Automating Interactive Protocol Verification

Lassaad Cheikhrouhou, Andreas Nonnengart, Werner Stephan,
Frank Koob[a], and Georg Rock[b]

German Research Center for Artificial Intelligence, DFKI GmbH
[a]Federal Office for Information Security, BSI [b]PROSTEP IMP GmbH

Abstract. Showing the *absence* of security hazards in cryptographic protocols is of major interest in the area of protocol security analysis. Standard model checking techniques - despite their advantages of being both fast and automatic - serve as mere debuggers that allow the user at best to detect security risks if they exist at all. In general they are not able to guarantee that *all* such potential hazards can be found, though. A full verification usually involves induction and therefore can hardly be fully automatic. Therefore the definition and application of suitable heuristics has turned out to become a central necessity. This paper describes how we attack this problem with the help of the Verification Support Environment (VSE) and how we nevertheless arrive at a high degree of automation.

1 Introduction

Protocols that (try to) provide certain security properties in open network environments by using cryptographic primitives like encryption, signing, and hashing, play a crucial role in many emerging application scenarios. Machine readable travel documents are a typical example of a technology whose common acceptance heavily depends on how far security properties like confidentiality and authenticity can really be *guaranteed* by their designers [1]. However, if carried out on an informal basis, the analysis of cryptographic protocols has turned out to be fairly error prone. Justifiably so, the *formal* specification and analysis of cryptographic protocols has become one of the fastest growing research areas, although with a hardly manageable variety of different approaches.

Formal protocol models usually incorporate the exchanged messages between protocol participants. These messages are sent via an open network which is accessible by an attacker who can intercept, and even change or forge (some of) these messages. The capabilities of the attacker as well as of honest protocol participants are restricted by the assumption that encryption and decryption is impossible, unless the correct cryptographic key is at hand. In this setting we speak of the so-called Dolev-Yao model as introduced (rather implicitly) in [2].

Formal analysis approaches to protocol analysis can roughly be divided into two main categories: *model checking* and *(interactive) verification*. Whereas model checking has the big advantage that it is both fast and automatic, it in general relies on fixed scenarios with a given number of protocol participants

and a given number of protocol runs. They thus serve well as a *systematic* way of debugging security protocols. The general validity of security properties, however, can only be guaranteed under certain circumstances.

(Interactive) verification methods, on the other hand, not only try to find existing security hazards; they are also able to show the *non-existence* of such risks. A typical example for such an approach can be found in Paulson's inductive verification approach [3]. There neither the number of participants nor the length of protocol traces is restricted in any way. Obviously, reasoning about such (inductively defined) *event traces* and the knowledge gained by an attacker (e. g. while eavesdropping on exchanged messages) heavily relies on *induction* proofs. This seriously complicates matters and one can hardly expect to get a fully automatic verification system to solve such problems.

In this paper we focus on the techniques we developed to adopt Paulson's approach for the Verification Support Environment (VSE). VSE is a kind of case tool for formal software development that closely combines a front end for specification (including refinement) and the management of structured developments with an interactive theorem prover [4]. We emphasize on our attempt to lower the burden of interactive proof generation to an extent that makes the VSE framework for protocol verification applicable within the limited time frames of commercial developments. In order to do this, we exploited the specific struture of the inductive proofs in our domain and implemented several proof heuristics to carry out most of the proof tasks automatically. This allowed us to reach an automation degree of about 90%.

2 BAC - A Cryptographic Protocol Example

One of the commercial protocols that were verified in VSE was the Extended Access Control (EAC) protocol. It includes (in 12 steps) three related phases in the inspection procedure of an electronic passport (ePass A) by a terminal (B): Basic Access Control (BAC), Chip Authentication and Terminal Authentication. In this paper we emphasize on the steps of the first phase to examplify our work.

