Automatic Verification of Protocols with Lists of Unbounded Length

Bruno Blanchet
INRIA Paris-Rocquencourt, France
Bruno.Blanchet@inria.fr

Miriam Paiola
INRIA Paris-Rocquencourt, France
Miriam.Paiola@inria.fr

ABSTRACT

We present a novel automatic technique for proving secrecy and authentication properties for security protocols that manipulate lists of unbounded length, for an unbounded number of sessions. This result is achieved by extending the Horn clause approach of the automatic protocol verifier ProVerif. We extend the Horn clauses to be able to represent lists of unbounded length. We adapt the resolution algorithm to handle the new class of Horn clauses, and prove the soundness of this new algorithm. We have implemented our algorithm and successfully tested it on several protocol examples, including XML protocols coming from web services.

Categories and Subject Descriptors
C.2.2 [Computer-communication networks]: Network Protocols—Protocol verification

Keywords
Security protocols; verification; lists; web services; Horn clauses; resolution.

1. INTRODUCTION

Security protocols are protocols that rely on cryptographic primitives such as encryption and signatures for securing communication between several parties. They aim at ensuring security properties such as secrecy or authentication. However, attacks are often found against protocols that were thought correct. Furthermore, security flaws cannot be detected by testing since they appear only in the presence of an attacker. The confidence in these protocols can then be increased by a formal analysis that proves the desired security properties. To ease formal verification, one often uses the symbolic, so-called Dolev-Yao model [13], which abstracts from the details of cryptographic primitives and considers messages as terms. In this work, we also rely on this model.

The formal verification of security protocols with fixed-size data structures has been extensively studied. However, some protocols, for instance XML protocols of web services, use more complex data structures, such as lists. The verification of protocols that manipulate such data structures has been less studied and presents additional difficulties: these complex data structures add another cause of undecidability.

Like other protocol verifiers, ProVerif can analyze protocols with lists if we fix the lengths of the lists a priori. However, if the protocol is verified only for some lengths, attacks may exist for other lengths. If there is an attack against a protocol, then there is a bound such that this attack appears with lists shorter than the bound, but this bound depends on the protocol and is not easy to compute. So our goal is to extend ProVerif to the verification of protocols with lists of any length without bounding the length (and still not bounding the number of sessions). To obtain this result, we extend the language of Horn clauses, introducing a new kind of clauses, generalized Horn clauses, to be able to represent lists of any length. We adapt the resolution algorithm to deal with these new clauses, and prove the soundness of the new algorithm. The obtained algorithm can prove secrecy and authentication properties. (More precisely, the authentication property that we consider is non-injective agreement [17]: if some participant \( B \) terminates the protocol with some parameters, then \( A \) started the protocol with the same parameters.) This algorithm performs approximations, so it may fail to prove security properties that actually hold, and it does not always terminate. This is unavoidable because the problem of verifying protocols with an unbounded number of sessions is undecidable. However, the algorithm works well in practice: we have implemented it and successfully tested it on several protocol examples (see Section 6). Applications of our results include in particular the treatment of XML protocols such as web services, XML documents being modeled using possibly nested lists. In this paper, we focus mainly on these protocols. We illustrate our work on the SOAP extension [9] to XML signatures [4]. XML signatures allow one to sign several parts of an XML
1.1 Related Work

The first approach considered for proving protocols with recursive data structures was interactive theorem proving: a recursive authentication protocol was studied for an unbounded number of participants, using Isabelle/HOL [20], and using rank functions and PVS [10]. However, this approach requires considerable human effort.

Truderung [22] showed a decidability result (in NEXPTIME) for secrecy in recursive protocols, which include transformations of lists, for a bounded number of sessions. This result was extended to a class of recursive protocols with XOR [16] in 3-NEXPTIME. Chirici et al. [11, 12] present an extension of the constraint-based approach in symbolic protocol verification to handle a class of protocols (Well-Tagged protocols with Autonomous keys) with unbounded lists in messages. They prove that the insecurity problem for Well-Tagged protocols with Autonomous keys is decidable for a bounded number of sessions.

Several approaches were considered for verifying XML protocols [6, 21, 15, 3], by translating them to the input format of a standard protocol verifier: the tool TulaFale [6] uses ProVerif as back-end. Kleiner and Roscoe [21, 15] translate WS-Security protocols to FDR; Backes et al. [3] use AVISPA. All these approaches have little or no support for lists of unbounded length. For instance, TulaFale has support for list membership with unbounded lists, but does not go further.

Recently, Paiola and Blanchet [19] showed that, for a certain class of Horn clauses, if secrecy is proved by ProVerif for lists of length one, then secrecy also holds for lists of unbounded length. Their target applications are group protocols and XML protocols. However, this work is limited to secrecy, and the class of protocols that they handle is limited to protocols that treat all elements of lists uniformly. When their reduction result does not apply, a different approach is needed. We propose such an approach in this paper. We present a practical algorithm that can prove both secrecy and authentication properties of protocols that manipulate different list elements in different ways.

1.2 Outline

The next section recalls the technique used by ProVerif. In Section 3, we introduce our running example and motivate the introduction of a new type of clauses, the generalized Horn clauses, that we will use to model group protocols. In Section 4, we formally define generalized Horn clauses, and their semantics by giving their translation into Horn clauses. In Section 5, we adapt the resolution algorithm to generalized Horn clauses and prove its soundness. Section 6 summarizes our experimental results. Because of space constraints, the proofs of all our results and additional details on our algorithm can be found in the long version of the paper, available at [8], together with our implementation.

2. A REMINDER ON PROVERIF

ProVerif translates the initial protocol into a set of Horn clauses. The syntax of these clauses is defined in Figure 1.

\[ p := \text{patterns} \]
\[ \sigma[p_1, \ldots, p_n] \]
\[ F := \text{constructor application} \]
\[ R := F_1 \land \cdots \land F_n \Rightarrow F \]

\text{Horn clause}

Figure 1: Syntax of Horn clauses

The patterns represent messages that are exchanged between participants of the protocol. A variable can represent any pattern. Names represent atomic values, such as keys and nonces. Each participant can create new names. Instead of creating a fresh name at each run of the protocol, the created names are considered as functions represented by the pattern \( \sigma[p_1, \ldots, p_n] \). These functions take as arguments the messages previously received by the principal that creates the name as well as session identifiers, which are variables that take a different value at each run of the protocol, to distinguish names created in different runs. As shown in, e.g., [7], this representation of names is a sound approximation. When a name has no arguments, we write \( a \) instead of \( a[] \). We use \( a \) for a generic name and identifiers in sans serif font (e.g., sk) for fixed names.

The fact \( \text{att}(p) \) means that the attacker may have the pattern (message) \( p \). ProVerif models authentication as correspondence assertions, such as "if event \( e(x) \) has been executed, then event \( e'(x) \) has been executed". It uses the fact \( \text{m-event}(p) \) to represent that the event \( p \) must have been executed, and the fact \( \text{event}(p) \) to represent that the event \( p \) may have been executed.

A clause \( F_1 \land \cdots \land F_n \Rightarrow F \) means that, if all facts \( F_j \) are true, then the conclusion \( F \) is also true. We use \( R \) for a clause, \( H \) for its hypothesis, and \( C \) for its conclusion. The hypothesis of a clause is considered as a multiset of facts. A clause with no hypothesis \( \Rightarrow F \) is written simply \( F \).

Cryptographic primitives are represented by functions. There are two kinds of functions: constructors and destructors. A constructor \( f \) is a function that explicitly appears in the patterns that represent messages and builds new patterns of the form \( f(p_1, \ldots, p_n) \). Destructors manipulate patterns. A destructor \( g \) is defined by a set \( \text{def}(g) \) of rewrite rules of the form \( g(p_1, \ldots, p_n) \to p \) where \( p_1, \ldots, p_n \) are patterns with only variables and constructors and the variables of \( p \) appear in \( p_1, \ldots, p_n \). Using constructors and destructors, one can represent data structures and cryptographic operations. For instance, \( \text{senc}(m, k) \) is the constructor that represents the symmetric key encryption of the message \( m \) under the key \( k \). The corresponding destructor \( \text{sdenc}(m', k) \) returns the decryption of \( m' \) if \( m' \) is a message encrypted under \( k \). The rewrite rule that defines \( \text{sdenc} \) is

\[ \text{sdenc}(\text{senc}(m, k), k) \to m. \]

A protocol is represented by three sets of Horn clauses:

- initial knowledge of the attacker: we have a fact \( \text{att}(p) \) for each \( p \) initially known by the attacker.
- abilities of the attacker: \( \text{att}(a[x]) \)
for each constructor \( f \) of arity \( n \):
\[
\text{att}(x_1) \land \cdots \land \text{att}(x_n) \Rightarrow \text{att}(f(x_1, \ldots, x_n))
\]
for each destructor \( g \),
\[
\text{for each rule } g(p_1, \ldots, p_n) \rightarrow p \text{ in def } (g):
\text{att}(p_1) \land \cdots \land \text{att}(p_n) \Rightarrow \text{att}(p)
\]
The first clause represents the ability of the attacker to create fresh names: all fresh names that the attacker may create are represented by the names \( a[x] \) for any \( x \). The other clauses mean that if the attacker has some messages, then he can apply constructors and destructors to them.

