x \mapsto T]\Gamma' \vdash P$. Since $x\{^M/_x\} = M$, our statement directly follows from the hypothesis. □

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

Proof. By induction on the shape of P. □

Theorem 1 (Subject reduction)**.** *Let Γ be a type environment, $\rho, S, \mathcal{P}$ a configuration. If for all $(P, L, V) \in \mathcal{P}$ we have $L, \Gamma[\widehat{\rho}] \vdash P$, and if $\rho, S, \mathcal{P} \to \rho', S', \mathcal{P}'$, then for all $(P', L', V') \in \mathcal{P}'$ we have $L', \Gamma[\widehat{\rho'}] \vdash P'$.*

Proof sketch. The statement can be proven by a case-by-case analysis of the semantic step $\rho, S, \mathcal{P} \to \rho', S', \mathcal{P}'$. □

Subject reduction enforces that well-typed processes remain well-typed over transitions, and in particular that values of the right type are inserted and removed from sets, and that if and update only occur when the concerned sets are locked.

c) 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_{ch} with the honest protocol, and cannot perform set operations (*if, update, lock *and* unlock *constructs are excluded).*

V. Authentication in Set-π

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 [18]. 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-π 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 formulation. To simplify matters, it is common to require that the authenticated message M includes something fresh, i.e., a unique identifier that the sender chooses [23], [9]. Thanks to this construction, the same e_1 event cannot occur more than once. Then, the injective agreement goal boils down to checking that no e_2 event occurs twice (and that non-injective agreement holds).

Let us thus extend Set-π with event declarations:

$$Sys ::= \ldots \mid \mathsf{new}\ e : \mathsf{event}(T);\ Sys$$

and event processes:

$$P, Q ::= \ldots \mid \mathsf{event}\ e(M);\ P$$

and introduce the semantic rule (EVT):

$$\rho, S, \mathcal{P} \uplus \{(\mathsf{event}\ e(M);\ P, L, V)\} \overset{e(M)}{\rightarrow} \rho, S, \mathcal{P} \uplus \{(P, L, V)\}$$

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-π 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* $\rho_i, S_i, \mathcal{P}_i \overset{e_2(M)}{\rightarrow} \rho_{i+1}, S_{i+1}, \mathcal{P}_{i+1}$ *occurs in the trace, then also* $\rho_j, S_j, \mathcal{P}_j \overset{e_1(M)}{\rightarrow} \rho_{j+1}, S_{j+1}, \mathcal{P}_{j+1}$ *occurs, for* $j < i$.

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:

$$\begin{array}{lcl} \mathsf{new}\ e : \mathsf{event}(T);\ Sys & \rightarrow & \mathsf{new}\ e : \mathsf{set}\ T;\ Sys \\ \mathsf{event}\ e(M);\ P & \rightarrow & \mathsf{lock}(e);\ \mathsf{update}(M \in e); \\ & & \mathsf{unlock}(e);\ P \end{array}$$

Every event declaration becomes a set declaration in the translated process (assuming that the names for sets and events are disjoint). Whenever an event $e(M);\ P$ 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 $\mathsf{update}(b^+);\ \mathsf{event}\ e(M);\ P$, we transform it into the process $\mathsf{lock}(e);\ \mathsf{update}(b^+; M \in e);\ \mathsf{unlock}(e);\ P$.

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 $\mathsf{update}(b^+)$ and event $e(M)$, then there is also a trace where event $e'(M)$ happens before the set operation. Given a process P we denote its event-free encoding as $agree(P)$.

Theorem 2. *Let P be an extended process with events. If there is no reachable state S from $P' = agree(P)$ that satisfies the expression $M \in e_2 \wedge M \notin e_1$, then there is an 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. □

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* $\rho_i, S_i, \mathcal{P}_i \overset{e_2(M)}{\rightarrow} \rho_{i+1}, S_{i+1}, \mathcal{P}_{i+1}$ *occurs in the trace, then also* $\rho_j, S_j, \mathcal{P}_j \overset{e_1(M)}{\rightarrow} \rho_{j+1}, S_{j+1}, \mathcal{P}_{j+1}$ *occurs, for some $j < i$; furthermore, there does not exists $k > i$ such that* $\rho_k, S_k, \mathcal{P}_k \overset{e_2(M)}{\rightarrow} \rho_{k+1}, S_{k+1}, \mathcal{P}_{k+1}$.

For proving injective agreement properties the transformation becomes:

$$\begin{array}{lcl} \mathsf{new}\ e : \mathsf{event}(T);\ Sys & \rightarrow & \mathsf{new}\ e : \mathsf{set}\ T; \\ & & \mathsf{new}\ twice\text{-}e : \mathsf{set}\ T;\ Sys \\ \mathsf{event}\ e(M);\ P & \rightarrow & \mathsf{lock}(e, twice\text{-}e); \\ & & \mathsf{if}\ M \notin e\ \mathsf{then} \\ & & \quad \mathsf{update}(M \in e); \\ & & \quad \mathsf{unlock}(e, twice\text{-}e);\ P \\ & & \mathsf{else} \\ & & \quad \mathsf{update}(M \in twice\text{-}e); \\ & & \quad \mathsf{unlock}(e, twice\text{-}e);\ P \end{array}$$

Every event declaration becomes a pair of set declarations for e and $twice$-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 $twice$-e, and performs $\mathsf{update}(M \in e);\ P$ when M has not yet been inserted in e; when it is already present in e it performs $\mathsf{update}(M \in twice\text{-}e);\ P$, and in both cases unlocks e and $twice$-e; finally we add set declarations for e and $twice$-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 $inj\text{-}agree(P)$.

