Combining ProVerif and Automated Theorem Provers for Security Protocol Verification - System Description

Di Long Li and Alwen Tiu

The Australian National University, Canberra ACT 2600, Australia

Abstract. Symbolic verification of security protocols typically relies on an attacker model called the Dolev-Yao model, which does not model adequately various algebraic properties of cryptographic operators used in many real-world protocols. In this work we describe an integration of a state-of-the-art protocol verifier ProVerif, with automated first order theorem provers (ATP). The integration allows one to model directly algebraic properties of cryptographic operators as a first-order equational theory and the specified protocol can be exported to a first-order logic specification in the standard TPTP format for ATP. An attack on a protocol corresponds to a refutation using the encoded first order clauses. We implement a tool that analyses this refutation and extracts an attack trace from it, and visualises the deduction steps performed by the attacker. We show that the combination of ProVerif and ATP can find attacks that cannot be found by ProVerif when algebraic properties are taken into account in the protocol verification.

1 Introduction

Security protocols are used pervasively in communication networks, such as the SSL/TLS protocol [11] used in secure communication on the web. Designing a security protocol is an error-prone process due to the subtle security requirements and their dependency on the attacker model. Early work on symbolc analysis of protocols has uncovered many flaws in protocol designs, e.g., the classic example of Needham-Schröder's Public Key authentication protocol [20], whose flaw was found through a formal analysis in a model checker [18].

An important part of formal protocol analysis is the definition of the attacker model. A commonly used attacker model is the Dolev-Yao model [12], which assumes, among others, that encryption is perfect, i.e., an attacker will not be able to decrypt a cipher text unless he or she knows the decryption key. While this model has been shown effective in uncovering security flaws in many protocols (see, e.g., [18,25]), it also misses many attacks on protocols that rely on algebraic properties of the operators used in the protocols [9]. One example is the exclusive-or (XOR) operator commonly used in protocols for RFID. As shown in [10], some attacks on these protocols can only be found once the properties of XOR are taken into account, e.g., the associativity, commutativity, and inverse properties.

There are existing protocol verifiers that allow some algebraic properties in the attacker model, such as Maude-NPA [14,15] and Tamarin [19,13]. Maude-NPA, which is based on a rewriting framework, allows a rich set of algebraic properties, specified as equational theories and rewrite systems. These include associativity, commutativity and identity properties, and the finite variant class of equational theories [8]. However, there are some restrictions on how algebraic properties can be specified, and in certain cases [15], specific unification algorithms may need to be added to the prover. Tamarin has recently added support for XOR [13], which allows it to find flaws in the 5G network protocol [3], but just as Maude-NPA, it is not guaranteed to handle theories outside the finite variance class, e.g., XOR with a homomorphic function.

In this system description, we present a new tool, Proverif-ATP, which is a combination of a widely used protocol verifier ProVerif [4] and first-order automated theorem provers (ATP). Our tool allows a user to specify any algebraic properties in the attacker model as an equational theory, without having to modify the prover, or having to prove any particular meta-theory about the equational theory (e.g., whether the equational theory can be represented as a convergent rewrite system, or whether unification modulo the theory is decidable, etc), as this will be handled by the ATPs backend in our tool. Specifically, we made the following contributions:

– We implement an interface from ProVerif to arbitrary ATPs that allows the latter to verify the protocols specified in ProVerif. This is done via a translation from protocol specifications in ProVerif to first-order logic specifications in the TPTP format [24], a common input format accepted by a large number of ATPs. The TPTP output from ProVerif can then be fed into any ATP that accepts the TPTP input format.
– We implement a tool, called Narrator, that interprets the refutation proofs produced by the backend ATP and presents them in a form that is easier for the user to 'debug' a protocol specification. In particular it allows extraction of attack traces from the proofs. An issue with the default encoding from ProVerif to first-order specifications is that it obscures the structures of the original protocol, that makes it difficult to relate the resolution proofs produced by ATP (which can range in the thousands of steps) to attacks on the protocol. To this end, we introduce a tagging mechanism to annotate the encoding of a protocol with information about the protocol steps. This allows us to separate the attacker knowledge that is intercepted from protocol steps from the knowledge deduced by the attacker using (algebraic) properties of cryptographic operators. Narrator visualises the refutation proof as a directed acyclic graph, with different types of nodes color-coded to help the user to spot important steps in the protocol that contribute to the attack.

In Section 2 we give a brief overview of the architecture of ProVerif-ATP. Section 3 illustrates the use of ProVerif-ATP v0.1.0 through a running example of verifying a protocol featuring XOR [7]. Section 4 shows some results in using our tool to verify protocols that are currently out of the scope of ProVerif, and in some instances involving XOR with a homomorphic operator, also out of the

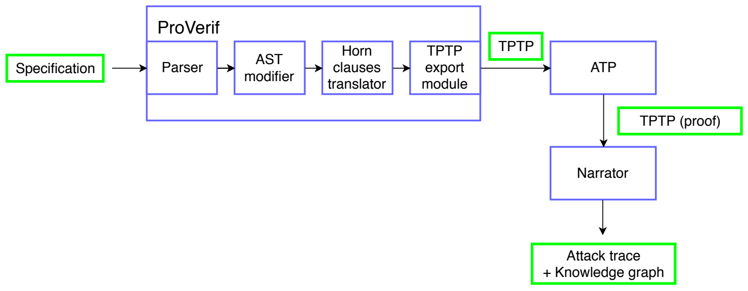


Fig. 1: Overall architecture of ProVerif-ATP

scope of Tamarin. [1] Section 5 concludes the paper and discusses future work. The source code of the ProVerif-ATP tool and the example protocols tested are available online.[2]

2 Overview of the Proverif-ATP tool

The Proverif-ATP tool consists of a version of ProVerif, modified to output protocol specifications in TPTP syntax, and a tool to interpret the resolution proof (in TPTP format [24]) in terms of attack traces and the visualisation of derivations of attacks. Figure 1 shows the architecture of Proverif-ATP. The backend solver can be any ATP, although in this work, we have used mainly Vampire [16]. We explain briefly each component in the following. More details of the implementation are available in the ProVerif-ATP repository.