2.1 The Protocol and its Properties

The inspection procedure starts with the BAC protocol to protect the data on the ePass chip from unauthorized access and to guarantee (for the terminal) that this chip corresponds to the machine readable zone (MRZ) of the inspected ePass. The protocol comprises the following steps:

1. $B \longleftarrow A: \quad K_{BAC}(A)$
2. $B \longrightarrow A: \quad B, Get\_Random\_Number$
3. $A \longrightarrow B: \quad r_A$
4. $B \longrightarrow A: \quad \{r_B, r_A, K_B\}_{K_{BAC}(A)}$
5. $A \longrightarrow B: \quad \{r_A, r_B, K_A\}_{K_{BAC}(A)}$

This protocol realizes a mutual authentication between A and B, if both participants have access to the basic access key $K_{BAC}(A)$. The key is stored on

the chip of A and can be computed by a terminal B from the MRZ. Such a BAC run can only be successful if B has physical (optical) access on the ePass for it has to read the MRZ in order to be able to compute the correct key. This is represented in the first protocol step by the right-to-left arrow. It indicates that B is both the active part of the communication and the receiver of the message. After the second step which signals that the terminal determined the key $K_{BAC}(A)$, the participants A and B authenticate each other in two interleaved challenge-responses using the random numbers (nonces) r_A and r_B. Here, $\{M\}_K$ denotes the cipher of a message M by a key K. The ciphers in the fourth and the fifth message are obtained by symmetric encryption. Encrypting as well as decrypting these messages obviously requires to possess the key $K_{BAC}(A)$.

In addition to the used nonces, the fresh key materials (session keys) K_B and K_A are exchanged. These will be used for the computation of the session keys that are needed for secure messaging in the subsequent phase of the inspection procedure.

The main properties of the BAC protocol are:

- BAC1: The ePass A authenticates the terminal B and agrees with B on the session key K_B.[1]
- BAC2: The terminal B authenticates the ePass A and agrees with A on the session keys K_B and K_A.
- BAC3: The session keys K_A and K_B are confidential.

2.2 Specification of Protocol Runs and Properties

The above properties are defined and verified for all possible traces of the BAC protocol. The set of these traces is associated with a corresponding predicate (here BAC) which holds for the empty trace ϵ and is inductively defined for every extension of a BAC-trace tr with an admissible event ev:

$$BAC(ev\#tr) \Leftrightarrow (BAC(tr) \wedge (Reads1(ev,tr) \vee Says2(ev,tr) \vee Says3(ev,tr) \\ \vee Says4(ev,tr) \vee Says5(ev,tr) \vee Gets\_event(ev,tr) \vee Fake\_event(ev,tr)))$$

This inductive definition covers all the alternatives to extend an arbitrary BAC-trace by a new event: five protocol steps ($Reads1, Says2, \ldots, Says5$), message reception ($Gets\_event$), and sending a faked message by the attacker ($Fake\_event$). The definitions of these predicates determine the structure of the corresponding events and express the corresponding conditions depending on the extended trace tr and the event parameters.

The definitions of the gets-case and the fake-case are *generic*, since they do not depend on the considered protocol. For instance, a send-event $Says(spy, ag, m)$ from the attacker (spy) to any agent ag may occur in the fake-case, when

[1] Note that these first five steps of the protocol do not allow A to deduce that B receives the session key K_A, since this key is sent to B in the last protocol step. This is proven in the complete version in subsequent steps where B confirms the reception of the session key K_A.

the message m can be constructed from the attacker's extended knowledge. This condition is expressed by $isSynth(m, analz(spies(tr)))$, where $spies(tr)$ contains the messages that are collected by the attacker and $analz$ extends this set by message decomposition and decryption.

The definitions of the remaining cases are protocol specific, since they encode the conditions of the corresponding protocol steps according to the protocol rules. These conditions express the occurrence of previous events, the existence of certain (initial) information and the freshness of newly generated information. For instance, the definition of the second step requires that the extended trace tr contains an event $Reads(B, A, K_{BAC}(A))$ which represents the access by B of the initial information $K_{BAC}(A)$ of A via a secure channel.