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

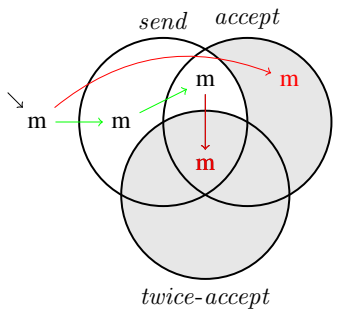

Fig. 5: State transitions of messages in the CANAuth example

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. □

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.

VI. Translation

The translation takes a process in Set-π 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 $\mathsf{msg}(M, N)$ to denote that the system has produced an output of N on channel M, predicates of the form $\mathsf{att}(M)$ to denote that the attacker process knows M, predicates of the form $\mathsf{name}(a)$ to denote that a new name is produced by the protocol, and clauses that conclude $\mathsf{transfer}(\cdot,\cdot)$ 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:

$$\mathsf{in}(ch, x : a);\ \mathsf{in}(ch, y : b);\ \mathsf{out}(ch, f(x, y))$$

produces the clause:

$$\mathsf{msg}(ch, x) \wedge \mathsf{msg}(ch, y) \Rightarrow \mathsf{msg}(ch, f(x, y))$$

Names and variables in the predicates are annotated with a special constructor *val* that defines their current membership class. For example if we have three sets in our system $\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3$, the term $val(a, 1, 0, x_{\mathsf{s}_3,a})$ 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 $x_{\mathsf{s}_3,a}$. By doing so two clauses can be unified only if the terms are in consistent states. For example $val(a, 1, 0, x_{\mathsf{s}_3,a})$ unifies with $val(a, x_{\mathsf{s}_1,a}, 0, 1)$ but not with

$$\langle\!|H_1 \wedge \cdots \wedge H_n \Rightarrow C|\!\rangle_\alpha = \langle\!|H_1|\!\rangle_\alpha \wedge \cdots \wedge \langle\!|H_n|\!\rangle_\alpha \Rightarrow \langle\!|C|\!\rangle_\alpha$$
$$\langle\!|\mathsf{p}(M_1, \ldots, M_n)|\!\rangle_\alpha = \mathsf{p}(\langle\!|M_1|\!\rangle_\alpha, \ldots, \langle\!|M_n|\!\rangle_\alpha)$$
$$\langle\!|f(M_1, \ldots, M_n)|\!\rangle_\alpha = f(\langle\!|M_1|\!\rangle_\alpha, \ldots, \langle\!|M_n|\!\rangle_\alpha)$$
$$\langle\!|a^l[V]|\!\rangle_\alpha = \begin{cases} val(a^l[V], \alpha(\mathsf{s}_1, a^l[V]), \ldots, \alpha(\mathsf{s}_n, a^l[V])) \\ \qquad\qquad\qquad\qquad \text{if } l \in labels(P_0) \\ val(a^\top[], \alpha(\mathsf{s}_1, a^l[V]), \ldots, \alpha(\mathsf{s}_n, a^l[V])) \quad \text{otrw} \end{cases}$$
$$\langle\!|x|\!\rangle_\alpha = val(x, \alpha(\mathsf{s}_1, x), \ldots, \alpha(\mathsf{s}_n, x))$$

Fig. 6: Applying the set-abstraction

$$restrict(\alpha, M \in s) = \text{if } \alpha(s, M) \neq 0 \text{ then } \{\alpha'\} \text{ else } \emptyset$$
$$\text{where } \alpha'(s', M') = \begin{cases} 1 & \text{if } M' = M \wedge s' = s \\ \alpha(s', M') & \text{otherwise} \end{cases}$$
$$restrict(\alpha, M \notin s) = \text{if } \alpha(s, M) \neq 1 \text{ then } \{\alpha'\} \text{ else } \emptyset$$
$$\text{where } \alpha'(s', M') = \begin{cases} 0 & \text{if } M' = M \wedge s' = s \\ \alpha(s', M') & \text{otherwise} \end{cases}$$
$$restrict(\alpha, b_1 \wedge b_2) = \bigcup \{restrict(\alpha', b_2) \mid \alpha' \in restrict(\alpha, b_1)\}$$
$$restrict(\alpha, b_1 \vee b_2) = restrict(\alpha, b_1) \cup restrict(\alpha, b_2)$$
$$restrict(\alpha, \neg(b_1 \wedge b_2)) = restrict(\alpha, (\neg b_1) \vee (\neg b_2))$$
$$restrict(\alpha, \neg(b_1 \vee b_2)) = restrict(\alpha, (\neg b_1) \wedge (\neg b_2))$$
$$restrict(\alpha, \neg\neg b) = restrict(\alpha, b)$$
$$zero(\alpha, a) = \alpha'$$
$$\text{where } \alpha'(s, M) = \begin{cases} 0 & \text{if } a \text{ occurs in } M \\ \alpha(s, M) & \text{otherwise} \end{cases}$$
$$relax(\alpha, L) = \alpha'$$
$$\text{where } \alpha'(s, M) = \begin{cases} \alpha(s, M) & \text{if } s \in L \\ x_{s,M} & \text{otherwise} \end{cases}$$

Fig. 7: Functions for updating α

$val(a, x_{\mathsf{s}_1,a}, 1, 1)$, 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 α, 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:

$$\alpha : (\mathbb{S} \times \mathbb{M}) \to (\{0, 1\} \cup \{x_{s,M} \mid s \in \mathbb{S}, M \in \mathbb{M}\})$$

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 ,

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 $\langle\!|p|\!\rangle_\alpha$ of Figure 6 recursively applies the set-abstraction to the clauses. When it encounters an annotated name $a^l[V]$ in the protocol, it produces a constructor $val(\dots)$ where the first parameter is the name itself—with no annotations in case of attacker names—and the remaining parameters represent the membership of $a^l[V]$ to the sets $s_1, \dots, s_n$; 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 $\mathsf{set}\ T$) to contain only name types and monadic constructors over name types. For example, $\mathsf{set}\ Seed$, $\mathsf{set}\ pk(Seed)$ and $\mathsf{set}\ sk(Seed)$ are acceptable set types, while $key(Seed, Nonce)$ is not a monadic constructor, hence $\mathsf{set}\ key(Seed, Nonce)$ is not an acceptable set type.