- the protocol itself: for each message \( p \) of the protocol sent by agent \( A \), we have the clause \( \text{att}(p_1) \land \cdots \land \text{att}(p_n) \land m\text{-event}(p'_1) \land \cdots \land m\text{-event}(p'_{n'}) \Rightarrow \text{att}(p) \), where \( A \) receives messages \( p_1, \ldots, p_n \) and executes events \( p'_1, \ldots, p'_{n'} \) before sending message \( p \). Indeed, if the attacker has \( p_1, \ldots, p_n \), then he can send them to \( A \) and intercept \( A \)'s reply \( p \); the events \( p'_1, \ldots, p'_{n'} \) are executed by \( A \) during this operation.

Similarly, for each event \( p \) executed by \( A \), we have the clause \( \text{att}(p_1) \land \cdots \land \text{att}(p_n) \land m\text{-event}(p'_1) \land \cdots \land m\text{-event}(p'_{n'}) \Rightarrow \text{event}(p) \), where \( A \) receives messages \( p_1, \ldots, p_n \) and executes events \( p'_1, \ldots, p'_{n'} \) before executing event \( p \).

Let \( R_1 \) be the set of these clauses. This representation of protocols by Horn clauses is approximate, mainly because Horn clauses that represent the protocol can be applied any number of times instead of only once per session. However, it is sound: if the attacker knows \( p \) after the events \( p_1, \ldots, p_n \) have been executed, then \( \text{att}(p) \) is derivable from \( R_1 \) and the facts \( m\text{-event}(p_1), \ldots, m\text{-event}(p_n) \). In particular, if \( \text{att}(p) \) is not derivable from \( R_1 \) and any facts \( m\text{-event}(p') \), then the protocol preserves the secrecy of \( p \). If the event \( p \) is executed after the events \( p_1, \ldots, p_n \), then \( \text{event}(p) \) is derivable from \( R_1 \) and the facts \( m\text{-event}(p_1), \ldots, m\text{-event}(p_n) \). (This is proved by Theorem 1 in [7] when the clauses are generated from a pi calculus model of the protocol.)

ProVerif determines whether a fact is derivable from the clauses using resolution with free selection [2]; it combines pairs of clauses by resolution; the literals upon which we resolve are chosen by a selection function. Next, we detail this algorithm.

We say that \( R_1 \) subsumes \( R_2 \) when \( R_2 \) can be obtained by adding hypotheses to an instance of \( R_1 \). In this case, all facts derivable using \( R_2 \) can also be derived by \( R_1 \), so \( R_2 \) can be eliminated. Formally, subsumption is defined by:

**Definition 1 (Subsumption).** We say that \( R_1 = H_1 \Rightarrow C_1 \) subsumes \( R_2 = H_2 \Rightarrow C_2 \), and we write \( R_1 \supseteq R_2 \), if and only if there exists a substitution \( \sigma \) such that \( \sigma C_1 = C_2 \) and \( \sigma H_1 \subseteq H_2 \) (multiset inclusion).

When the conclusion of a clause \( R \) unifies with a hypothesis of a clause \( R' \), resolution creates a new clause that corresponds to applying \( R \) and \( R' \) one after the other.

**Definition 2 (Resolution).** Let \( R \) and \( R' \) be two clauses, \( R = H \Rightarrow C \) and \( R' = H' \Rightarrow C' \). Assume that there exists \( F_0 \in H' \) such that \( C \) and \( F_0 \) are unifiable, and \( \sigma \) is their most general unifier. In this case, we define \( R \cup R' = (H \cup (H' \setminus \{ F_0 \})) \Rightarrow \sigma C' \). The clause \( R \cup R' \) is the result of resolving \( R' \) with \( R \) upon \( F_0 \).

**SATUR**

1. \( R \leftarrow \text{ELIM}(R_0) \).
2. Repeat until a fixpoint is reached for each \( R \in \mathcal{R} \) such that sel\( (R) = \emptyset \), for each \( R' \in \mathcal{R} \), for each \( F_0 \in \text{sel}(R') \) such that \( R \cup R' \) is defined, \( R \leftarrow \text{ELIM}((R \cup R') \cup \mathcal{R}) \).
3. Return \( \{ R \in \mathcal{R} \mid \text{sel}(R) = \emptyset \} \).

**Figure 2: ProVerif’s Algorithm**

The facts upon which we resolve are selected through a selection function:

**Definition 3 (Selection Function).** A selection function is a function from clauses to sets of facts, such that sel\( (H \Rightarrow C) \subseteq H \). If \( F \in \text{sel}(R) \), we say that \( F \) is selected in \( R \). If \( \text{sel}(R) = \emptyset \), we say that no hypothesis is selected in \( R \), or that the conclusion of \( R \) is selected.

The algorithm is correct with any selection function that never selects facts of the form \( m\text{-event}(p) \), but the choice of the selection function can change the behavior of the algorithm. Facts \( att(x) \) where \( x \) is a variable can be unified with all facts \( att(M) \), so we should avoid selecting \( att(x) \) to reduce the number of possible resolution steps. Hence a good selection function satisfies the following formula:

\[
\text{sel}(H \Rightarrow C) = \begin{cases} \{ F_0 \} & \text{where } F_0 \in H, \\
\{ F_0 \} & \text{for all variables } x, F_0 \neq att(x), \text{ and} \\
\emptyset & \text{if there exists no such } F_0 
\end{cases}
\]

The resolution algorithm is shown in Figure 2. It transforms the initial set of clauses into a new one that derives the same facts. The algorithm \( \text{SATUR}(R_0) \) has 3 steps:

- The first step inserts in \( R \) the clauses in \( R_0 \) after elimination of subsumed clauses by \( \text{ELIM} \); when \( R' \supseteq R \) and both \( R \) and \( R' \) are in \( \mathcal{R} \), \( R \) is removed by \( \text{ELIM}(\mathcal{R}) \).
- The second step is a fixpoint iteration that adds the clauses created by resolution: if the conclusion of a clause \( R \) such that \( \text{sel}(R) = \emptyset \) unifies with \( F_0 \in \text{sel}(R') \), then the resolution \( R \cup R' \) is added to \( \mathcal{R} \). Then, subsumed clauses are eliminated by \( \text{ELIM} \).
- Finally, the algorithm returns the clauses in \( \mathcal{R} \) with no selected hypothesis.

Let \( F_{me} \) be any set of facts of the form \( m\text{-event}(p) \).

**Theorem 1 (Lemma 2 in [7]).** Let \( F \) be a closed fact and \( R_0 \) a set of clauses. \( F \) is derivable from \( R_0 \cup F_{me} \) if and only if it is derivable from \( \text{SATUR}(R_0) \cup F_{me} \).

To prove that a closed fact \( \text{att}(p) \) is not derivable from \( R_0 \cup F_{me} \), we use the following result, where \( att' \) is a new predicate:

**Corollary 1.** If \( \text{SATUR}(R_1 \cup \{ \text{att}(p) \Rightarrow \text{att}'(p) \}) \) contains no clause of the form \( H \Rightarrow \text{att}'(p) \), then \( \text{att}(p) \) is not derivable from \( R_1 \cup F_{me} \) for any \( F_{me} \). So, by soundness of the clauses, the protocol preserves the secrecy of \( p \).
Indeed, if $\text{SATUR}(R_1 \cup \{\text{att}(p) \Rightarrow \text{att}'(p)\})$ contains no clause of the form $H \Rightarrow \text{att}'(p')$, then $\text{att}'(p')$ is not derivable from $\text{SATUR}(R_1 \cup \{\text{att}(p) \Rightarrow \text{att}'(p)\}) \cup F_{\text{me}}$, so by Theorem 1, $\text{att}'(p')$ is not derivable from $R_1 \cup \{\text{att}(p) \Rightarrow \text{att}'(p)\} \cup F_{\text{me}}$, so $\text{att}(p)$ is not derivable from $R_1 \cup F_{\text{me}}$. Similarly, we also have:

**Corollary 2** (Corollary 2 in [7]). Suppose that all clauses of $\text{SATUR}(R_1)$ that conclude $\text{event}(e(p))$ for some $p$ are of the form $m \cdot \text{event}(e'(p')) \land H \Rightarrow \text{event}(e(p'))$ for some $H$ and $p'$. Then, for all $F_{\text{me}}$, for all $p$, if $\text{event}(e(p))$ is derivable from $R_1 \cup F_{\text{me}}$, then $m \cdot \text{event}(e'(p')) \in F_{\text{me}}$. So by soundness of the clauses, the protocol satisfies the correspondence "if $e(x)$ has been executed, then $e'(x)$ has been executed".

### 3. MOTIVATION

This section motivates this work, explaining the introduction of a new type of clauses through a running example.

#### 3.1 Running Example

As a running example, we use a version of the SOAP extension to XML signatures [9]. SOAP envelopes are XML documents with a mandatory `Body` containing a request, response or a fault message together with an optional `Header` element containing application-specific information about the message (for example security information). In particular, the SOAP header can carry digital signature information within a SOAP envelope, as follows:

```xml
<Envelope>
  <Header/>
  <Body Id="#theBody"> request </Body>
</Envelope>
```

The `Signature` header contains two components. The first component is a `SignedInfo` element, which is basically a list of references to the elements of the message that are signed, designated by their identifier and accompanied by a `DigestValue`, a hash of their content. The hash may be computed with the hash function SHA-1. The second component of the `Signature` header is the signature of the `SignedInfo` element using a secret key $sk$.

We consider a simple protocol in which a client sends such a message to a server. The server processes the document and checks the signature before authorizing the request given in the `Body`: if the `SignedInfo` contains a `Reference` to an element with tag `Body`, then he will authorize the request. This protocol should guarantee that the server authorizes only requests signed by legitimate clients.