The protocol properties are formalized for arbitrary traces tr for which the predicate BAC holds. For instance, confidentiality properties are expressed with the help of the attacker's extended knowledge. Basically, a message m is confidential if it is not contained in and can not be constructed from this knowledge. Consider for instance the formalization of the confidentiality property BAC3:

$$\begin{aligned}&(BAC(tr) \wedge \{r_1, r_2, aK\}_{K_{BAC}(A)} \in parts(spies(tr)) \wedge \\ &\ \neg bad(A) \wedge \forall ag : (Reads(ag, A, K_{BAC}(A)) \in tr \Rightarrow \neg bad(ag))) \\ &\Rightarrow aK \notin analz(spies(tr))\end{aligned}$$

This secrecy property is defined for every session key aK which occurs in a message of the form $\{r_1, r_2, aK\}_{K_{BAC}(A)}$. It holds for the session key K_B of step 4, as well as for the session key K_A of step 5, assuming that A is not compromised, i.e. $bad(A)$[2] does not hold, and its basic access key is not accessible to the attacker. In contrast to $analz$, the function $parts$ includes the encrypted sub-messages even if the keys needed for decryption can not be obtained from the given message set. It is used, especially in authenticity properties, to express the occurrence of sub-messages including the encrypted ones.

2.3 Specification and Verification Support in VSE

The formal specification of cryptographic protocols in VSE is supported by

- a library of VSE theories that define the basic data types for messages, events etc. and the analysis operators, e.g., $analz$, which are needed to formalize protocols (from a given class) in the VSE specification language (VSE-SL),
- and a more user friendly specification language called VSE-CAPSL (as an extension of CAPSL [5]) with an automatic translation to VSE-SL.

The translation of VSE-CAPSL specifications leads to definitions of the protocol traces in the above style and formalizes some of the stated protocol properties. Additionally, certain lemmata like possibility, regularity, forwarding and unicity lemmata, that give rise to the generic proof structure as discussed in [3] are generated.

[2] The attacker is bad ($bad(spy)$) and possesses the knowledge of any other bad agent.

The main protocol properties and the lemmata generated by the system serve as proof obligations for the VSE prover. For each proof obligation a complete proof has to be constructed in the (VSE) sequent calculus. Besides the (basic and derived) inference rules, including the application of lemmata and axioms, powerful simplification routines can be selected by the user for this purpose. The number of the (protocol specific) proof obligations and the size of their proofs definitely shows that the inductive approach will only be successful in real world environments if the burden of user interaction is lowered drastically. Therefore we have concentrated on a heuristic proof search that applies higher-level knowledge available from a systematic analysis of the given class of proof obligations. This approach will be described in a detailed way in the next section.

3 Proof Automation by Heuristics

Heuristics are intended as a means to incrementally provide support for the user of an interactive system. Our approach is bottom up in the sense that starting with routine tasks whose automatization just saves some clicks we proceed by building heuristics that cover more complex proof decisions using the lower level ones as primitives. In particular the heuristics do not constitute a fixed, closed proof procedure: They can freely be called on certain subtasks to obtain goals whose further reduction requires a decision from the user before the next heuristics can continue to generate partial proofs for the new subgoals. Since full automation can hardly be reached, this kind of *mixed initiative* is our ultimate goal, although some of our most recent high-level heuristics were even able to prove certain lemmata completely on their own.

3.1 Protocol Properties and their Proofs

The main properties of cryptographic protocols that we are interested in are *confidentiality* and *authenticity*. *Sensitive* data, like session keys or nonces that are often used for the generation of new data items that were not used in the current protocol run, like for example new session keys, have to be protected against a malicious attacker.

Authentication properties are often formulated from the perspective of a particular participant. In the protocol presented in section 2.1 the authentication property BAC2 is formulated from the perspective of the terminal. Authentication proofs as the one for the BAC2 property are often based on *authenticity* properties of certain messages. The authenticity of such a message is used to identify the sender of that message.

In addition to the top-level properties we have to prove so called *structuring lemmata*. These are generated automatically by the system and used by the heuristics. Their proof uses the same basic scheme that is used for confidentiality and authenticity.

This general scheme for structural induction on the traces of a particular protocol is more or less straightforward. The heuristics discussed in the following

implement an application specific refinement of this scheme. This refinement takes into account both the type of the proof goals and the available lemmata and axioms.

The organization of the independent building blocks in the general scheme described below was left to the user in the beginning of our work. Meanwhile more complex tasks can be achieved automatically by high-level heuristics that implement more global proof plans.

3.2 The Top-Level Proof Scheme

All inductive proofs of protocol properties are structured into the following (proof-) tasks. For each task there is a collection of heuristics that are potentially applicable in these situations.

1. Set up a proof by structural induction on traces.
2. Handle the base case.
3. Handle the step cases:
   (a) Reduce certain formulas to *negative* assumptions in the induction hypothesis.
   (b) Add information about individual protocol steps.
   (c) Reduce the remaining differences and apply the induction hypothesis.