The function $[\![P]\!]HVL\alpha$ 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 α, and produces a set of clauses representing the protocol behaviour.

Lastly the set of functions *restrict*, *zero* and *relax* modify the set-abstraction for various constructs of Set-π. The function *restrict* takes a set-abstraction α and a boolean formula b and produces the set of all consistent abstractions that satisfy b, while *zero* inserts the constant 0 for fresh names, and *relax* introduces variables in the image of α for unlocked sets.

Having introduced the auxiliary functions for manipulating the set-abstraction, we now come back to explaining the translation process. The function $relax(\alpha, L)$ is applied at each step of the translation, as it inserts variables in the image of α 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 $!^l P$ translates P with the introduction of a new session variable x_l in the list of influencing variables V. Parallel composition $P_1 \mid P_2$ is translated as the union of the clauses generated by both processes.

Input $\mathsf{in}(M, x : T)$ adds the predicate $\mathsf{msg}(M, N')$ 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 $pt_x^{ri(V)}(T)$; the substitution $\{^{N'}/_x\}$ is then applied on the continuation. Output $\mathsf{out}(M, N)$ produces a clause with head $\langle\!|\mathsf{msg}(M, N)|\!\rangle_\alpha$ and with hypotheses $\langle\!|H|\!\rangle_\alpha$. The rule for $\mathsf{new}^l\ x : a$ introduces a restricted name: the value class of $a^l[V]$ is set to 0 for every set, the predicate $\mathsf{name}(a^l[V])$ 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 $\mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2$ looks for a substitution σ that successfully unifies a definition of the rewrite rule for the destructor g with the actual parameters $M_1, \dots, M_n$, and then finds a substitution θ that unifies the terms in the set-abstraction α 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 α', with no restriction, which is a standard over-approximation found in similar works, e.g. [7].

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

d) 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:

$$\mathsf{msg}(x_{ch}, x_{msg}) \wedge \mathsf{att}(x_{ch}) \Rightarrow \mathsf{att}(x_{msg})$$

The attacker can insert known messages into channels:

$$\mathsf{att}(x_{ch}) \wedge \mathsf{att}(x_{msg}) \Rightarrow \mathsf{msg}(x_{ch}, x_{msg})$$

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

$$\mathsf{att}(x_1) \wedge \dots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(f(x_1, \dots, x_n))$$

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

$$\mathsf{att}(M_1) \wedge \dots \wedge \mathsf{att}(M_n) \Rightarrow \mathsf{att}(M)$$

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

$$\Rightarrow \langle\!|\mathsf{att}(a^\top[])|\!\rangle_{S_0}; \quad \Rightarrow \langle\!|\mathsf{name}(a^\top[])|\!\rangle_{S_0}$$

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

$$\Rightarrow \langle\!|\mathsf{att}(ch^{l_0}[])|\!\rangle_{S_0}$$

e) Clauses representing the set-transitions: We now introduce clauses that express the meaning of the $\mathsf{transfer}(\cdot,\cdot)$ predicate: roughly speaking, when the fixedpoint contains $\mathsf{transfer}(M, M')$, 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 $[\![P_0]\!]\emptyset\emptyset\emptyset\alpha_0$. Let M_a, M'_a be two terms of type a, and consider every clause $H \Rightarrow C[M_a] \in Cl$ for some context $C[\cdot]$ that is not a $\mathsf{transfer}$ predicate. Finally let α and α' be two set-abstractions such that $H' \Rightarrow \mathsf{transfer}(\langle\!|M'_a|\!\rangle_\alpha, \langle\!|M'_a|\!\rangle_{\alpha'}) \in Cl$, and let $\sigma = mgu(M_a, M'_a)$. For each such case we add the following Horn clause:

$$\langle\!|C[M'_a]|\!\rangle_{\alpha\,\sigma} \wedge \mathsf{transfer}(\langle\!|M'_a|\!\rangle_\alpha, \langle\!|M'_a|\!\rangle_{\alpha'})_\sigma \Rightarrow \langle\!|C[M'_a]|\!\rangle_{\alpha'\,\sigma}$$

$[\![0]\!]HVL\alpha = \emptyset$

$[\![!^l\ P]\!]HV\emptyset\alpha = [\![P]\!]H(x_l :: V)\emptyset(relax(\alpha, \emptyset))$

$[\![P_1 \mid P_2]\!]HV\emptyset\alpha = [\![P_1]\!]HV\emptyset(relax(\alpha, \emptyset)) \cup [\![P_2]\!]HV\emptyset(relax(\alpha, \emptyset))$

$[\![\mathsf{in}(M, x : T);\ P]\!]HVL\alpha = [\![P\{^{N'}/_x\}]\!](H \wedge \mathsf{msg}(M, N'))(N' :: V)L\alpha'$
where $\alpha' = relax(\alpha, L),\ N' = pt_x^{ri(V)}(T)$