The components of Proverif-ATP are chained together using a script `pvatp`, which invokes the modified version of ProVerif, translates the ProVerif specification into a TPTP file, invokes the ATP, and generates the HTML files of the Narrator with both the ProVerif specification and Vampire output embedded.

ProVerif. A detailed explanation ProVerif is out of the scope of this system description, and the reader is referred to the ProVerif manual [6] for details. In our tool, we use the specification language based on the typed version of the applied-pi calculus [2]. The specification of algebraic properties of the attacker model can be accommodated within the syntax of ProVerif by simply writing down the equations corresponding to the algebraic properties of the model. So syntactically there is no change required to the specification language for protocols in ProVerif. However, the proof engine of ProVerif does not accept arbitrary equations, even if they are syntactically valid, and it also refuses to translate these equations to first-order clauses. Since we will not be using ProVerif proof engine, the former is not a concern. We explain next how we address the latter.

[1] Some examples which Proverif-ATP can handle but which Tamarin fails are available via https://github.com/darrenldl/ProVerif-ATP/tree/master/related-work/

[2] https://github.com/darrenldl/ProVerif-ATP

Translation to first-order logic. The applied-pi calculus specification in ProVerif is translated to Horn clauses before it is given to its proof engine. This translation was proven correct by Abadi and Blanchet [1,5]. ProVerif has a built-in translator from the internal Horn clause representation of a protocol to a first-order specification in a format accepted by SPASS [27]. However, this translator cannot handle arbitrary equations – which is crucial to our aim. It attempts to perform reduction and syntactic unification to simplify certain terms in a protocol. In the case where the attacker model contains equational theories that cannot be represented as a convergent rewrite system, the simplification performed by ProVerif may fail and some parts of the protocol are not translated. To solve this, we modify the built-in translation of ProVerif to handle specific cases involving equations.

Proverif built-in translation also obscures the relative ordering of protocol steps and it is not obvious how to differentiate among clauses that encode only the attacker model and clauses from different steps of the protocol. To solve this, our tool tags all input (in the latest version) and output actions in the protocol. These tags do not change the meaning of the protocol (they preserve attacks), but it allows our tool (Narrator) to make sense of the refutation proof output by ATP. The tagging is done automatically at the abstract syntax of the applied-pi specification of the protocol, prior to the translation to TPTP.

Currently we only support secrecy queries, i.e., those queries that assert that the attacker knows a certain secret. Queries other than secrecy, such as authentication, need to be translated to an equivalent secrecy query. ProVerif provides a general authentication query, in the form of correspondence assertions [28,6], but their translation to first-order clauses is not yet supported. Since our goal here is to see how well ATP can handle algebraic properties in attacker models, we do not yet attempt to encode correspondence assertions in our translation, as this is orthogonal to the issue of algebraic models.

Note that if an attack is found, an attack trace is produced to serve as a proof certificate, which is to be manually verified in Narrator as described below, so in principle the correctness of the translation is not critical to the workflow.

ATP. The output (in TPTP format) of the translator of ProVerif is input to the ATP. The encoding of protocols as Horn clauses in ProVerif equates an attack on the protocol with provability of an attack goal. So if there is an attack on the protocol, we expect a proof of the attack goal as the output of the ATP (in the TPTP format for proofs). Since the expected input and output of an ATP are in TPTP format, we can in principle use any ATP as the backend prover in our tool, e.g., provers that compete in CASC competitions.[3] In this system description, we use Vampire [16] as the backend prover.

Narrator. The purpose of the Narrator is to make sense of the TPTP output, which can consist of thousands of inference steps. A lot of these steps are derivations of particular messages by the attacker, using the axioms for the attacker

[3] http://tptp.cs.miami.edu/~tptp/CASC/

model. Some crucial information such as steps in the protocol responsible for the attacks, is not easy to spot from such a proof. The backend ATP may employ sophisticated inferences and rewriting of clauses that make it difficult to distinguish even basic information such as which properties of the operator are responsible for the attacks, which steps are involved, and whether a particular information provided by a protocol participant is useful in deriving the attack. Narrator aims to categorise various inference steps to make it easier for a user to spot relevant information from the resolution proof.

Narrator can also produce an attack trace, in the form of sequences of messages exchanged between roles of a protocol with the attacker. This feature is still at an experimental stage, as some crucial information, such as the actual messages that trigger the attack, may not be apparent from the trace. This is due to the fact that some provers (such as Vampire) do not provide the unifiers used in producing the proof, when the option for TPTP output is enforced. Vampire does provide this information in its native proof format (via the option `--proof_extra`) [21]. Narrator currently supports only the TPTP output format for the reason that it is a standard format supported by many theorem provers.