This protocol is subject to a known wrapping attack [18]: an attacker can intercept an envelope and create a new envelope wrapping the `Body` in the `Header` and adding a new body with a different id and a different request. The server will verify the signature and authorize the fake request made by the attacker. This attack is possible because the server does not check that the signed `Body` is the one he authorizes.

#### 3.2 Need for Generalizing Horn Clauses

In order to model the previous example, we suppose that the XML parser parses the SOAP envelope as a pair, containing as first component a list of triplets (tag, id, corresponding content) and as second component the content of the mandatory body. The list in the first component is helpful so that one can retrieve the content of an element from its id by looking up the list. The content of the `Signature` header is modeled as a pair `(SignedInfo, SignatureValue); SignedInfo` is a list of pairs containing an id and the hash of the content corresponding to the id; `SignatureValue` is the signature of `SignedInfo` with a secret key $sk$. We use such a simple model for illustrative purposes.

A given XML document can then be represented using lists of fixed length (and other standard functions). We denote by $(p_1, \ldots, p_n)$ a list of fixed length $n$; such lists are modeled as a family of constructors, one for each length.

However, the receiver of the SOAP envelope accepts messages containing any number of headers, and the `SignedInfo` element may also contain references to any number of elements in the message. Therefore, we need lists of variable length in order to model the expected message. So we introduce the construct `list(i \leq M, p_i)` which stands for $(p_1, \ldots, p_M)$, for an unknown $M$ (inspired by [12, 19]).

Thanks to this new construct, we can model the SOAP envelope from the point of view of the server: `list(i \leq M, (tag_i, id_i, cont_i)), r)`, where `tag_i`, `id_i`, and `cont_i` are variables representing tags, identifiers, and contents respectively and $r$ is the variable for the request. The server has to check the signature, that is, he has to verify that the list contains a `tag` equal to `Signature` and that `cont`, contains a correct signature. These checks can be represented by the following equations: $\text{tag}_i \equiv \text{Signature}, \text{cont}_i = (\text{signo, sign}(\text{signo, sk}))$, $\text{signo} = \text{list}(k \leq N, (\text{id}_k, \text{sha}(\text{cont}_k)))$, where the function $\phi$ is a mapping from the index $k$ of each element in the `signo` list to its index $\phi(k)$ in the list that represents the whole message. Furthermore, one of the signed elements must have tag `Body`, that is, $\phi(n) = d, \text{tag}_d = \text{Body}$. We cannot directly replace the variables `tag_i`, `id_i`, and `cont_i` with their values given by these equations, because in the list `list(i \leq M, (tag_i, id_i, cont_i))`, all elements `tag_i`, `id_i`, and `cont_i` need to have the same form, while the equations give different forms to different elements. So we keep the equations in the clause for future use. The equations allow us to handle protocols that treat elements of lists non-uniformly.

We can represent that the adversary has a SOAP envelope by $\text{att}((\text{list}(i \leq M, (\text{tag}_i, \text{id}_i, \text{cont}_i)), r))$. The adversary can build such a SOAP envelope from its components, so we need to express that the adversary has `(tag_i, id_i, cont_i)` for all $i \leq M$. We use a conjunction $\bigwedge_{i \in [1, M]} \text{att}((\text{tag}_i, \text{id}_i, \text{cont}_i))$ for this purpose. More generally, $\bigwedge_{(i_1, \ldots, i_k) \in F} \text{att}((\text{cont}_{tag_{i_1}}, \ldots, \text{cont}_{tag_{i_k}}))$ represents the conjunction of facts `F` for all indices $(i_1, \ldots, i_k) \in F$. After modeling the clauses, we need to perform resolution on these generalized clauses. Let $[1, M] = \{1, \ldots, M\}$ and suppose that we resolve $R_2 = \bigwedge_{i \in [1, M]} \text{att}(\text{sha}(\text{cont}_{tag_i})) \Rightarrow$...
event(e(r)) with the clause $R_1 = \text{att}(x) \Rightarrow \text{att}(sha1(x))$, that is, we use $R_1$ to derive one hypothesis of $R_2$, say the one of index $i = k_1 \in [1, M]$. So we unify $\text{att}(sha1(x))$ with $\text{att}(sha1(\text{cont}_φ(i)))$, yielding the equality $x = \text{cont}_φ(i)$. The obtained clause would then be:

$$\text{att}(x_1) \land \bigwedge_{i \in [1,M] \setminus \{k_1\}} \text{att}(sha1(\text{cont}_φ(i))) \land \{x_1 = \text{cont}_φ(k_1)\} \Rightarrow \text{event}(e(r))$$

The variable $x$ is renamed into $x_1$ to distinguish it from the variable $x$ in $R_1$. The obtained clause can then again be resolved with $R_1$ for some $i = k_2 \in [1, M] \setminus \{k_1\}$, yielding

$$\text{att}(x_1) \land \text{att}(x_2) \land \bigwedge_{i \in [1,M] \setminus \{k_1,k_2\}} \text{att}(sha1(\text{cont}_φ(i))) \land \{x_1 = \text{cont}_φ(k_1), x_2 = \text{cont}_φ(k_2)\} \Rightarrow \text{event}(e(r))$$

which can again be resolved with $R_1$, and so on. Since $M$ is not bounded, such resolution steps yield an infinite loop. To avoid this loop, we define a resolution step that simultaneously resolves $R$ with several instances of $R'$ for $i$ in any non-empty subset $I \subseteq [1, M]$. We name such a step hyperresolution by analogy with the hyperresolution rule that allows one to resolve one clause with several clauses [14].

The obtained clause would then be:

$$\text{att}(x_1) \land \bigwedge_{i \in I} \text{att}(sha1(\text{cont}_φ(i))) \land \{x_1 = \text{cont}_φ(i)\} \Rightarrow \text{event}(e(r))$$

where $I$ and $I'$ represent non-empty subsets of $[1, M]$. The patterns $\text{cont}_φ(i)$ are performed for all $i$, so we obtain

$$\bigwedge_{i \in I} \text{att}(\text{cont}_φ(i)) \Rightarrow \text{event}(e(r)).$$

When $I = [1, M]$, the second hypothesis is removed and the substitution of $\text{cont}_φ(i)$ for $x_i$ is performed for all $i$, so we obtain

$$\bigwedge_{i \in [1,M]} \text{att}(\text{cont}_φ(i)) \Rightarrow \text{event}(e(r)).$$

When $I \neq [1, M]$, to have a more symmetric notation and to simplify the language of sets used in conjunctions, we introduce a symbol $I'$ for $[1, M] \setminus I$, and keep the constraint that $I \sqcup I' = [1, M]$; $I$ and $I'$ are disjoint and their union is $[1, M]$. The clause is then denoted by

$$I \sqcup I' = [1, M], \bigwedge_{i \in I} \text{att}(\text{cont}_φ(i)) \land \bigwedge_{i \in I'} \text{att}(\text{sha1}(\text{cont}_φ(i))) \Rightarrow \text{event}(e(r)).$$

4. GENERALIZED HORN CLAUSES

This section formally defines the syntax and semantics of generalized Horn Clauses, which were motivated informally in the previous section.

4.1 Syntax

The syntax of these new clauses is defined in Figure 3. The patterns $p^G$ that represent messages are enriched with several new constructs. The variables may have indices $x_{i_1}, \ldots, x_{i_n}$. The pattern for function application $f(p^G_{i_1}, \ldots, p^G_{i_n})$ includes not only constructor application but also names $a[p^G_{i_1}, \ldots, p^G_{i_n}]$ where $a$ is a name without index. We consider tuples $(p^G_1, \ldots, p^G_m)$ and lists of fixed length $(p^G_1, \ldots, p^G_m)$ as particular constructors. We suppose that the implementation of the protocol uses distinct encodings for tuples and for lists, so that they cannot be confused with each other. The constructs $a^M[p^G_{i_1}, \ldots, p^G_{i_n}]$ represents a fresh name $a$ indexed by $i$ in $[1, M]$. For instance, in the context of group protocols, it may represent a name created by the group member number $i$, inside a group of size $M$. We use the construct $\text{list}(i \leq M, p^G)$ to represent lists of length $M$.

We extend facts to model the possibility of having a conjunction of facts depending on indices, so that the facts become $\bigwedge_{i \in J} \text{pred}(p^G_i)$. The set $J$ can be a product of a set symbol $I$ or a singleton $\{\}$ and of different sets $[1, M]$, depending on different bounds $M$. The symbol $[1, M]$ represents the set $\{1, \ldots, M\}$. The symbol $[1, M]$ allows us to keep the information that an index ranges over the full set of indices of bound $M$, while other set symbols $I$ represent an unknown, non-empty set of indices. We write $[1, M]$ instead of $\{\} \times [1, M]$. For example, intuitively, $\bigwedge_{i \in [1, M]} \text{pred}(p^G_i)$ represents $\text{pred}(p^G_{\{1\}}) \land \cdots \land \text{pred}(p^G_{\{M\}})$, where $p^G_{\{i\}}$ denotes $p^G$ in which $i$ has been replaced with $i'$. The conjunction $\bigwedge_{(i_1, \ldots, i_n) \in J} \text{pred}(p^G_i)$ is then simply $\text{pred}(p^G_J)$.