$[\![\mathsf{out}(M, N);\ P]\!]HVL\alpha = [\![P]\!]HVL(relax(\alpha, L)) \cup \{(\!|H \Rightarrow \mathsf{msg}(M, N)|\!)_\alpha\}$

$[\![\mathsf{new}^l\ x : a;^l\ P]\!]HVL\alpha = [\![P\{^{a^l[V]}/_x\}]\!](H \wedge \mathsf{name}(a^l[V]))VL\alpha' \cup \{(\!|H \Rightarrow \mathsf{name}(a^l[V])|\!)_{\alpha'}\}$
where $\alpha' = zero(\alpha, a^l[V])$

$[\![\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2]\!]HVL\alpha =$
$\{[\![\sigma(P_1)]\!]\sigma(H)\sigma(V)L\alpha'' \mid \mathsf{reduc}\ \forall \vec{x'} : \vec{T'}\ .\ g(M'_1, \ldots, M'_n) \rightarrow M';$ is in the scope of let,
σ is an m.g.u. that satisfies $M_1 \doteq M'_1 \wedge \cdots \wedge M_n \doteq M'_n \wedge x \doteq M'$,
θ is an m.g.u. that satisfies $\forall s, N_1, N_2,\ \sigma(N_1) = \sigma(N_2) \Rightarrow \alpha'(s, N_1) \doteq \alpha'(s, N_2)$,
and α'' satisfies $\forall N\ .\ \alpha''(s, \sigma(N)) = \theta(\alpha'(s, N))\}$
$\cup\ [\![P_2]\!]HVL\alpha'$
where $\alpha' = relax(\alpha, L)$

$[\![\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2]\!]HVL\alpha = \{[\![P_1]\!]HVL\alpha' \mid \alpha' \in restrict(relax(\alpha, L), b)\} \cup$
$\{[\![P_2]\!]HVL\alpha' \mid \alpha' \in restrict(relax(\alpha, L), \neg b)\}$

$[\![\mathsf{lock}(L');\ P]\!]HVL\alpha = [\![P]\!]HV(L \cup L')(relax(\alpha, L))$

$[\![\mathsf{unlock}(L');\ P]\!]HVL\alpha = [\![P]\!]HV(L \setminus L')(relax(\alpha, L))$

$[\![\mathsf{update}(b^+);\ P]\!]HVL\alpha = \{(\!|H|\!)_{\alpha'} \Rightarrow \mathsf{transfer}((\!|M|\!)_{\alpha'}, (\!|M|\!)_{\alpha''}) \mid M \in fv(b^+) \cup fn(b^+)\} \cup [\![P]\!]HVL\alpha'''$
where $\alpha' = relax(\alpha, L)$ and $\alpha'' = update(\alpha, b^+)$ and $\alpha''' = relax(\alpha'', L)$

Fig. 8: Translation rules for processes into Horn clauses

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

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.

f) 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:

$P_1 \triangleq\ !^{l_1\ \{r,a,at\}\ l_2}\mathsf{in}(ch, \langle x_m, x_s\rangle :$
$\langle msg(cnt), hmac(msg(cnt), key)\rangle));^{l_3}$
$\mathsf{let}\ \_ = eq(x_s, hmac(x_m, k))\ \mathsf{in}\ ^{l_4}$
$\mathsf{let}\ x_c = getcnt(x_m)\ \mathsf{in}\ ^{l_5}$
$\mathsf{if}\ x_c \notin r\ \mathsf{then}\ ^{l_6}$
$\mathsf{if}\ x_m \notin a\ \mathsf{then}\ ^{l_7}\mathsf{update}(x_c \in r; x_m \in a);\ ^{l_8}$
$\mathsf{else}\ ^{l_9}\mathsf{update}(x_c \in r; x_m \in at);\ ^{l_{10}}$

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 α over the recursive calls of $[\![\cdot]\!]$, and P^l to denote the subprocess of P_1 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 10 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 $\mathcal{F}_{P_0}$ covers all possible behaviours of a process, when started in the given configuration (ρ, 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 α that has no variables (i.e., indetermined set memberships) and we can thus can 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 $(\!|\mathsf{msg}(M, N)|\!)_S$. This ensures that whatever behaviour is present in the semantics is also captured by the translation.

We denote by $\mathcal{C}_{P_0}$ the set of clauses produced by the translation, including the fixed clauses, and by $\mathcal{F}_{P_0}$ the set