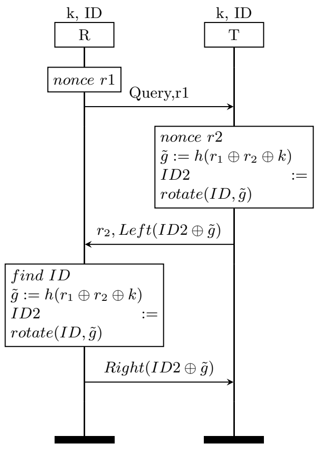
3 Protocol verification with Proverif-ATP: an example

We use the CH07 protocol [7], for mutually authenticating a RFID reader and a RFID tag, as an example. As shown in [10], the protocol fails to guarantee the aliveness property [17] of the tag to the reader - the reader completed a run of the protocol believing to have been communicating with a tag, while no tag was present. Figure 2a shows the CH07 protocol specification in a message sequence chart. In the figure, R denotes an instance of an RFID reader, and T denotes an instance of an RFID tag. The constant **k** denotes the secret key shared by R and T, and **ID** denotes an identifier of T (e.g., serial number). There are various bitwise manipulation functions used in the protocol: for simplicity, we shall treat these as uninterpreted function symbols except for $\oplus$, which denotes XOR.

R initiates the protocol by sending a query message containing a *nonce* (a fresh random number) $r1$ to **T**. **T** generates a nonce $r2$, and computes $\tilde{g}$, which is used to rotate ID to obtain $ID2$. **T** then sends out $r2$ along with the left half of $ID2$. **R** looks up the ID of **T**, and computes the same $\tilde{g}$ and $ID2$, then sends out the right half of $ID2$ as response to **T**. It is implicit that **R** and **T** check all received messages against the self-computed copies, namely $Left(ID2 \oplus \tilde{g})$ sent from **T** to **R**, and $Right(ID2, \tilde{g})$ sent from **R** to **T**. At the end of a succesful run of the protocol, **R** is convinced of the identity of **T** and vice versa.

The attack shown in [10] consists of two sessions. In the first session, the attacker observes a completed run between R and a tag T, and records all messages exchanged. Then using these messages, the attacker impersonates T in the second session. This attack relies on the properties of XOR to craft a special value of r_2 in the second session (see [10] for details).

Figure 2b shows a fragment of the ProVerif specification of the protocol. Note that the properties of XOR are encoded as equations (line starting with `equation`



k, ID
R
k, ID
T
$nonce\ r1$
Query,r1
$nonce\ r2$
$\tilde{g} := h(r_1 \oplus r_2 \oplus k)$
$ID2 := rotate(ID, \tilde{g})$
$r_2, Left(ID2 \oplus \tilde{g})$
$find\ ID$
$\tilde{g} := h(r_1 \oplus r_2 \oplus k)$
$ID2 := rotate(ID, \tilde{g})$
$Right(ID2 \oplus \tilde{g})$

(a) CH07 protocol

```
equation forall x:bitstring, y:bitstring, z:bitstring;
     xor(x, xor(y, z)) = xor(xor(x, y), z).
equation forall x:bitstring, y:bitstring; xor(x, y) =
     xor(y, x).
equation forall x:bitstring; xor(x, ZERO) = x.
equation forall x:bitstring; xor(x, x) = ZERO.

free objective : bitstring [private].
query attacker(objective).

let R =
  new r1:bitstring;
  out(c, (QUERY, r1));
  in(c, (r2 : bitstring, left_xor_ID2_g : bitstring));
  let g     = h(xor(xor(r1, r2), k)) in
  let ID2   = rotate(ID, g) in
  let left  = split_L(xor(ID2, g)) in
  let right = split_R(xor(ID2, g)) in
  if left = left_xor_ID2_g then (
    out(c, right);
    out(c, objective)
  ).

let sess_1 =
  new r1_s1:bitstring;
  out(c, r1_s1);
  new r2_s1:bitstring;
  let g_s1     = h(xor(xor(r1_s1, r2_s1), k)) in
  let ID2_s1   = rotate(ID, g_s1) in
  let left_s1  = split_L(xor(ID2_s1, g_s1)) in
  let right_s1 = split_R(xor(ID2_s1, g_s1)) in
  out(c, (r2_s1, left_s1));
  out(c, right_s1).

process
  sess_1 | R
```

(b) A part of ProVerif encoding

Fig. 2: CH07 specifications

in the figure). Since we cannot model authentication directly in Proverif-ATP, we reformulate the problem as a secrecy problem. For this, we assume that the attacker has obtained messages from the first session (process `sess_1` in Figure 2b). For the second session onwards, we modify the protocol so that the reader will output a secret (`objective` in Figure 2b) after a successful authentication. To ensure that we capture the authentication property properly, we need to make sure that this secret is not accidentally output as a result of a legitmate interaction with **T**; this is modelled by simply removing the tag T from the protocol runs in the second session.

Before the protocol specification is translated to TPTP, the input/output actions in the protocol are tagged with constants denoting certain information, such as the principal performing those actions, and the steps within the protocol. The tagged protocol is then passed on to the translator unit to produce the TPTP specification of the protocol. The following is an example of a fragment of the TPTP file produced by the translation.

```
fof(ax226, axiom, pred_attacker(tuple_2(tuple_2(constr_QUERY, name_r1), constr_R_STEP_1))).

![VAR_R2_293] : (pred_attacker(tuple_2(VAR_R2_293, constr_split_L(constr_xor(constr_rotate(
     name_ID, constr_h(constr_xor(constr_xor(name_r1, VAR_R2_293), name_k))), constr_h(
     constr_xor(constr_xor(name_r1, VAR_R2_293), name_k)))))) => pred_attacker(tuple_2(
     name_objective, constr_R_STEP_3)))).
```

The encoding uses a predicate 'pred_attacker' to encode the attacker knowledge. The first clause above, for example, shows the knowledge obtained from an output step with no dependency on previous inputs. The second clause shows an example of an interactive protocol step: if the right condition (input) is present, then an output is sent (hence would be known to the attacker).