In the generalized Horn clause $Cts, F^G_1 \land \cdots \land F^G_n \land E \Rightarrow \text{pred}(p^G)$, there are two new members: a set of constraints $Cts$ and a set of equations $E$. The first one is a set of constraints on sets used in conjunctions: the constraint $I_1 \sqcup \cdots \sqcup I_h = J$ means that $I_1, \ldots, I_h$ are pairwise disjoint and their union is $J$. The second one is a set of equations that represent the substitutions that cannot be done because the equations hold for some but not all values of the indices. These equations can be equations over patterns $E$ and equations over indices $E'$. The clause $Cts, F^G_1 \land \cdots \land F^G_n \land E \Rightarrow \text{pred}(p^G)$ means that, if the constraints in $Cts$ are satisfied,
and the facts $F^G_1, \ldots, F^G_n$ and the equations in $E$ hold, then the fact $\text{pred}(p^G)^*$ also holds. The conclusion of a clause does not contain a conjunction $C$: we can simply leave the indices of $\text{pred}(p^G)$ free to mean that $\text{pred}(p^G)$ can be concluded for any value of these indices.

The clauses are required to satisfy the following invariants:

1. In a conjunction $\bigwedge_{i_1, \ldots, i_n} C$, the indices $i_1, \ldots, i_n$ are pairwise distinct.
2. Each set symbol $I$ occurs at most once in each constraint in $\text{Cts}$.
3. Each set symbol $I$ occurs in at most one left-hand side of a constraint in $\text{Cts}$.
4. If a set symbol $I$ occurs in a conjunction or in the right-hand side of a constraint in $\text{Cts}$, then it occurs in exactly one left-hand side of a constraint in $\text{Cts}$.

We use $H^G$ for hypothesis and $C^G$ for conclusions. When $\text{Cts}$ or $E$ are empty, we omit them in the clause.

### 4.2 Representation of the Protocol

The representation of the abilities of the attacker includes the clauses given in Section 2. For our running example, $\text{att}(pk(\text{sk}))$, $\text{att}(\text{Signature})$, $\text{att}(\text{Body})$ represent that the attacker initially knows the public key $pk(\text{sk})$ and the constants $\text{Signature}$ and $\text{Body}$, and the clauses

\[
\begin{align*}
\text{att}[a[x]] & \\
\text{att}(x) & \land \text{att}(y) \Rightarrow \text{att}(\text{sign}(x, y)) & \\
\text{att}(x) & \Rightarrow \text{att}(pk(x)) & \\
\text{att}(x) & \Rightarrow \text{att}(\text{sha1}(x)) & \\
\text{att}(x) & \land \text{att}(y) \Rightarrow \text{att}((x, y))
\end{align*}
\]

represent that the attacker can create fresh names, sign messages, create its own public keys, apply hash functions, compose and decompose pairs and lists of length one. We have similar clauses for triples and lists of length two. (These arities are the only ones used in our example.)

In addition, we have clauses for $\text{list}$, which generalize clauses for tuples and lists of fixed length:

\[
\begin{align*}
\forall n \in [1, M], \text{att}(x_i) & \Rightarrow \text{att}(\text{list}(j \leq M, x_i)) \quad (1) \\
\text{att}(\text{list}(j \leq M, x_i)) & \Rightarrow \text{att}(x_i) \quad (2)
\end{align*}
\]

Let us now give clauses that represent the protocol itself. To model the authentication, we use two events $b$ and $e$. The event $b(r)$ means that the client sends the request $r$; the event $e(r)$ means that the server authorizes the request $r$. Our goal is to prove that, if $e(r)$ is executed, then $b(r)$ has been executed. We suppose that the only element signed by the client is the $\text{body}$. Hence the document can be represented as follows:

\[
\text{[(\text{Signature}, ids, (\text{body, sha1(Req)})), sign(\text{(body, sha1(Req))}, \text{sk})], (\text{Body, idb, Req})}, \text{Req})
\]

where $\text{ids}$ is the identifier of the $\text{Signature}$ and $\text{idb}$ the one of the $\text{Body}$. The client executes the event $b(\text{Req})$, then sends the SOAP envelope on the network and the attacker intercepts it, so we have the clause:

\[
\text{m-event}(b(\text{Req})) \Rightarrow \text{att}(\text{[(\text{Signature, ids, (\text{body, sha1(Req))}, sign(\text{(body, sha1(Req))}, \text{sk})]}), (\text{Body, idb, Req})}, \text{Req})
\]

The server expects a document $(\text{list}(i \leq M, (\text{tag}_i, \text{ids}_i, \text{cont}_i)), r)$; he checks the signature, and if the check succeeds, he executes the event $e(r)$:

\[
\begin{align*}
\text{att}(\text{list}(i \leq M, (\text{tag}_i, \text{ids}_i, \text{cont}_i)), r) & \land \\
\text{tag}_j & \equiv \text{Signature}, \text{cont}_i \equiv (\text{sign(a, sign(b, sk)))}, \text{sign(a) \equiv list(k \leq N, (id(a), \text{sha1(cont(a))})}, \\
\phi(m) & \equiv d, \text{tag}_a \equiv \text{Body}) \Rightarrow \\
\text{event}(e(r))
\end{align*}
\]

This clause uses the equations explained in Section 3.2.

As already mentioned in Section 3.1, this protocol has a wrapping attack. The corrected version of this protocol additionally requires that the signed body is the one that the server authorizes. This additional check can be modeled by adding the equation $\text{cont}_a \equiv r$ in the clause (4).

### 4.3 Type System

In this section, we define a simple type system for generalized Horn clauses, to guarantee that the indices of all variables vary in the appropriate interval.

**Definition 4.** An index $i$ is bound if:

- it appears as an index of a conjunction defining a fact, so, for instance, in the fact $\forall x, y, z. \text{pred}(p^G), i, \ldots, h_6$ are bound in $\text{pred}(p^G)$.
- it appears as an index for a list constructor, that is, in the pattern $\text{list}(\leq M, p^G)$, $i$ is bound in $p^G$.

We identify facts, equations, and clauses up to renaming of bound indices. For simplicity, we suppose that the bound indices of clauses have been renamed so that they have pairwise distinct names, and names distinct from the names of free indices. The set of free indices of a fact $F_G^*$, of a clause $C^G$, or of an hypothesis $H^G$ is denoted by $\text{f}(F_G^*), \text{f}(C^G), \text{f}(H^G)$ respectively.

In the type system, the type environment $\Gamma$ is a list of type declarations:

\[
\begin{align*}
\cdot & : [1, M] \text{ means that } i \text{ is of type } [1, M], \text{ that is, intuitively, the value of index } i \text{ can vary between 1 and the value of the bound } M; \\
\cdot & : [1, M_1] \times \cdots \times [1, M_h] \rightarrow [1, M] \text{ means that the function } \phi \text{ expects as input } h \text{ indices of types } [1, M_1], \ldots, [1, M_h]; \\
\cdot & : [1, M_1] \times \cdots \times [1, M_h] \text{ means that the variable } x \text{ expects indices of types } [1, M_1], \ldots, [1, M_h]; \\
\cdot & : [1, M_1] \times \cdots \times [1, M_h] \text{ means that the elements of set } I \text{ are tuples of } h \text{ indices, each of type } [1, M_1] \text{ for } j = 1, \ldots, h.
\end{align*}
\]

The type system defines the judgments:

\[
\begin{align*}
\Gamma \vdash \iota : [1, M], \text{ which means that } \iota \text{ has type } [1, M] \text{ in the type environment } \Gamma; \\
\Gamma \vdash J : [1, M_1] \times \cdots \times [1, M_h], \text{ which means that the elements of set } J \text{ are tuples of } h \text{ indices, each of type } [1, M_j] \text{ for } j = 1, \ldots, h \text{ in the type environment } \Gamma.
\end{align*}
\]
• $\Gamma \vdash p^G$, $\Gamma \vdash F^G$, $\Gamma \vdash E$, $\Gamma \vdash E'$, $\Gamma \vdash I$, $\Gamma \vdash R^G$, which mean that $p^G$, $F^G$, $E$, $E'$, $I$, $R^G$, respectively, are well-typed in the type environment $\Gamma$.

The typing rules are straightforward and are given in the long version of the paper [8]. They basically guarantee that all indices have their expected type.

We suppose that all clauses are well-typed, and we consider that each clause $R^G$ comes with its type environment $\Gamma$ such that $\Gamma \vdash R^G$. It is easy to verify that the clauses of our running example are all well-typed.

4.4 Translation from Generalized Horn Clauses to Horn Clauses

A generalized Horn clause represents several Horn clauses: for each value of the bounds $M$, set symbols $I$, functions $\phi$, and free indices $i$ that occur in a generalized Horn clause $R^G$, $R^G$ corresponds to a certain Horn clause. This section formally defines this correspondence.

**Definition 5.** Given a well-typed generalized Horn clause $\Gamma \vdash R^G$, an environment $T$ for $\Gamma \vdash R^G$ is a function that associates:

- to each bound $M$ that appears in $R^G$ or $\Gamma$ an integer $M^T$;
- to each index $i$ such that $i : [1, M] \in \Gamma$, an index $i^T \in \{1, \ldots, M^T\}$;
- to each index function $\phi$ such that $\phi : [1, M_1] \times \cdots \times [1, M_k] \to [1, M] \in \Gamma$, a function $\phi^T : \{1, \ldots, M_1^T\} \times \cdots \times \{1, \ldots, M_k^T\} \to \{1, \ldots, M^T\}$;
- to each set symbol $I$ such that $I : [1, M_1] \times \cdots \times [1, M_k] \in \Gamma$, a set $I^T$ such that $\emptyset \subset I^T \subseteq \{1, \ldots, M_1^T\} \times \cdots \times \{1, \ldots, M_k^T\}$;

Given an environment $T$ and values $v_1, \ldots, v_h$, we write $T[v_1 \mapsto v_1, \ldots, v_h \mapsto v_h]$ for the environment that associates to indices $i_1, \ldots, i_h$ the values $v_1, \ldots, v_h$ respectively and that maps all other values like $T$.