$$
\begin{aligned}
[\![B]\!]\emptyset[]\emptyset\alpha_0 &= [\![P^{l_1}]\!](H_1 = \emptyset)(V_1 = [x_{l_1}])(L_1 = \emptyset)(\alpha_1 = \mathit{relax}(\alpha_0, \emptyset)) \\
&= [\![P^{l_2}]\!](H_2 = \emptyset)(V_2 = V_1)(L_2 = \{r, a, at\})(\alpha_2 = \mathit{relax}(\alpha_1, \emptyset)) \\
&= [\![P^{l_3}]\!](H_3 = \{\mathsf{msg}(ch[], T_3 = \langle msg(x_{cnt,1}), hmac(msg(x_{cnt,2}), x_{key,3})\rangle)\}) \\
&\qquad (V_3 = T_3 :: V_2)(L_3 = L_2)(\alpha_3 = \mathit{relax}(\alpha_2, L_2)) \\
&= [\![P^{l_4}]\!](H_4 = \{\mathsf{msg}(ch[], T_4 = \langle msg(x_{cnt,1}), hmac(msg(x_{cnt,1}), k[])\rangle)\}) \\
&\qquad (V_4 = T_4 :: V_2)(L_4 = L_3)(\alpha_4 = \theta(\mathit{relax}(\alpha_3, L_3))) \\
&= [\![P^{l_5}]\!](H_5 = H_4)(V_5 = V_4)(L_5 = L_4)(\alpha_5 = \mathit{relax}(\alpha_4, L_4)) \\
&= [\![P^{l_6}]\!](H_6 = H_5)(V_6 = V_5)(L_6 = L_5)(\alpha_6 = \mathit{restrict}(\mathit{relax}(\alpha_5, L_5), x_{cnt,1} \notin r)) \\
&= [\![P^{l_7}]\!](H_7 = H_6)(V_7 = V_6)(L_7 = L_6)(\alpha_7 = \mathit{restrict}(\mathit{relax}(\alpha_6, L_6), msg(x_{cnt,1}) \notin a)) \cup \\
&\quad [\![P^{l_{10}}]\!](H_{10} = H_6)(V_{10} = V_6)(L_{10} = L_6)(\alpha_{10} = \mathit{restrict}(\mathit{relax}(\alpha_6, L_6), msg(x_{cnt,1}) \in a)) \\
[\![P^{l_7}]\!]H_7V_7L_7\alpha_7 &= \{\mathsf{msg}(msg(val(ch[])), \langle msg(val(x_1, x_{1,s}, 0, 0, x_{1,at})), hmac(msg(val(x_1, x_{1,s}, 0, 0, x_{1,at}), val(k[]))\rangle)) \\
&\qquad \Rightarrow \mathsf{transfer}(val(x_1, x_{1,s}, 0, 0, x_{1,at}), val(x_1, x_{1,s}, 1, 1, x_{1,at}))\} \\
[\![P^{l_{10}}]\!]H_{10}V_{10}L_{10}\alpha_{10} &= \{\mathsf{msg}(msg(val(ch[])), \langle msg(val(x_1, x_{1,s}, 0, 1, x_{1,at})), hmac(msg(val(x_1, x_{1,s}, 0, 1, x_{1,at}), val(k[]))\rangle)) \\
&\qquad \Rightarrow \mathsf{transfer}(val(x_1, x_{1,s}, 0, 1, x_{1,at}), val(x_1, x_{1,s}, 1, 1, 1))\}
\end{aligned}
$$

Fig. 9: Horn clauses generated by the translation

of ground facts derivable from $\mathcal{C}_{P_0}$. First we introduce the order relation $\preceq_L$ on the facts $\mathcal{F}_{P_0}$ derivable from the initial protocol.

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

(i) $\forall s_j \in L\ .\ S_1(s_j) = S_2(s_j)$;
(ii) $\forall \mathsf{p}(M_1, \ldots, M_k)\,.\,(\!|\mathsf{p}(M_1, \ldots, M_k)|\!)_{S_1} \in \mathcal{F}_{P_0} \Rightarrow (\!|\mathsf{p}(M_1, \ldots, M_k)|\!)_{S_2} \in \mathcal{F}_{P_0}$.

Intuitively the $\preceq_L$ 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 $\preceq_\emptyset$, as it allows any set to be modified.

Next we formalise the definition for set-abstraction α that was introduced in Section VI.

Definition 5 (Set-abstraction)**.** *The mapping α abstracts S under the environment ρ 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 α abstracts a state S if every pair (s, M) that maps to a variable in α 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 $\preceq_L$. The interested reader can find the full proofs in the extended version of this article .

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

Since *relax* inserts unique variables in α' 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 α satisfies the properties of Definition 5.

Lemma 4 (*restrict* preserves the set-abstraction)**.** *Let α be a set abstraction, ρ an environment, S a state and $A = \mathit{restrict}(\alpha, b)$. If $\rho, S \models b$ and α abstracts S, then there exists an $\alpha' \in A$ such that α' abstracts S.*

Restrict produces a set of set-abstractions each representing a possible way of satisfying the formula b. Lemma 4 establishes that if α abstracts S then at least one of these restrictions on α satisfies the abstraction of S.

Lemma 5 (transfer preserves $S \preceq_L S'$)**.** *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 $\mathsf{transfer}((\!|M|\!)_S, (\!|M|\!)_{S'}) \in \mathcal{F}_{P_0}$ then for any set of locks L such that $\{s_1, \ldots, s_n\} \cap L = \emptyset$ we have $S \preceq_L S'$.*

Lemma 5 establishes that transfer predicates actually capture the state transitions, hence following the definition of the order $\preceq_L$ 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 $\rho_0 = [x_{ch} \mapsto ch^{l_0}[]]$.

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

Proof of sketch. Let B be a subprocess of A, ρ 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) $S_0 \preceq_\emptyset S$, and
(iii) for every maximal subterm M of B closed under ρ, we have $(\!|\rho(\mathsf{att}(M))|\!)_S \in \mathcal{F}_{P_0}$,

$$\frac{}{\rho, V, L, S \Vdash 0}\ \text{T-NIL} \qquad \frac{\forall S' \text{ s.t. } S \preceq_\emptyset S'\ (\rho, V, \emptyset, S' \Vdash Q_1 \wedge \rho, V, \emptyset, S' \Vdash Q_2)}{\rho, V, \emptyset, S \Vdash Q_1 \mid Q_2}\ \text{T-PAR}$$

$$\frac{\forall S' \text{ s.t. } S \preceq_\emptyset S'\ (\rho\{^l/_{x_l}\}, (x_l :: V), \emptyset, S' \Vdash Q)}{\rho, V, \emptyset, S \Vdash !^l\, Q}\ l \in \mathbb{N}\ \text{T-REPL}$$