Analysis of Refutation Proof in Narrator Narrator has three major modes, knowledge graph mode, and two ProVerif code + attack trace examination modes (one shows the raw ProVerif code, one shows the pretty-printed ProVerif code). We first examine the structure of the attack through the attack trace generated by Narrator, then we utilise the knowledge graph and explanation mechanism to further analyse aspects of the attack.

```
1.    sess_1.1    sess_1 -> I : r1_s1

2.    sess_1.2    sess_1 -> I : tuple_2(r2_s1,split_L(xor(rotate(ID,h(xor(xor(r1_s1,r2_s1),k)
     )),h(xor(xor(r1_s1,r2_s1),k)))))

3.    R.1         R -> I : tuple_2(QUERY,r1)

4.    R.3         I -> R : tuple_2(X28,split_L(xor(rotate(ID,h(xor(xor(r1,X28),k))),h(xor(xor
     (r1,X28),k)))))

5.    R.3         R -> I : objective
```

Fig. 3: Narrator attack trace output

In the attack trace in Figure 3, `I` denotes the intruder/attacker, `R` denotes the reader, and `sess_1` denotes the knowledge of a previous session as we have specified. Observe that the attacker is able to construct the appropriate message to send to `R` at step 4 (`R.3 I -> R`) for `R` to complete its execution and output objective at step 3. This matches the typical challenge and response structure - the reader `R` challenges the tag (attacker in this case) to give the appropriate message to solicit a response (message `objective`). The crucial step is step 4, which indicates that the reader somehow accepts the response from the attacker as legitimate. However, in this case, the backend ATP (Vampire) does not produce a ground instance of the message being sent; instead, there is a variable `X28` whose value is supposed to be the nonce r_2 crafted by the attacker. Vampire can produce the substitutions produced at each step of inferences, but this feature is currently supported only in its native proof format, and not for the TPTP output. When we examine Vampire native proof output, we do see the exact term for X28 produced, which matches the attack mentioned in [10]. We have attempted running Vampire in the 'question-answering' mode, which would produce answer substitutions in TPTP format; but in that mode AVATAR must be disabled, which makes Vampire unable to find the attack for this protocol.

However, using Narrator we can still trace how the message at step 4 is constructed by the attacker, by observing the relationships between different

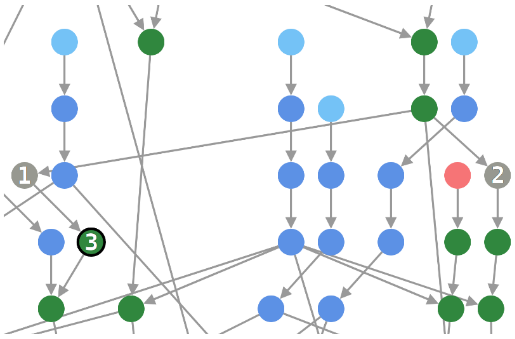


Fig. 4: Partial view of knowledge graph with manual numbering

nodes in the refutation proof. We represent the refutation proof as a directed acyclic graph, which we call *knowledge graph*. The nodes in the graph are color-coded to distinguish different categories of formulas as follows:

`Unsure`	Orange	`NegatedGoal`	Dark gray
`Axiom`	Light blue	`Contradiction`	Dark gray
`InitialKnowledge`	Light green	`ProtocolStep`	Light red
`Rewriting`	Dark blue	`InteractiveProtocolStep`	Dark red
`Knowledge`	Dark green	`Alias`	Light gray
`Goal`	Dark gray		

In this example, we focus on unconditional protocol steps (light red) and interactive protocol steps (dark red) nodes. Unconditional protocol steps are outputs that happen unconditionally, and interactive steps are outputs which depend on input from the attacker.

Suppose we want to trace how the objective in step 5 in Figure 3 is triggered. That step is ascribed to `R.3` – which denotes that this is the third step of the reader R (an information derived from our tagging mechanism). From the trace we know that `R.3` is an interactive step, so it would appear as a dark-red node in the knowledge graph. In this case, there is only one such node. Tracing down the path from this node, we see it branches into node 1 and 2 in Figure 4. The two nodes are introduced by Vampire's AVATAR architecture, which utilises a SAT or an SMT solver for improved capability [26]. By clicking on the two gray nodes individually, we can see the following two formulas

```
    spl0_2 <=> ! [X0] : ~attacker(tuple_2(X0,split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(
k,xor(r1,X0))))))))
    spl0_0 <=> attacker(tuple_2(objective,R_STEP_3))
```

The first formula corresponds to the message sent from `I` to `R` at step `R.3`. To see how this formula is constructed, we visit the next node from this grey node (node 3 in Figure 4) and choose the "Explain construction of chain" option in the menu. Figure 5 contains the partial copy of the explanation shown, where `X0` is equivalent to `X28` in Figure 3. See the full version of this paper in the project GitHub repository for the full explanation.

```
From
  step R.1
  axiom ! [X17,X18] : attacker(tuple_2(X17,X18)) => attacker(X17)
  axiom ! [X19,X20] : attacker(tuple_2(X19,X20)) => attacker(X20)
attacker knows
  r1
...
From
  axiom ! [X2,X3] : xor(X2,X3) = xor(X3,X2)
  axiom ! [X4,X5,X6] : xor(X4,xor(X5,X6)) = xor(xor(X4,X5),X6)
  axiom ! [X8,X9] : (attacker(X9) & attacker(X8)) => attacker(xor(X8,X9))
  r1_s1
  xor(r1,r2_s1)
attacker knows
  xor(r1,xor(r1_s1,r2_s1))

From
  step sess_1.2
  axiom ! [X17,X18] : attacker(tuple_2(X17,X18)) => attacker(X17)
  axiom ! [X19,X20] : attacker(tuple_2(X19,X20)) => attacker(X20)
  axiom ! [X2,X3] : xor(X2,X3) = xor(X3,X2)
attacker learns
  split_L(xor(h(xor(k,xor(r1_s1,r2_s1))),rotate(ID,h(xor(k,xor(r1_s1,r2_s1))))))

Attacker rewrites
  split_L(xor(h(xor(k,X2)),rotate(ID,h(xor(k,X2)))))
to
  split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(k,xor(r1,X0))))))

  xor(r1,X2)
to
  X0

From
  axiom ! [X15,X16] : (attacker(X16) & attacker(X15)) => attacker(tuple_2(X15,X16))
  split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(k,xor(r1,X0))))))
  X0
attacker knows
  tuple_2(X0,split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(k,xor(r1,X0)))))))
```