Given an environment $T$ for $\Gamma \vdash R^G$, the generalized Horn clause $R^G$ is translated into the standard Horn clause $R^{GT}$ defined as follows. We denote respectively $p^{GT}, E^T$, $\cdots$ the translation of $p^G, E, \cdots$ using the environment $T$.

The translation of an index term $i$ such that $\Gamma \vdash i : [1, M]$ is an integer $i^T \in \{1, \ldots, M^T\}$ defined as follows:

\[
T = \begin{cases} 
   i^T & \text{if } i = i \\
   \phi^T(i^T, \ldots, i^T) & \text{if } i = \phi(i_1, \ldots, i_k)
\end{cases}
\]

The translation of a pattern $p^G$ is defined as follows:

\[
(x_1, \ldots, x_n)^T = (x_1^T, \ldots, x_n^T)
\]

\[
f(p^G_1, \ldots, p^G_n)^T = f(p^{GT}_1, \ldots, p^{GT}_n)
\]

\[
a_{M_1}^G[p^G_1, \ldots, p^G_n]^T = a_{M_1}^T[p^{GT}_1, \ldots, p^{GT}_n]
\]

\[
\text{list}(i \leq M, p^G)^T = (g^{GT[i-1]}_n, \ldots, g^{GT[i-M]})
\]

The translation of $\text{list}(i \leq M, p^G)$ is a list of length $M^T$.

The translation of a set computation $J$ is a set of tuples defined by:

\[
J^T = \begin{cases} 
   \{()\} & \text{if } J = \{()\} \\
   J^T \times \{1, \ldots, M^T\} & \text{if } J = J' \times [1, M]
\end{cases}
\]

In this definition, the cross product $\times$ decomposes tuples before building the whole tuple, so that $J^T \times \{1, \ldots, M^T\} = \{(v_1, \ldots, v_n, v) \mid (v_1, \ldots, v_n) \in J^{T'}, v \in \{1, \ldots, M^T\}\}$.

Given a conjunction $\mathcal{C} = \bigwedge_{(i_1, \ldots, i_n) \in \mathcal{C}}$ and an environment $T$, we define the set of environments $T^C = \{T[v_1 \mapsto v_1, \ldots, v_h \mapsto v_h] \mid (v_1, \ldots, v_h) \in J^T\}$: these environments map the indices $i_1, \ldots, i_n$ of the conjunction to all their possible values in $J^T$ and map all other values like $T$.

The translation of a fact $F = C \text{ pred}(p^G)$ is

\[
(C \text{ pred}(p^G))^T = \text{pred}(p^G) \land \cdots \land \text{pred}(p^G)
\]

where $\{p_1, \ldots, p_k\} = \{p^{GT'} \mid T' \in T^C\}$, and $(F_1 \land \cdots \land F_n)^T = F_1^T \land \cdots \land F_n^T$.

The translation of a set of equations $\mathcal{E}$ is the set $\mathcal{E}^T$ obtained by translating the equations $E, E'$ as follows:

- $(C \equiv C')^T = \begin{cases} 
   \text{true} & \text{if for all } T' \in T^C, i^T = i'^T \\
   \text{false} & \text{otherwise}
\end{cases}$

- $(C \equiv C')^T = \{p^{GT'} \land p^{GT'} \mid T' \in T^C\}$

If $\forall E' \in E, E' \equiv C^T$, then $\mathcal{E}^T = \bigcup_{E' \in E} E'$; otherwise, $\mathcal{E}^T$ is undefined.

The equations over indices can be evaluated to true or false knowing the environment $T$. The equations over generalized patterns are translated into equations over patterns. A set of equations $\mathcal{E}$ is translated into a set of equations over patterns if all equations over indices are true. Otherwise, the translation of $\mathcal{E}$ is undefined.

Given a set of constraints $Cts$ and an environment $T$, we say that $T$ satisfies $Cts$ when, for all equations $I_1 \vdash \cdots \vdash I_m = J$ in $Cts$, we have that $I_1^T \vdash \cdots \vdash I_m^T = J^T$, that is, $I_1^T, \ldots, I_m^T$ are pairwise disjoint and their union is $J^T$.

Given a set of equations $\{p_1 = p_1', \ldots, p_n = p_n\}$ over standard patterns, we define as usual its most general unifier $\text{mu}(\{p_1 = p_1', \ldots, p_n = p_n\})$ as the most general substitution $\sigma$ such that $\sigma p_i = \sigma p_i'$ for all $i \in \{1, \ldots, n\}$, $\text{dom}(\sigma) \subseteq \text{fv}(\text{im}(\sigma)) \subseteq \text{fv}(p_1, p_1', \ldots, p_n, p_n')$, and $\text{dom}(\sigma) \cap \text{fv}(\text{im}(\sigma)) = \emptyset$, where $\text{fv}(p)$ designates the (free) variables of $p$, $\text{dom}(\sigma)$ is the domain of $\sigma$: $\text{dom}(\sigma) = \{x \mid \sigma x \neq x\}$, and $\text{im}(\sigma)$ is the image of $\sigma$: $\text{im}(\sigma) = \{\sigma x \mid \sigma x \neq x\}$. We denote by $\{x_1 \mapsto p_1, \ldots, x_n \mapsto p_n\}$ the substitution that maps $x_i$ to $p_i$ for all $i = 1, \ldots, n$.

Finally, we define the translation of the generalized Horn clause $R^{GT} = Cts, H^G \land \mathcal{E} \Rightarrow \text{pred}(p^G)$ as follows. If $T$ does not satisfy $Cts$, $\mathcal{E}^T$ is undefined, or the unification of $\mathcal{E}^T$ fails, then $R^{GT}$ is undefined. Otherwise, $R^{GT} = \text{mu}(\mathcal{E}^T) R^{GT} \Rightarrow \text{mu}(\mathcal{E}^T) \text{pred}(p^G)$.

When $R^{GT}$ is a set of well-typed generalized Horn clauses (i.e., a set of pairs of a type environment $\Gamma$ and a clause $R^G$ such that $\Gamma \vdash R^G$), we define $R^{GT} = \{R^{GT'} \mid \Gamma \vdash R^G \in R^G, T\}$ as the environment for $\Gamma \vdash R^G$ and $R^{GT}$ is defined). In terms of abstract interpretation, the sets of generalized Horn clauses ordered by inclusion constitute the abstract domain, the sets of Horn clauses ordered by inclusion the concrete domain, and $R^{GT}$ is the concretization of $R^G$.

5. RESOLUTION FOR GENERALIZED HORN CLauses

In this section, we adapt ProVerif’s resolution algorithm to generalized Horn clauses: we first define adapted algo-
rithms for subsumption, resolution, and unification, and then use these components to build the new resolution algorithm.

5.1 Subsumption

To adapt the subsumption test to generalized Horn clauses, we first define adapted notions of substitutions, then define subsumption and prove its soundness.

**Definition 6 (Substitutions).** We denote by \( \rho \) a substitution that maps:

- bounds to bounds \( \rho(M) = N \);
- index variables to index terms: \( \rho(i) = i \);
- set symbols to set symbols: \( \rho(I) = I' \);
- function symbols \( \phi \) to function symbols: \( \rho(\phi) = \phi' \).

We denote by \( \sigma^G \) a substitution that maps variables to patterns: \( \sigma^G(x_{1}, \ldots, x_{k}) = p^G \) where \( f(p^G) \subseteq \{i_{1}, \ldots, i_{k}\} \).

As usual, substitutions are extended from variables to index terms, patterns, facts, equations, constraints, and clauses as homomorphisms: for instance, \( \rho(\phi(t_{1}, \ldots, t_{k})) = (\rho(\phi))(\rho(t_{1}), \ldots, \rho(t_{k})) \). However, since the grammar of generalized Horn clauses is richer than usual, we need to notice two points:

- Substitutions avoid the capture of bound indices: if \( i \notin \text{dom}(\rho) \) and \( i \notin f(\text{im}(\rho)) \), \( \rho(\text{list}(i \leq M, p^{G})) = \text{list}(i \leq \rho(M), \rho(p^{G})) \), which can be guaranteed by renaming \( i \) if necessary. We have similar formulas for indices bound by conjunctions as well as for substitutions \( \sigma^{G} \).
- The substitutions \( \sigma^{G} \) are extended from variables \( x_{1}, \ldots, x_{n} \) with indices \( i_{1}, \ldots, i_{k} \) to variables \( x_{1}, \ldots, x_{k} \) with terms \( \rho(\phi(t_{1}, \ldots, t_{k})) = p^{G} \), then \( \sigma^{G}(x_{1}, \ldots, x_{n}) = \rho\sigma^{G} \) where \( \rho = \{i_{1} \mapsto i_{1}, \ldots, i_{k} \mapsto i_{k}\} \).