$$\frac{\forall S' \text{ s.t. } S \preceq_L S'\ \forall N \text{ s.t. } \Gamma \vdash N : T \quad \{\!|\rho(\mathsf{msg}(M, N))|\!\}_{S'} \in \mathcal{F}_{P_0} \Rightarrow (mgu(N', N) \circ \rho), (N' :: V), L, S' \Vdash Q\{^{N'}/_x\}}{\rho, V, L, S \Vdash \mathsf{in}(M, x : T);\ Q}\ N' = pt_x^{ri(V)}(T)\ \text{T-IN}$$

$$\frac{\{\!|\rho(\mathsf{msg}(M, N))|\!\}_S \in \mathcal{F}_{P_0} \wedge \forall S' \text{ s.t. } S \preceq_L S'\ (\rho, V, L, S' \Vdash Q)}{\rho, V, L, S \Vdash \mathsf{out}(M, N);\ Q}\ \text{T-OUT}$$

$$\frac{\{\!|\rho(\mathsf{name}(a^l[V]))|\!\}_S \in \mathcal{F}_{P_0} \wedge \forall S' \text{ s.t. } S \preceq_L S'\ \rho, V, L, S' \Vdash Q\{^{a^l[V]}/_x\}}{\rho, V, L, S \Vdash \mathsf{new}^l\ x : a;\ Q}\ \text{T-NEW}$$

$$\frac{\forall S' \text{ s.t. } S \preceq_L S'\ (\forall M \text{ s.t. } g(M_1, \ldots, M_n) \to_\rho M\ \ \rho, V, L, S' \Vdash Q_1\{^M/_x\}) \wedge \rho, V, L, S' \Vdash Q_2}{\rho, V, L, S \Vdash \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ Q_1\ \mathsf{else}\ Q_2}\ \text{T-LET}$$

$$\frac{\forall S' \text{ s.t. } S \preceq_L S',\ (\rho, S' \models b \Rightarrow \rho, V, L, S' \Vdash Q_1) \wedge (\rho, S' \models \neg b \Rightarrow \rho, V, L, S' \Vdash Q_2)}{\rho, V, L, S \Vdash \mathsf{if}\ b\ \mathsf{then}\, Q_1 \mathsf{else}\ Q_2}\ \text{T-IF}$$

$$\frac{\forall S' \text{ s.t. } S \preceq_L S'\ \rho, V, (L \cup L'), S' \Vdash Q}{\rho, V, L, S \Vdash \mathsf{lock}(L');\ Q}\ \text{T-LOCK} \qquad \frac{\forall S' \text{ s.t. } S \preceq_L S'\ \rho, V, (L \setminus L'), S' \Vdash Q}{\rho, V, L, S \Vdash \mathsf{unlock}(L');\ Q}\ \text{T-UNLOCK}$$

$$\frac{\forall S' \text{ s.t. } S \preceq_L S',\quad (\forall\ M \in fv(b^+) \cup fn(b^+)\ \mathsf{transfer}(\{\!|\rho(M)|\!\}_{S'}, \{\!|\rho(M)|\!\}_{S''}) \in \mathcal{F}_{P_0}) \wedge (\forall S''' \text{ s.t. } S'' \preceq_L S''',\ \rho, V, L, S''' \Vdash Q)}{\rho, V, L, S \Vdash \mathsf{update}(b^+);\ Q}\ S'' = update(S', \rho(b^+))\ \text{T-SET}$$

Fig. 10: Inference system for correctness

then:

$$\rho, V, \emptyset, S \Vdash B$$

Proof by induction over the depth of B.

In particular, we have that (i) $fv(A) = x_{ch}$, hence $\rho_0(A)$ is closed; (ii) $S_0 \preceq_\emptyset S_0$ by reflexivity; and (iii) the only maximal subterm of A that is bound by ρ_0 is x_{ch}, and by construction of the translation $\{\!|\rho_0(\mathsf{att}(x_{ch}))|\!\}_{S_0} \in \mathcal{F}_{P_0}$. Hence the attacker process types. □

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

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 α, an environment ρ; if:

(i) $\rho(Q)$ is a closed process, $\rho(V)$ and $\rho(H)$ are ground,
(ii) α abstracts S under ρ,
(iii) $\mathcal{C}_{P_0} \supseteq [\![Q]\!]HVL\alpha$,
(iv) for every predicate p in H, we have that $\{\!|\rho(p)|\!\}_S \in \mathcal{F}_{P_0}$

Then $\rho, V, L, S \Vdash Q$.

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

In particular, (i) ρ_0 closes P_0 by construction, $\rho_0(\emptyset)$ is trivially ground, (ii) α_0 abstracts S_0 under ρ_0 by construction, (iii) $\mathcal{C}_{P_0} \supseteq [\![P_0]\!]\emptyset\emptyset\emptyset\alpha_0$ by definition of the translation, (iv) holds vacuously. Therefore the conditions (i–iv) are satisfied and hence $\rho_0, \emptyset, \emptyset, S_0 \Vdash P_0$. □

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

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

Theorem 5 (Correctness of the analysis)**.** *Let $Sys[\cdot]$ 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 $\rho_0 = [x_{ch} \mapsto ch^{l_0}[]]$.*

If the typing $[] \vdash Sys[\mathsf{new}^{l_0}\ x_{ch} : ch;\ P_0 \mid A]$ holds, and if $\rho_0, S_0, \mathcal{P}_0 = \{(P_0 \mid A, \emptyset, \emptyset)\} \to^* \rho_n, S_n, \mathcal{P}_n = \mathcal{P}_{n'} \uplus \{(\mathsf{out}(M, N);\ P', L, V)\}$; then $\{\!|\rho(\mathsf{msg}(M, N))|\!\}_{S_n} \in \mathcal{F}_{P_0}$.