Fig. 5: Sample explanation given by Narrator (. . . indicates skip)

4 Evaluation

We evaluate the effectiveness of ProVerif-ATP on a number of protocol specifications under various attacker models featuring algebraic operators, such as, XOR, Abelian group operators, homomorphic encryption and associative pairing; see [9] for details of these properties. In most cases ProVerif simply fails because the equations modelling these properties are outside the scope of what it can handle. Table 1 shows the list of protocols verified and the results of the verification. We encode different attack queries as separate entries, thus there may be multiple entries for one protocol. See the full version of this paper on GitHub repository for more details of these protocols. Most of the protocols shown can be verified using Vampire version 4.2.2 as the ATP backend. In four protocols, involving XOR, Vampire fails due to out-of-memory error (indicated by 'MEM' in the table), and in one case, Vampire found a model (indicated by 'SAT' in the table) – meaning there is no attack under the corresponding attacker model, which is confirmed in ProVerif as well. The experiments were done in a dedicated virtual server, featuring a 8 core 3.6GHz processor and 32GB

Table 1: List of protocols tested with ProVerif-ATP

Protocol	**ProVerif**	**Vampire**	**Vampire time (seconds)**
LAK06 [10]	×	✓	0.102
SM08 [10]	×	✓	84.684
LD07 [10]	×	✓	0.015
OTYT06 [10]	×	✓	0.008
CH07 [10]	×	✓	84.244
KCLL06 [10]	×	× (MEM)	-
Bull Authentication Protocol [22]	×	× (MEM)	-
Shamir-Rivest-Adleman Three Pass [9]	✓	✓	0.013
DH [9]	✓	✓	0.015
WEP [9]	×	✓	0.182
Salary Sum [9]	×	✓	0.571
NSPK (attack 1) [18]	✓	✓	3.151
NSPK (attack 2) [18]	✓	✓	2.629
NSLPK modified (attack 1) [9]	×	✓	0.032
NSLPK modified (attack 2) [9]	×	✓	0.034
NSLPK with ECB (attack 1) [9]	×	✓	2.060
NSLPK with ECB (attack 2) [9]	×	✓	2.300
NSLPK with XOR (attack 1) [23]	×	× (MEM)	-
NSLPK with XOR (attack 2) [23]	×	× (MEM)	-
NS with CBC (attack 1) [9]	✓	✓	0.070
NS with CBC (attack 2) [9]	No attack	SAT	-
NS with CBC (attack 3) [9]	✓	✓	0.087
Denning-Sacco Symmetric Key with CBC [9]	✓	✓	0.027

RAM. Vampire was set to run in default mode, with a 24 hours timeout and 30.9GB memory limit. In the default mode Vampire only uses one core.

5 Conclusion and future work

Our preliminary tests suggest that the integration of ProVerif and ATP is useful as it improves the class of protocols one can verify. While we have not done a thorough comparison with other protocol verifiers, we do note that Tamarin can handle more security properties than Proverif-ATP (which is restricted to secrecy), but ProVerif-ATP can handle some cases where Tamarin cannot. Our aim is to show how an existing protocol verifier can benefit from ATPs to extend the coverage of its analysis with minimal efforts, ie., without coming up with dedicated procedures to solve equational reasoning needed in protocol analysis, as such we see our work as complementary to other dedicated protocol verifiers rather than their competitors or replacements.

A crucial part of our work is making the output of ATP intelligible for users, for which the unifiers used in the derivation of an attack would be very helpful. The current specification of the TPTP proof output format does not require unifiers to be included in the output proofs, something we hope will change in the future. As future work, we intend to compile a set of benchmark problems derived from protocol analysis (with algebraic operators) and propose its inclusion in the future CASC competition, to encourage developers of ATPs to improve their provers to solve this class of problems. Another direction is to also investigate the encoding of hyperproperties, such as non-inteference and equivalence.