We need a notion of subsumption between type environments, which is intuitively \( \Gamma \subseteq \Delta \). However, we cannot use exactly \( \rho \Gamma \subseteq \Delta \) because \( \rho \) maps indices to index terms and not to indices, so we define \( \Gamma_{1} \preceq_{\rho} \Gamma_{2} \) by:

- for each \( i : [1, M] \in \Gamma_{1} \), we have \( \Gamma_{2} \vdash \rho i : [1, \rho M] \);
- for each \( \phi : [1, M_{1}] \times \cdots \times [1, M_{k}] \rightarrow [1, M] \in \Gamma_{1} \), we have \( \rho \phi : [1, \rho M_{1}] \times \cdots \times [1, \rho M_{k}] \rightarrow [1, \rho M] \in \Gamma_{2} \);
- for each \( f : [1, M_{1}] \times \cdots \times [1, M_{k}] \in \Gamma_{1} \), we have \( \rho f : [1, \rho M_{1}] \times \cdots \times [1, \rho M_{k}] \in \Gamma_{2} \);
- for each \( x : [1, M_{1}] \times \cdots \times [1, M_{k}] \in \Gamma_{1} \), we have \( x : [1, \rho M_{1}] \times \cdots \times [1, \rho M_{k}] \in \Gamma_{2} \).

**Definition 7 (Subsumption).** Given two well-typed generalized Horn clauses, \( \Gamma_{1} \vdash C_{1}^{G} \) and \( \Gamma_{2} \vdash C^{G} \), with \( C_{1}^{G} = C_{\text{cls}}^{G}, H_{1}^{G} \land E_{1} \Rightarrow C_{1}^{G} \) and \( C_{2}^{G} = C_{\text{cls}}^{G}, H_{2}^{G} \land E_{2} \Rightarrow C_{2}^{G} \), we say that \( \Gamma_{1} \vdash C_{1}^{G} \) subsumes \( \Gamma_{2} \vdash C_{2}^{G} \), and we write \( \Gamma_{1} \preceq_{\rho} C_{1}^{G} \preceq_{\rho} C_{2}^{G} \), if there exist substitutions \( \rho \) and \( \sigma^{G} \) such that \( \sigma^{G} \rho C_{1}^{G} = C_{2}^{G} \), \( \sigma^{G} \rho H_{1}^{G} \subseteq H_{2}^{G} \) (multi-set inclusion), \( \sigma^{G} \rho E_{1} \subseteq E_{2} \) (modulo commutativity of \( \cdot \)), \( \rho C_{\text{cls}}^{G} \subseteq C_{\text{cls}}^{G} \) (modulo associativity and commutativity of \( \cdot \)) and \( \Gamma_{1} \preceq_{\rho} \Gamma_{2} \).

This definition is similar to subsumption for standard clauses but uses the richer substitutions introduced above.

**Example 1.** The clause

\[
R_{1}^{G} = \{ I \cup I_{1} = [1, M] \}, \{ \land_{m \in I_{1}, y_{m} = e^{M}} \} \Rightarrow \text{att}(y)
\]

typed by the type environment \( \Gamma_{1} = \{ I : [1, M], I_{1} : [1, M], x_{1} : [1, M], \phi : [1, M] \rightarrow [1, L], y : [1, M], l : [1, M] \} \), subsumes the clause

\[
R_{2}^{G} = \{ I' \cup I'_{1} = [1, N] \}, \{ \land_{m \in I'_{1}, z_{m} = c^{N}} \} \Rightarrow \text{att}(z_{\psi(k)})
\]

typed by the environment \( \Gamma_{2} = \{ I' : [1, N], I'_{1} = [1, N], \psi : [1, N] \rightarrow [1, N], \phi : [1, N] \rightarrow [1, O], k : [1, N], z_{m} : [1, N] \} \). With \( \rho = \{ M \rightarrow N, L \rightarrow O, I \rightarrow I', I_{1} \rightarrow I'_{1}, \phi \rightarrow \phi, l \rightarrow \psi(k) \} \) and renaming the bound indices \( i \) into \( j \) and \( m \) into \( n \), we obtain:

\[
\rho R_{1}^{G} = \{ I' \cup I'_{1} = [1, N] \}, \{ \land_{m \in I'_{1}, y_{m} = c^{N}} \} \Rightarrow \text{att}(y)_{\psi(k)}
\]

and conclude that \( R_{1}^{G} \) subsumes \( R_{2}^{G} \) using the substitution \( \sigma^{G} = \{ x_{1} \mapsto (r_{\psi(k)}', u'y_{1}), y_{1} \mapsto z_{1} \} \).

The following theorem shows the soundness of subsumption for generalized Horn clauses.

**Theorem 2 (Subsumption).** Let \( \Gamma_{1} \vdash C_{1}^{G} \) and \( \Gamma_{2} \vdash C_{2}^{G} \) be two well-typed clauses. If \( \Gamma_{1} \vdash C_{1}^{G} \preceq_{\rho} \Gamma_{2} \vdash C_{2}^{G} \), then, for all environments \( T_{k} \) for \( \Gamma_{2} \vdash C_{2}^{G} \) such that \( R_{k}^{GT_{k}} \) is defined, there exists an environment \( T_{1} \) for \( \Gamma_{1} \vdash C_{1}^{G} \) such that \( R_{k}^{GT_{1}} \) is defined and \( R_{k}^{GT_{1}} \) \( \preceq_{\rho} \) \( R_{k}^{GT_{2}} \).

As for standard clauses, our resolution algorithm for generalized Horn clauses will eliminate subsumed clauses. This theorem shows that, if \( R_{2}^{G} \) is eliminated in the algorithm for generalized Horn clauses because it is subsumed by \( R_{1}^{G} \), then all corresponding clauses \( R_{2}^{GT_{2}} \) would be eliminated in the algorithm for standard clauses: \( R_{2}^{GT_{2}} \) is subsumed by some \( R_{1}^{GT_{1}} \).

The subsumption test is approximate, because clauses that have the same meaning may sometimes be written in different forms. For instance, one can compose two functions \( \rho \) and \( \psi \), always writing \( \rho(\psi(\iota, j)) \) instead of a single function \( \psi(i, j) \). These two formulas yield the same translation into standard Horn clauses, but are considered different by the subsumption test. This approximation does not contradict the soundness of subsumption, and the subsumption test is precise enough for the proof to succeed on our examples.

Because of the presence of indices, the algorithm for computing the substitutions \( \rho \) and \( \sigma^{G} \) required in Definition 7 is more complex than for standard clauses. We detail it in the long version of the paper [8].

5.2 Resolution and Hyperresolution

As introduced in Section 3.2, resolution becomes hyper-resolution for generalized clauses. We first give the formal definition of hyperresolution, then explain it.
Definition 8 (Hyperresolution). Let \( R^G_{\text{pred}} = \text{Cls}_1 \), \( H^G_{\text{r}} \land E_1 \Rightarrow C^G_1 \) and \( R^G_{\text{pred}} = \text{Cls}_2, F^G_{\text{pred}} \land H^G_{\text{r}} \land E_2 \Rightarrow C^G_2 \) be two clauses. We rename \( R^G_{\text{r}} \) and \( R^G_{\text{pred}} \) so that they do not use common names for bounds \( M_i \), variables \( x_i \), indices \( i \), functions on indices \( \phi_i \) and set symbols \( J \).

First case: \( F^G_{\text{r}} = \bigwedge_{(i_1, \ldots, i_n) \in \text{pred}_2(p^G_{\text{r}})} (p^G_{\text{r}}) \quad (h \neq 0) \). Let \( \Gamma_1 \vdash R^G_{\text{r}} \) and \( J \) such that \( \Gamma_2 \vdash \{ [1, M_1] \times \cdots \times [1, M_n] \} \). The immersion of \( R^G_{\text{r}} \) into \((i_1, \ldots, i_n) \in J \) is the clause \( \text{imm}(R^G_{\text{r}}; (i_1, \ldots, i_n) \in J) \) obtained by replacing:

1. all free indices \( i \) of \( R^G_{\text{r}} \) with \( \phi_i \in \{i_1, \ldots, i_n\} \), where a new function symbol \( \phi \) is associated to each free index \( i \) of \( R^G_{\text{r}} \);
2. all index terms \( \phi_1(i_1, \ldots, i_k) \) with \( \phi_1(i_1, \ldots, i_n, i_1, \ldots, i_k) \), where a new function symbol \( \phi_1 \) is associated to each function symbol \( \phi \) in \( R^G_{\text{r}} \);
3. all variables \( x_1, \ldots, x_k \) in \( R^G_{\text{r}} \) with \( x_1, \ldots, x_n, i_1, \ldots, i_k \);
4. all conjunctions \( \bigwedge_{(i_1, \ldots, i_k) \in [1, M_1] \times \cdots \times [1, M_k]} \) in \( H^G_{\text{r}} \) and \( E_1 \) with \( \bigwedge_{(i_1, \ldots, i_k, j_1, \ldots, j_k) \in [1, M_1] \times \cdots \times [1, M_k]} \), where a new set symbol \( I \) is associated to each set symbol \( I \) in \( R^G_{\text{r}} \);
5. all conjunctions \( \bigwedge_{(i_1, \ldots, i_k) \in [1, M_1] \times \cdots \times [1, M_k]} \) in \( H^G_{\text{r}} \) and \( E_1 \) with \( \bigwedge_{(i_1, \ldots, i_k) \in [1, M_1] \times \cdots \times [1, M_k]} \) in \( C^G_{\text{r}} \) and \( \bigwedge_{(i_1, \ldots, i_k) \in [1, M_1] \times \cdots \times [1, M_k]} \) in \( C^G_{\text{r}} \).