Proof. By Lemma 7 we know that $\rho_0, \emptyset, \emptyset, S_0 \Vdash P_0$; by Lemma 6 we know that $\rho_0, \emptyset, \emptyset, S_0 \Vdash A$, hence all processes in $\mathcal{P}_0$ type in the initial state S_0.

Let $\rho_0, S_0, \mathcal{P}_0 \to \rho_1, S_1, \mathcal{P}_1 \to \cdots \to \rho_n, S_n, \mathcal{P}_n$. 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 $\mathsf{out}(M, N);\ P'$ types in state S_n and hence $\{\!|\rho_n(\mathsf{msg}(M, N))|\!\}_{S_n} \in \mathcal{F}_{P_0}$. □

Theorem 5 establishes the final relation between the inference system of Figure 10 and the instrumented semantics. We use this result to link the facts generated by the translation to a query of interest.

Corollary 1 (Checking queries)**.** *If* $\rho_0, S_0, \mathcal{P}_0 =$

Example	Average time	Vulnerable
CANAuth	0.0174s	no
MaCAN	0.0244s	yes
Key registration	0.0254s	no
Yubikey single	0.0194s	no

Fig. 11: Experimental results

$\{(P_0 \mid A, \emptyset, \emptyset)\} \rightarrow^* \rho_j, S_j, \mathcal{P}_j = \mathcal{P}'_j \uplus \{(\mathsf{out}(M, N); P', L, V)\} \rightarrow^* \rho_n, S_n, \mathcal{P}_n$ *and* $\rho_n, S_n \models b$ *then there exists an* $\alpha_b \in \mathit{restrict}(\alpha_0, b)$ *where* $\theta = \mathit{mgu}(S_n, \alpha_b)$ *and* $(\!|\rho_n(\mathsf{msg}(M, N))|\!)_{\theta \circ \alpha_b}$.

Proof. Follows from Theorem 5 and because $S_j \preceq_\emptyset S_n$. □

Therefore we can express any query of the form:

$$\mathsf{msg}(M, N) \text{ where } b$$

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 V.

VIII. Experimental Evaluation

We implemented our analysis into a prototype tool written in Haskell, that translates processes specified in Set-π 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 [17]. We recorded the running times for our test suite on a 2,7 GHz Intel Core i7 with 8 GB of RAM running OS X. They are comparable to similar ProVerif models in applied-π, 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

Here we present a key-registration protocol where an honest principal A registers its current pair of asymmetric keys (pk_A, sk_A) to the server S. An initial pair of keys is distributed securely to A and S, where A knows both public and secret keys while S only knows the public key.

Later in the protocol, before the current key expires, A registers a new key to the server by sending the following message:

$$A \rightarrow S : senc(sk, (new, a, pk'))$$

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.

$$S \rightarrow A : penc(pk', (confirm))$$

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

$$\begin{aligned}
A \triangleq\ & \mathsf{in}(kdb_a, sk_a : SKey);\\
& \mathsf{if}\ sk_a \in ring_a\ \mathsf{then}\\
& \mathsf{new}\ s'_a : Seed;\\
& \mathsf{update}(sk(s'_a) \in ring_a);\\
& \mathsf{out}(ch, senc(sk_a, (new_key, a, pk(s'_a))));\\
& \mathsf{out}(kdb_a, sk(s'_a));\\
& \mathsf{in}(ch, x_c : senc(PKey, x_t));\\
& \mathsf{let}\ x_r = pdec(sk(s'_a), x_c)\ \mathsf{in}\\
& \mathsf{if}\ x_r = confirm\ \mathsf{then}\\
& \mathsf{update}(sk_a \notin ring_a);\\
& \mathsf{out}(ch, sk_a);\ 0
\end{aligned}$$

$$\begin{aligned}
S \triangleq\ & \mathsf{in}(ch, x_s : senc(SKey, (x_t, x_{t'}, PKey)));\\
& \mathsf{in}(kdb_s, pk_a : PKey);\\
& \mathsf{let}\ (= new, = a, pk'_a) = pdec(pk_a, xs)\ \mathsf{in}\\
& \mathsf{if}\ pk_a \in valid_a \wedge pk'_a \notin valid_a \wedge pk'_a \notin revoked_a\ \mathsf{then}\\
& \mathsf{update}(pk_a \in revoked_a; pk_a \notin valid_a; pk'_a \in valid_a);\\
& \mathsf{out}(ch, penc(pk'_a, confirm));\\
& \mathsf{out}(kdb_s, pk'_a);\ 0
\end{aligned}$$

$$\begin{aligned}
Sys \triangleq\ & \mathsf{reduc}\ \forall x : Seed, m : t\ .\\
& \qquad pdec(pk(x), senc(sk(x), m)) \rightarrow m;\\
& \mathsf{reduc}\ \forall x : Seed, m : t\ .\\
& \qquad sdec(sk(x), penc(pk(x), m)) \rightarrow m;\\
& \mathsf{reduc}\ \forall x : Seed\ .\ keypair(sk(x), pk(x)) \rightarrow true;\\
& \mathsf{new}\ ring_a : \mathsf{set}\ sk(Seed);\\
& \mathsf{new}\ valid_a : \mathsf{set}\ pk(Seed);\\
& \mathsf{new}\ revoked_a : \mathsf{set}\ pk(Seed);\\
& \mathsf{new}\ kdb_a : SKey;\ \mathsf{new}\ kdb_s : SKey;\ \mathsf{new}\ s_a : Seed;\\
& \mathsf{update}(sk(s_a) \in ring_a; pk(s_a) \in valid_a);\\
& \mathsf{out}(kdb_a, sk(s_a));\\
& \mathsf{out}(kdb_s, pk(s_a));\\
& (!^{\{ring_a\}}\ A \mid !^{\{valid_a, revoked_a\}}\ S)
\end{aligned}$$

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.