References

1. Abadi, M., Blanchet, B.: Analyzing security protocols with secrecy types and logic programs. J. ACM **52**(1), 102–146 (Jan 2005). https://doi.org/10.1145/1044731.1044735, http://doi.acm.org/10.1145/1044731.1044735
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 104–115. POPL '01, ACM, New York, NY, USA (2001). https://doi.org/10.1145/360204.360213, http://doi.acm.org/10.1145/360204.360213
3. Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., Stettler, V.: A formal analysis of 5g authentication. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1383–1396. CCS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243846
4. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001. pp. 82–96 (June 2001). https://doi.org/10.1109/CSFW.2001.930138
5. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and proverif. Foundations and Trends® in Privacy and Security **1**(1-2), 1–135 (2016). https://doi.org/10.1561/3300000004, http://dx.doi.org/10.1561/3300000004
6. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: ProVerif 2.00:automatic cryptographic protocol verifier, user manual and tutorial. Tech. rep. (2018)
7. Chien, H.Y., Huang, C.W.: A lightweight rfid protocol using substring. In: Kuo, T.W., Sha, E., Guo, M., Yang, L.T., Shao, Z. (eds.) Embedded and Ubiquitous Computing. pp. 422–431. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77092-3_37
8. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) Term Rewriting and Applications, pp. 294–307. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_22
9. Cortier, V., Delaune, S., Lafourcade, P.: A survey of algebraic properties used in cryptographic protocols. Journal of Computer Security **14**(1), 1–43 (Feb 2006). https://doi.org/10.3233/jcs-2006-14101
10. van Deursen, T., Radomirovic, S.: Attacks on RFID protocols. IACR Cryptology ePrint Archive **2008**, 310 (2008), http://eprint.iacr.org/2008/310
11. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.2. Tech. rep. (Aug 2008). https://doi.org/10.17487/rfc5246
12. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2), 198–208 (March 1983). https://doi.org/10.1109/TIT.1983.1056650
13. Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R.: Automated unbounded verification of stateful cryptographic protocols with exclusive or. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 359–373 (July 2018). https://doi.org/10.1109/CSF.2018.00033
14. Escobar, S., Meadows, C., Meseguer, J.: Maude-npa: Cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures. pp. 1–50. Springer Berlin Heidelberg, Berlin,

Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_1, https://doi.org/10.1007/978-3-642-03829-7_1

15. Escobar, S., Meadows, C.A., Meseguer, J.: Maude-npa, Version 3.1. Tech. rep. (2017)
16. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
17. Lowe, G.: A hierarchy of authentication specifications. In: Proceedings 10th Computer Security Foundations Workshop. pp. 31–43 (June 1997). https://doi.org/10.1109/CSFW.1997.596782
18. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using fdr. In: Margaria, T., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 147–166. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_43
19. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The tamarin prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 696–701. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_48
20. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (Dec 1978). https://doi.org/10.1145/359657.359659, http://doi.acm.org/10.1145/359657.359659
21. Reger, G.: Better proof output for vampire. In: Kovacs, L., Voronkov, A. (eds.) Vampire 2016. Proceedings of the 3rd Vampire Workshop. EPiC Series in Computing, vol. 44, pp. 46–60. EasyChair (2017). https://doi.org/10.29007/5dmz, https://easychair.org/publications/paper/1DlL
22. Ryan, P., Schneider, S.: An attack on a recursive authentication protocol a cautionary tale. Information Processing Letters **65**(1), 7–10 (1998). https://doi.org/10.1016/S0020-0190(97)00180-4, http://www.sciencedirect.com/science/article/pii/S0020019097001804
23. Steel, G.: Deduction with xor constraints in security api modelling. In: Nieuwenhuis, R. (ed.) Automated Deduction – CADE-20, pp. 322–336. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11532231_24
24. Sutcliffe, G.: The tptp problem library and associated infrastructure. Journal of Automated Reasoning **59**(4), 483–502 (Dec 2017). https://doi.org/10.1007/s10817-017-9407-7, https://doi.org/10.1007/s10817-017-9407-7
25. Viganò, L.: Automated security protocol analysis with the avispa tool. Electronic Notes in Theoretical Computer Science **155**, 61 – 86 (2006). https://doi.org/10.1016/j.entcs.2005.11.052, http://www.sciencedirect.com/science/article/pii/S1571066106001897, proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)
26. Voronkov, A.: Avatar: The architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 696–710. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46
27. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: Spass version 3.5. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22, pp. 140–145. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10
28. Woo, T.Y.C., Lam, S.S.: Authentication for distributed systems. Computer **25**(1), 39–52 (Jan 1992). https://doi.org/10.1109/2.108052

Appendix 1 Full explanation by Narrator for CH07

```
From
  step R.1
  axiom ! [X17,X18] : attacker(tuple_2(X17,X18)) => attacker(X17)
  axiom ! [X19,X20] : attacker(tuple_2(X19,X20)) => attacker(X20)
attacker knows
  r1

From
  step sess_1.2
  axiom ! [X17,X18] : attacker(tuple_2(X17,X18)) => attacker(X17)
  axiom ! [X2,X3] : xor(X2,X3) = xor(X3,X2)
attacker learns
  r2_s1

From
  axiom ! [X8,X9] : (attacker(X9) & attacker(X8)) => attacker(xor(X8,X9))
  r2_s1
  r1
attacker knows
  xor(r1,r2_s1)

From
  step sess_1.1
  axiom ! [X17,X18] : attacker(tuple_2(X17,X18)) => attacker(X17)
attacker learns
  r1_s1

From
  axiom ! [X2,X3] : xor(X2,X3) = xor(X3,X2)
  axiom ! [X4,X5,X6] : xor(X4,xor(X5,X6)) = xor(xor(X4,X5),X6)
  axiom ! [X8,X9] : (attacker(X9) & attacker(X8)) => attacker(xor(X8,X9))
  r1_s1
  xor(r1,r2_s1)
attacker knows
  xor(r1,xor(r1_s1,r2_s1))

From
  step sess_1.2
  axiom ! [X17,X18] : attacker(tuple_2(X17,X18)) => attacker(X17)
  axiom ! [X19,X20] : attacker(tuple_2(X19,X20)) => attacker(X20)
  axiom ! [X2,X3] : xor(X2,X3) = xor(X3,X2)
attacker learns
  split_L(xor(h(xor(k,xor(r1_s1,r2_s1))),rotate(ID,h(xor(k,xor(r1_s1,r2_s1))))))

Attacker rewrites
  split_L(xor(h(xor(k,X2)),rotate(ID,h(xor(k,X2)))))
to
  split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(k,xor(r1,X0))))))

  xor(r1,X2)
to
  X0

From
  axiom ! [X15,X16] : (attacker(X16) & attacker(X15)) => attacker(tuple_2(X15,X16))
  split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(k,xor(r1,X0))))))
  X0
attacker knows
  tuple_2(X0,split_L(xor(h(xor(k,xor(r1,X0))),rotate(ID,h(xor(k,xor(r1,X0)))))))
```

Appendix 2 Benchmark details

All of the following files reside in the GitHub repository's `examples/` directory. The other related files for a given protocol specification named `protocol.pv` follow the naming convention below.