Let \( \text{imm}(R^G_{\text{r}}; (i_1, \ldots, i_n) \in J) = \text{Cls}_1, H^G_{\text{r}} \land E_1 \Rightarrow \text{pred}_1(p^G_{\text{r}}) \). If \( \text{pred}_1 = \text{pred}_2 \), we define:

\[
R^G_{\text{pred}, \alpha} \land R^G_{\text{pred}, \beta} = \text{Cls}_1 \cup \text{Cls}_2, H^G_{\text{r}} \land E_1 \land E_2 \Rightarrow C^G_{\text{r}}
\]

Let \( I' \) and \( I'' \) be fresh set symbols. Let \( \text{imm}(R^G_{\text{r}}, (i_1, \ldots, i_n) \in I') = \text{Cls}_1, H^G_{\text{r}} \land E_1 \Rightarrow \text{pred}_1(p^G_{\text{r}}) \). If \( \text{pred}_1 = \text{pred}_2 \), we define:

\[
R^G_{\text{pred}, \alpha} \land \text{Cls}_1 \cup \text{Cls}_2, H^G_{\text{r}} \land E_1 \land E_2 \Rightarrow C^G_{\text{r}}
\]

Second case: \( F^G_{\text{r}} = \text{pred}_2(p^G_{\text{r}}) \) (ordinary resolution). If \( \text{pred}_1 = \text{pred}_2 \), we define \( R^G_{\text{pred}, \alpha} \land R^G_{\text{pred}, \beta} = \text{Cls}_1 \cup \text{Cls}_2, H^G_{\text{r}} \land E_1 \land E_2 \Rightarrow C^G_{\text{r}} \). If \( \text{pred}_1 = \text{pred}_2 \), \( R^G_{\text{pred}, \alpha} \land R^G_{\text{pred}, \beta} \) is undefined.

Intuitively, the immersion of \( R^G = H^G \Rightarrow C^G \) into \((i_1, \ldots, i_n) \in J \) corresponds to \( \bigwedge_{(i_1, \ldots, i_n) \in J} H^G_{\text{r}}(i_1, \ldots, i_n) \Rightarrow C^G_{\text{r}(i_1, \ldots, i_n)} \), which represents a combination of a distinct instance of \( R^G \) for each \((i_1, \ldots, i_n) \in J \), as outlined in Section 3.2. To represent distinct instances of \( R^G \), the free indices \( i \), variables \( x \), and functions \( \phi \) use \((i_1, \ldots, i_n) \) as additional indices or arguments. Sets \( I \) should also become functions of the indices \((i_1, \ldots, i_n) \); however, to simplify the syntax of generalized Horn clauses, we avoid sets that depend on indices: when \( \bigwedge_{(i_1, \ldots, i_n) \in I} \text{pred}(p^G) \in H^G \), instead of writing

\[
\bigwedge_{(i_1, \ldots, i_n) \in J} \bigwedge_{(i_1, \ldots, i_n) \in I} \text{pred}(p^G)
\]
This clause will be further simplified by the transformations
given in Section 5.3.

5.3 Simplification of Clauses

We use several additional transformations in order to simplify
clauses. Some of these transformations are easily
adapted from similar transformations already used in
ProVerif. We summarize here the new transformations and
refer the reader to the long version of the paper [8] for details
on all these transformations.

5.3.1 Unification

Instead of performing unification as part of resolution, we
simplify equations a posteriori. The main simplifications are
as follows. (The others are detailed in the long version of the
paper [8].)

- We decompose equations. We replace the equation
  \[ \phi \equiv f(p_G^1, \ldots, p_G^n) \]
  with \[ \phi \equiv p_G^1 \land \ldots \land p_G^n \]. We replace the equation \[ \phi \equiv \varphi \]
  with \[ \varphi \equiv p_H^i \] if \( i \leq h \) and \( i \neq I \), and replace every occurrence of
  \( M' \) in the clause with \( M \). We replace the equation
  \[ \bigwedge_{(i_1, \ldots, i_k) \in J \times [1, M]} \phi(i_1, \ldots, i_k) \]
  with \[ \bigwedge_{(i_1, \ldots, i_k) \in J \times [1, M]} \phi \] and replace every occurrence of
  \( M' \) in the clause with \( M \).

  The equation \[ \bigwedge_{(i_1, \ldots, i_k) \in J \times [1, M]} \phi \]
  can be handled in two ways. The preferred solution is to instantiate \( M \) into the integer \( h \). In this
  instantiation, variables \( x \) that have an index of type
  \( [1, M] \) become \( h \)-variables \( x_1, \ldots, x_M \), and functions
  \( \phi \) that have an argument of type \( [1, M] \) become \( h \)-functions
  \( \phi_1, \ldots, \phi_M \). After instantiation, we apply unification
  again on the obtained clause(s). This solution is applied when the clause contains no function \( \phi \) with a
  result of type \( [1, M] \), no set symbol \( I \) with a type that contains
  \( [1, M] \), and no name with an index of type
  \( [1, M] \). (Otherwise, the instantiated clause may not fit in
  our language of clauses.) When this instantiation cannot be applied, we replace the equation with
  \[ \bigwedge_{(i_1, \ldots, i_k) \in J \times [1, M]} \phi(i_1, \ldots, i_k) \]
  \[ \bigwedge_{(i_1, \ldots, i_k) \in J} x_{i_1, \ldots, i_k} \equiv p_G^i \]
  and add the constraint \( I \cup \{ i \} \subseteq J \times [1, M] \), where
  \( x \) is a fresh variable and \( I_1, \ldots, I_k \) are fresh set
  symbols. Intuitively, the variable \( x_{i_1, \ldots, i_k} \) contains the
  \( i \)-th element of the list. This solution introduces an
  approximation: we remember that the elements of the list are \( p_H^1, \ldots, p_H^{10} \) but we forget their order and their
  number of repetitions.

  The clause is removed when the two sides of the equation
  have different root function symbols. The equation is
  removed when two sides are identical.

- When there is an equation \[ \bigwedge_{(i_1, \ldots, i_k) \in J} x_{i_1, \ldots, i_k} \equiv p_G^i \],
  we replace \( x \) with its value (provided the indices of \( x \)
  match, and no occurrence of \( x \) in \( p_G^i \) can be replaced;
  the latter condition serves to avoid loops).

- When there is an equation \( i = t \) and \( t \) does not occur
  in \( i \), we replace \( i \) with \( t \) and delete the equation.

- When one of the two sides of an equation is a variable
  that does not occur anywhere else in the clause, we
  remove that equation.

Example 3. Applying these transformations, the clause
(8) becomes
  \[ R_G^i \phi^{att} R_G^j = \{ I \cup I' = [1, L] \}, \Lambda_{i \in I} \text{att}(\text{cont}_{\delta(i)}) \land \]
  \[ \Lambda_{i \in I} \text{att}(s\text{ha1}(\text{cont}_{\delta(i)})) \land \]
  \[ \text{att}(\text{sign}(\text{list}(i \leq L, \text{sha1}(\text{cont}_{\delta(i)}), \text{sk}))) \Rightarrow \text{event}(e(r)) \]

Indeed, the equation \[ \Lambda_{i \in I} \text{sha1}(x_i) = \text{sha1}(\text{cont}_{\delta(i)}) \]
first becomes \[ \Lambda_{i \in I} x_i = \text{cont}_{\delta(i)} \], then
\( \text{cont}_{\delta(i)} \) is substituted for
\( x_i \), and the equation is removed.

The unification algorithm is not complete, in that for some
complex equations, it may be unable to use the equations to
simplify the clause or to remove it, even though that would
be sound. This limitation may be lead to false attacks. In
practice, the algorithm is still precise enough to be able the
prove all desired properties of our case studies (Section 6).

5.3.2 Merging of Sets

When a clause uses two disjoint sets \( I \) and \( I' \) in the same
facts and equations (up to renaming), we merge these two
sets into a single set \( I'' \). This transformation is key to obtain
termination of the algorithm: when a clause \( R_G^i \) is obtained
by partial hyperresolution of two clauses \( R_G^j \) and \( R_G^k \), it
can be resolved again with \( R_G^j \) into \( R_G^k \), and so on, which
would yield an infinite loop. However, by merging sets in
\( R_G^i \), we can build a clause that is subsumed by \( R_G^i \), and so
removed, which stops the loop. We illustrate this point on
an example. The clause (9) can be resolved again by partial
hyperresolution with the clause (6). We obtain:

\[ \{ I \cup I' = [1, L], I \cup I' = I \}, \Lambda_{i \in I} \text{att}(\text{cont}_{\delta(i)}) \land \]
\[ \Lambda_{i \in I} \text{att}(s\text{ha1}(\text{cont}_{\delta(i)})) \land \]
\[ \text{att}(\text{sign}(\text{list}(i \leq L, \text{sha1}(\text{cont}_{\delta(i)}), \text{sk}))) \Rightarrow \text{event}(e(r)) \]

which could be resolved again with (6). However, after
replacing the two constraints with \( I \cup I' = [1, L] \), we merge
\( I \) and \( I' \) together into the set \( I_2 \):

\[ \{ I_2 \cup I' = [1, L] \}, \Lambda_{i \in I_2} \text{att}(\text{cont}_{\delta(i)}) \land \]
\[ \Lambda_{i \in I_2} \text{att}(s\text{ha1}(\text{cont}_{\delta(i)})) \land \]
\[ \text{att}(\text{sign}(\text{list}(i \leq L, \text{sha1}(\text{cont}_{\delta(i)}), \text{sk}))) \Rightarrow \text{event}(e(r)) \]

This clause will be removed by the algorithm of Section 5.4,
because it is subsumed by (9), so the loop is avoided.