B. 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 they 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} 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.

$$
\begin{aligned}
Sys \triangleq\ & \mathsf{new}\ yk\_press : \mathsf{event}(cnt);\\
& \mathsf{new}\ yk\_login : \mathsf{event}(cnt);\\
& \mathsf{new}\ used : \mathsf{set}\ cnt;\\
& \mathsf{reduc}\ \forall x : t, k : key\ .\ sdec(senc(x,k),k) \rightarrow x;\\
& \mathsf{new}\ ch : channel;\\
& \mathsf{new}\ ch\_server : channel;\\
& (!\,YK \mid !\,Srv)
\end{aligned}
$$

$$
\begin{aligned}
BP \triangleq\ & \mathsf{new}\ x_c : cnt;\\
& \mathsf{new}\ x_{nonce} : nonce;\\
& \mathsf{new}\ x_{tpr} : nonce;\\
& \mathsf{event}\ yk\_press(x_c);\\
& \mathsf{out}(ch, \langle x_{pid}, x_{nonce}, senc(\langle x_{sid}, x_c, x_{tpr}\rangle, k)\rangle);\ 0
\end{aligned}
$$

$$
\begin{aligned}
YK \triangleq\ & \mathsf{new}\ k : key;\\
& \mathsf{new}\ x_{pid} : pid;\\
& \mathsf{new}\ x_{sid} : sid;\\
& \mathsf{out}(ch\_server, \langle x_{pid}, x_{sid}, k\rangle);\\
& \mathsf{out}(ch, x_{pid});\\
& !\,BP
\end{aligned}
$$

$$
\begin{aligned}
Srv \triangleq\ & \mathsf{in}(ch, \langle x_{pid}, x_{nonce}, x_{enc}\rangle :\\
& \qquad \langle pid, nonce, senc(\langle sid, cnt, nonce\rangle, key)\rangle);\\
& \mathsf{in}(ch\_server, \langle = x_{pid}, x_{sid}, x_k\rangle : \langle pid, sid, key\rangle);\\
& \mathsf{let}\ \langle = x_{sid}, x_{cnt}, x_{tpr}\rangle = sdec(x_{enc}, x_k)\ \mathsf{in}\\
& \mathsf{lock}(used);\\
& \mathsf{if}\ x_{cnt} \notin used\ \mathsf{then}\\
& \quad \mathsf{update}(x_{cnt} \in used);\\
& \quad \mathsf{event}\ yk\_login(x_{cnt});\\
& \quad \mathsf{unlock}(used);\ 0
\end{aligned}
$$

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-π 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 [20] first presented the idea of encoding set memberships into the state abstraction. AIF is based on the low-level AVISPA Intermediate Format [23] and thus does not have the declarativity 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-π 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 π 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-π 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 flexibily 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-π 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, [14] 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 [21].

Another verification approach that supports the verification of stateful protocols is the Tamarin prover [19]. 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 [17] 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. [22] 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

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Symposium on Principles of Programming Languages*, 2001.

[2] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, 2014.

[3] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, Serena Elisa Ponta, Marco Rocchetto, Michaël Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS*, 2012.

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

[5] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *Security and Privacy*, 2013.

[6] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop*, 2001.

[7] Bruno Blanchet. From secrecy to authenticity in security protocols. In *Static Analysis Symposium*, 2002.

[8] Bruno Blanchet. Security protocols: from linear to classical logic by abstract interpretation. *Inf. Process. Lett.*, 95(5):473–479, 2005.

[9] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[10] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[11] Alessandro Bruni. SetPi tool page. https://bitbucket.org/setpi/setpi.

[12] Alessandro Bruni, Michal Sojka, Flemming Nielson, and Hanne Riis Nielson. Formal security analysis of the MaCAN protocol. In *Integrated Formal Methods*, 2014.

[13] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Resource-aware authorization policies for statically typed cryptographic protocols. In *Computer Security Foundations Symposium*, 2011.

[14] Sibylle B. Fröschle and Graham Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *ARSPA-WITS*, 2009.

[15] Jean Goubault-Larrecq. Towards producing formally checkable security proofs, automatically. In *Computer Security Foundations Symposium*, 2008.

[16] Anthony Van Herrewege, David Singelee, and Ingrid Verbauwhede. CANAuth-A simple, backward compatible broadcast authentication protocol for CAN bus. In *Proceedings of ECRYPT*, 2011.

[17] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *Security and Privacy*, 2014.

[18] Gavin Lowe. A hierarchy of authentication specification. In *Computer Security Foundations Workshop*, 1997.

[19] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, 2013.

[20] Sebastian Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Computer and Communications Security*, 2010.

[21] Sebastian Mödersheim. Deciding security for a fragment of aslan. In *European Symposium on Research in Computer Security*, 2012.

[22] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

[23] The AVISPA Project. AVISPA Project Deliverable 2.3: The Intermediate Format, 2003. http://www.avispa-project.org/publications.html.

[24] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Conference on Automated Deduction*, 1999.