`protocol.pv.reprinted`	This is a reexported copy of the `.pv` generated by ProVerif via the `-log-pv-only` flag. This file is mainly for storing the specification with a standardised formatting convention.
`protocol.pv.processed`	This is a reexprted copy of the `.pv` file after AST modification, generated by the modified ProVerif via the `-log-pv` flag. This file is mainly used to verify that the AST modification is done correctly, specifically the AST modification can be examined easily by comparing the file with `protocol.pv.reprinted`. This represents the actual AST used by ProVerif in the translation.
`protocol.p`	This is the exported TPTP file generated by the modified ProVerif with flags `-out tptp` and `-tag-out`. This is the file to be solved by ATP.
`protocol.solver_log`	This is the Vampire output used by Narrator
`protocol.bench_log`	This is the Vampire output used to gather the timing statistics. It is the same as `protocol.solver_log` but with statistics of Vampire attached.

2.1 LAK06

`File`	`LAK06-tag-auth.pv`
`Protocol description`	RFID protocol for mutual reader and tag authentication
`Setup description`	A challenge and response setup similar to CH07's is used - attacker is provided with knowledge of a previous legitimate session, and is required to provide the appropriate messages for the RFID reader to complete execution
`Attack type`	Aliveness property of tag guaranteed to reader is violated
`References`	[10]

2.2 SM08

`File`	`SM08-tag-auth.pv`
`Protocol description`	RFID protocol for mutual reader and tag authentication

`Setup description`	A challenge and response setup similar to CH07's is used - attacker is provided with knowledge of a previous legitimate session, and is required to provide the appropriate messages for the RFID reader to complete
`Attack type`	Aliveness property of tag guaranteed to reader is violated
`References`	[10]

2.3 LD07

`File`	`LD07-tag-auth.pv`
`Protocol description`	RFID protocol for mutual reader and tag authentication
`Setup description`	A challenge and response setup similar to CH07's is used - attacker is provided with knowledge of a previous legitimate session, and is required to provide the appropriate messages for the RFID reader to complete execution
`Attack type`	Aliveness property of tag guaranteed to reader is violated
`References`	[10]

2.4 OTYT06

`File`	`OTYT06-reader-auth.pv`
`Protocol description`	RFID protocol for mutual reader and tag authentication
`Setup description`	A challenge and response setup similar to CH07's is used - attacker is provided with knowledge of a previous legitimate session, and is required to provide the appropriate messages for the RFID tag to complete execution
`Attack type`	Aliveness property of reader guaranteed to tag is violated
`References`	[10]

2.5 CH07

`File`	`CH07-tag-auth.pv`
`Protocol description`	RFID protocol for mutual reader and tag authentication
`Setup description`	A challenge and response setup is used - attacker is provided with knowledge of a previous legitimate session, and is required to provide the appropriate messages for the RFID reader to finish execution. This is detailed in Section 3.
`Attack type`	Aliveness property of tag guaranteed to reader is violated
`References`	[10]

2.6 KCLL06

`File`	`KCLL06-reader-auth.pv`
`Protocol description`	RFID protocol for mutual reader and tag authentication
`Setup description`	A challenge and response setup similar to CH07's is used - attacker is provided with knowledge of a previous legitimate session, and is required to provide the appropriate messages for the RFID tag to complete execution
`Attack type`	Aliveness propertiy of reader guaranteed to tag is violated
`References`	[10]

2.7 Bull Authentication Protocol

`File`	`bull.pv`
`Protocol description`	Protocol for establishing fresh keys between a fixed number of principals, including a server. Each participant only has key with its immediate neighbors in the chain, and key with the server.
`Setup description`	The same setup in the original specification is used, with participants `A`, `B`, `C` and server `S`. Among the clients, `C` is compromised, leaking key materials (e.g. `Nc, Kbc`) to the attacker, then attacker is asked to decrypt `objective`, which is encrypted by key `Kab`.
`Attack type`	Secrecy violation
`References`	[22]

2.8 Shamir-Rivest-Adleman Three Pass

`File`	`Shamir-Rivest-Adleman-Three-Pass.pv`
`Protocol description`	Protocol that allows two parties to exchange a secret without prior known secrets
`Setup description`	The setup follows the original specification, and attacker is asked to decrypt `objective`, which is encrypted by a secret message
`Attack type`	Secrecy violation
`References`	[9]

2.9 DH

`File`	`DH.pv`
`Protocol description`	Diffie-Hellman Key Exchange Protocol allows two parties to establish a common secret
`Setup description`	The setup follows the original specification where authentication is not in place. Attacker is asked decrypt `objective`, which is encrypted by a secret message.
`Attack type`	Secrecy violation
`References`	[9]

2.10 WEP

`File`	`WEP-secrecy.pv`
`Protocol description`	Wired Equivalent Privacy Protocol protects data in transit
`Setup description`	The setup assumes a specific scenario where two messages are encrypted with same initial vector and key, and attacker knows the first message and its encrypted version. The attacker is then asked to decrypt the second message.
`Attack type`	Secrecy violation
`References`	[9]