The merging of sets is not always able to avoid loops that
come from partial hyperresolution, in particular when a fresh
bound \( M \) is created at each partial hyperresolution step.
This limitation is the main new cause of non-termination,
but its impact is limited in practice since the selection
function is tuned to avoid partial hyperresolution when possible.

5.3.3 Soundness of Simplification

Let \( \text{SIMP}(\Gamma \vdash R_G^i) \) be the set of well-typed generalized
Horn clauses obtained by simplifying \( R_G^i \) as described above.
This function is naturally extended to sets of clauses. The
following theorem shows the soundness of \( \text{SIMP} \).
\( \text{SATUR}^G(\mathcal{R}_0^G) = \)

1. \( \mathcal{R}^G \leftarrow \text{ELIM}^G(\text{SIMP}(\mathcal{R}_0^G)) \).

2. Repeat until a fixpoint is reached:
   - for each \( \Gamma \vdash \mathcal{R}^G \in \mathcal{R}^G \) such that \( \text{SEL}^G(\mathcal{R}^G) = \emptyset \),
   - for each \( \Gamma \vdash \mathcal{R}^G \in \mathcal{R}^G \), for each \( \mathcal{P}^G \in \text{SEL}^G(\mathcal{R}^G) \),
     \( \mathcal{R}^G \leftarrow \text{ELIM}^G(\text{SIMP}(\mathcal{R}_\text{Full}^G \cup \mathcal{R}^G)) \).

3. Return \( \{ \Gamma \vdash \mathcal{R}^G \in \mathcal{R}^G \mid \text{SEL}^G(\mathcal{R}^G) = \emptyset \} \).

**Figure 4: New Resolution Algorithm**

**Theorem 4.** Let \( \Gamma \vdash \mathcal{R}^G \) be a well-typed generalized Horn clause. For all environments \( T \) for \( \Gamma \vdash \mathcal{R}^G \), if \( \mathcal{R}^{GT} \) is defined, then there exist a clause \( \Gamma \vdash \mathcal{R}^{GT} \in \text{SIMP}(\Gamma \vdash \mathcal{R}^G) \) and an environment \( T' \) for \( \Gamma \vdash \mathcal{R}^{GT} \) such that \( \mathcal{R}_\text{Full}^{GT} \subseteq \mathcal{R}^{GT} \).

**5.4 Extension of the Resolution Algorithm**

The resolution algorithm for generalized Horn clauses simulates the one for standard Horn clauses. We need to define a generalized selection function \( \text{SEL}^G \).

**Definition 9.** (Generalized selection function). A generalized selection function is a function from generalized Horn clauses to sets of facts, such that \( \text{SEL}^G(\mathcal{C}_n, \mathcal{H}^G \wedge \mathcal{E}) \supseteq \mathcal{H}^G \).

Similarly to the case of standard Horn clauses, a good generalized selection function does not select \( \text{m-event}(p^G) \) nor \( \text{att}(x, ..., y^G) \). Furthermore, resolution is much simpler for facts \( \text{att}(p^G) \) without conjunction than for facts with conjunction, so we select a fact without conjunction when possible. Hence, we use the following selection function:

\[
\text{SEL}^G(\mathcal{C}_n, \mathcal{H}^G \wedge \mathcal{E}) = \begin{cases} 
\{ \text{att}(p^G) \} & \text{if att}(p^G) \in \mathcal{H}^G \text{ for some non-variable } p^G \\
\emptyset & \text{if the first case does not apply and } \\
\mathcal{C} = \text{att}(p^G) \text{ for some non-variable } p^G \\
\{ \text{att}(p^G) \} & \text{if the previous cases do not apply and } \\
\emptyset & \text{otherwise} 
\end{cases}
\]

The resolution algorithm is shown in Figure 4. It mimics the algorithm of Figure 2: \( \text{SATUR}^G(\mathcal{R}_0^G) \) first inserts in \( \mathcal{R}^G \) the clauses in \( \mathcal{R}_0^G \) after elimination of subsumed clauses by \( \text{ELIM}^G \); when \( \mathcal{R}^G \) subsumes \( \Gamma \vdash \mathcal{R}^G \) and both \( \Gamma \vdash \mathcal{R}^G \) and \( \Gamma \vdash \mathcal{R}^{GT} \) are in \( \mathcal{R}^G \), \( \Gamma \vdash \mathcal{R}^{GT} \) is removed by \( \text{ELIM}^G(\mathcal{R}^G) \). Next, the resolution algorithm performs hyperresolution until a fixpoint is reached. Finally, it returns the clauses in \( \mathcal{R}^G \) with no selected hypothesis.

Let \( \mathcal{F}_\text{Full} \) be any set of facts of the form \( \text{m-event}(p) \).

**Theorem 5.** Let \( F \) be a well-typed closed fact and \( \mathcal{R}_0^G \) a set of well-typed generalized Horn clauses. If \( F \) is derivable from \( \mathcal{R}_0^G \cup \mathcal{F}_\text{Full} \), then \( F \) is derivable from \( \text{SATUR}^G(\mathcal{R}_0^G)^G \cup \mathcal{F}_\text{Full} \).

This result shows the soundness of our algorithm. Its proof is adapted from the proof of \( \text{SATUR} \) (Theorem 1) and uses the soundness theorems for resolution, simplification, and subsumption.

To prove that a closed fact \( \text{att}(p^{GT}) \) is not derivable from \( \mathcal{R}_0^G \cup \mathcal{F}_\text{Full} \), we use the following result, where \( \text{att'} \) is a new predicate:

**Corollary 3.** If \( \text{SATUR}^G(\mathcal{R}_0^G) \cup \{ \text{att}(p^G) \} \) contains no clause of the form \( \Gamma \vdash \text{Cts}, \mathcal{H}^G \wedge \mathcal{E} \Rightarrow \text{att}(p^G) \), then, for all environments \( T \), \( \text{att}(p^{GT}) \) is not derivable from \( \mathcal{R}^{GT} \cup \mathcal{F}_\text{Full} \).

This corollary is proved like Corollary 1, and makes it possible to prove that the protocol preserves the secrecy of \( p \) for lists of any length.

**Corollary 4.** Suppose that all clauses of \( \text{SATUR}^G(\mathcal{R}_0^G) \) that conclude \( \text{event}(e(p^G)) \) for some \( p^G \) are of the form \( \Gamma \vdash \text{Cts}, \mathcal{H}^G \wedge \mathcal{E} \Rightarrow \text{event}(e(p^G)) \). Then, for all \( \mathcal{F}_\text{Full} \), for all \( p \), if \( \text{event}(e(p)) \) is derivable from \( \mathcal{R}^{GT} \cup \mathcal{F}_\text{Full} \), then \( \text{m-event}(e'(p)) \in \mathcal{F}_\text{Full} \).

This corollary makes it possible to prove that the protocol satisfies the correspondence “if \( e(x) \) has been executed, then \( e'(x) \) has been executed”.

**6. EXPERIMENTAL RESULTS**

Our algorithm cannot prove authentication for our running example. Indeed, the result of \( \text{SATUR}^G \) contains a clause of the form

\[
\text{att}(r) \wedge \text{m-event}(b(\text{Req})) \wedge \mathcal{E} \Rightarrow \text{event}(e(r))
\]

where \( r \) does not occur in \( \mathcal{E} \). This clause does not satisfy Corollary 4: the event \( e(r) \) may be executed for any \( r \) that the adversary has, even though only \( b(\text{Req}) \) has been executed. This clause corresponds to the wrapping attack described in Section 3.1.

In contrast, our algorithm proves authentication for the corrected version of our running example. The only clause in the result of \( \text{SATUR}^G \) that concludes \( \text{event}(e(p^G)) \) is of the form

\[
\text{m-event}(e'(\text{Req})) \wedge \mathcal{E} \Rightarrow \text{event}(e'(\text{Req}))
\]

Hence, using Corollary 4, we can conclude that, if event \( e(r) \) has been executed, then event \( b(r) \) has been executed as well.

(In our case, the only possible value of \( r \) is the request \( \text{Req} \) of the client.)

In addition to this protocol, we tested our tool on the XML protocols studied in [5]: password digest, password-based signature, X.509 signature, and firewall-based authentication. These are web services security protocols, whose goal is to authenticate a client to a web service. The first two protocols depend on password-based authentication: a password is shared between the user and the server. In the first protocol, a digest of the password, a nonce, and a timestamp is sent by the client to the server. In the second protocol, the client signs its request using a signature key generated from the password. The X.509 signature protocol uses public-key signatures based on X.509 certificates. Finally, the firewall-based authentication protocol uses a SOAP-level firewall in addition to the server and the client. The client uses a password-based signature; the firewall verifies this signature,
and when this authentication succeeds, adds a new firewall header, signed using its X.509 certificate, indicating that it has authenticated the client. For all these protocols, we automatically proved the authentication property proved manually in [5].

Finally, we also modeled the Asokan-Ginzboorg group protocol [1], and proved the secrecy of the session key. This property was already proved in [19], but on a model less precise than ours. For all these examples, our tool terminates in less than 0.5 s on a MacBook Air 1.4 Ghz. The input files for all these case studies can be found at [8].

8. ACKNOWLEDGMENTS

This work was partly supported by the ANR project ProSe (decision number ANR-2010-VERS-004-01). It was partly done while the authors were at École Normale Supérieure, Paris.

9. REFERENCES