2.11 Salary Sum

`File`	`Salary-Sum.pv`
`Protocol description`	Protocol aims to allow a group of participants to compute the sum of their salary without divulging any individual salary
`Setup description`	The setup uses four principals `A`, `B`, `C`, `D`, and they have knowledge of each others' public keys, however, authentication is not in place. Attacker is asked to obtain `Sa`, salary of `A`.
`Attack type`	Secrecy violation
`References`	[9]

2.12 NSPK (attack 1)

`File`	`NSPK-agree-A-to-B.pv`
`Protocol description`	Needham-Schroeder Public Key Protocol aims to authenticate two parties via use of shared public key knowledge
`Setup description`	The setup consists of two parties and a trusted key server used for registering public keys and distributing public keys. In this encoding, `A` is restricted to communicate only with the attacker, and `B` is restricted to communicate only with `A`. The attacker is asked to make `B` finish its execution, completing an impersonation attack.
`Attack type`	Violation of weak agreement property of `A` to `B`
`References`	[18]

2.13 NSPK (attack 2)

`File`	`NSPK-agree-A-to-B-secrecy.pv`
`Protocol description`	Same as above
`Setup description`	Same as above, with the addition that attacker is asked to also obtain nonce `Nb`
`Attack type`	Violation of weak agreement property of `A` to `B`, and secrecy property
`References`	[18]

2.14 NSLPK modified (attack 1)

`File`	`NSLPK-modified-agree-A-to-B.pv`
`Protocol description`	Needham-Schroder-Lowe Public Key Protocol aims to authenticate two parties. This variant introduces a flaw by using an associative pairing function.
`Setup description`	Same as the one for NSPK in Section 2.12
`Attack type`	Violation of weak agreement property of `A` to `B`
`References`	[9]

2.15 NSLPK modified (attack 2)

`File`	`NSLPK-modified-agree-A-to-B-secrecy.pv`
`Protocol description`	Same as above
`Setup description`	Same as above, with the addition that attacker is asked to also obtain nonce `Nb`
`Attack type`	Violation of weak agreement property of `A` to `B`, and secrecy property
`References`	[9]

2.16 NSLPK with ECB (attack 1)

`File`	`NSLPK-ECB-agree-A-to-B.pv`
`Protocol description`	Needham-Schroder-Lowe Public Key Protocol aims to authenticate two parties. This variant introduces a flaw by using ECB mode encryption.
`Setup description`	Same as the one for NSPK in Section 2.12
`Attack type`	Violation of weak agreement property of `A` to `B`
`References`	[9]

2.17 NSLPK with ECB (attack 2)

`File`	`NSLPK-ECB-agree-A-to-B-secrecy.pv`
`Protocol description`	Same as above
`Setup description`	Same as above, with the addition that attacker is asked to also obtain nonce `Nb`
`Attack type`	Violation of weak agreement property of `A` to `B`, secrecy property
`References`	[9]

2.18 NSLPK with XOR (attack 1)

`File`	`NSLPK-XOR-agree-A-to-B.pv`
`Protocol description`	Needham-Schroder-Lowe Public Key Protocol aims to authenticate two parties. This variant introduces a flaw by using XOR as part of the pairing function.
`Setup description`	Same as the one for NSPK in Section 2.12.
`Attack type`	Violation of weak agreement property of `A` to `B`
`References`	[23]

2.19 NSLPK with XOR (attack 2)

`File`	`NSLPK-XOR-agree-A-to-B-secrecy.pv`
`Protocol description`	Same as above
`Setup description`	Same as above, with the addition that attacker is asked to also obtain nonce `Nb`
`Attack type`	Violation of weak agreement property of `A` to `B`, and secrecy property
`References`	[23]

2.20 NS with CBC (attack 1)

`File`	`NS-CBC-alive.pv`
`Protocol description`	Needham-Schroeder Symmetric Key Protocol aims to establish a fresh key between the two parties and provide mutual authentication between them. This variant introduces an additional attack where `A` can be tricked into accepting a publicly known nonce `Na` as a key shared with `B` (see Section 2.22 for this attack).
`Setup description`	The setup consists of two parties and a trusted key server used for generating and distributing the symmetric keys. The attacker is asked to impersonate `B` in the second session.
`Attack type`	Violation of aliveness property of `B` to `A` (in second session)
`References`	[9]

2.21 NS with CBC (attack 2)

`File`	`NS-CBC-secrecy.pv`
`Protocol description`	Same as above
`Setup description`	The setup consists of two parties and a trusted key server used for generating and distributing the symmetric keys, and `A` and `B` both know who to communicate with. The attacker is asked to obtain key `Kab`.
`Attack type`	Violation of secrecy
`References`	[9]

2.22 NS with CBC (attack 3)

`File`	`NS-CBC-alive-known-key.pv`
`Protocol description`	Same as above
`Setup description`	The setup consists of two parties and a trusted key server used for generating and distributing the symmetric keys. The attacker is asked to decrypt a message encrypted by `Kab'` (key established in second session),
`Attack type`	Violation of aliveness property of `B` to `A` (in second session), and secrecy
`References`	[9]

2.23 Danning-Sacco Symmetric Key with CBC

`File`	`Denning-Sacco-CBC-alive.pv`
`Protocol description`	This protocol is an amended version of Needham-Schroeder Symmetric Key Protocol with timestamps, with the same purpose as the original one
`Setup description`	The setup consists of `B` and trusted key server `S`. The attacker is asked to impersonate `A`.
`Attack type`	Violation of aliveness property of A to B
`References`	[9]