An inductive proof about traces of a given protocol is initialized by a heuristic (`traceInd`) which selects the induction variable representing the protocol trace and reduces the proof goals representing the base case and the step case to a simplified normal form.

Base Case: The proof goals of the base case will contain assumptions that contradict properties of the empty trace ϵ. These assumptions are searched for by a heuristic that applies appropriately instantiated lemmata (axioms) to close the proof goals by contradiction. For instance, an assumption of the form $ev \in \epsilon$, stating that the event ev is part of the empty trace, obviously contradicts an axiom about traces. This is detected by the corresponding heuristic (`nullEvent`). Similar, but more complex heuristics for this task are for example based on the fact that nonces and session keys do not exist before starting a protocol run.

Step Case: Like in all other mechanized induction systems in the step case(s) we try to reduce the given goal(s) to a situation where the inductive hypothesis can be applied.

For this purpose the VSE strategy provides heuristics from three groups.

Negative assumptions are used to impose certain restrictions on the set of traces under consideration. For example, we might be interested only in traces without certain events. In the BAC protocol most of the properties do not include the optical reading of the considered basic access key by a compromised agent. This is formulated by the negative assumption $\forall ag : (Reads(ag, A, K_{BAC}(A)) \in tr \Rightarrow \neg bad(ag))$. The difference between this formula and the corresponding assumption in the step case, i.e. in $\forall ag : ((Reads(ag, A, K_{BAC}(A)) \in (ev\#tr)) \Rightarrow$

$\neg bad(ag)$), can be eliminated without knowing any details about the event ev added to the trace (induction variable) tr. This kind of difference reduction is performed by a heuristic (`eventNotInTrace`) which treats negative assumptions about the membership of events in traces.

In the next step(s) we exploit the assumptions under which a certain event ev can be added to a trace tr. This is achieved by a heuristic called `protocolSteps`. First a *case split* is carried out according to the protocol rules as formalized in section 2.2. Next the conditions for each particular extension is added to the goals which are simplified to a normal form.

The remaining differences stem from formulas of the form (i) $ev' \in (ev\#tr)$, (ii) $m \in analz(spies(ev\#tr))$, and (iii) $m \in parts(spies(ev\#tr))$ in the goals. For the three cases there are heuristics that start the difference reduction. For $analz$ and $parts$ they rely on *symbolic evaluation*.

Sometimes the induction hypothesis can be applied directly in the resulting goals. In other cases it is necessary to apply appropriate *structuring lemmata* and to make use of specific *goal assumptions*. These goal assumptions can originate from the definition of the corresponding protocol step (by `protocolSteps` in (b)) or during the symbolic evaluation itself.

While the proof tasks (a) and (b) are carried out in a canonical way the last proof task is much more complex and, for the time being, typically requires user interaction. The treatment of certain subgoals resulting from symbolic evaluation of $analz$ or $parts$ often involves proof decisions, like:

- Which goal assumptions should be considered?
- Which structuring lemmata have to be applied?
- Do we need a *new* structuring lemma?

However, also in those cases where *some* user interaction is necessary, heuristics are available to continue (and complete) the proof afterwards.

First steps towards an automatization of complex proof decisions in subtasks, tasks, and even complete proofs were made by designing high-level heuristics that combine the ones discussed so far.

3.3 Definition of Heuristics

Heuristics achieve their proof tasks by expanding the proof tree starting with a given goal. The basic steps that are carried out are the same as those the user might select (see section 2.3). In addition, heuristics might backtrack and choose to continue with another (open) subgoal.

Heuristics in VSE, like tactics in other systems, determine algorithmically the execution of the mentioned steps. In addition to the purely logical information, heuristics in VSE have access to local (goals) and global *control information* that is read to decide about the next step to be performed and updated to influence the further execution. The additional information slots are used to model the internal control flow of a single heuristic as well as to organize the composition of heuristics. The existing VSE module for writing and executing

heuristics was extended to meet the requirements of semi-automatic protocol verification. This concerns in particular those parts of the heuristics that depend on certain theories.

4 Results and Future Work

We developed about 25 heuristics which we used in the formal verification of several real world protocols, e.g., different versions of the EAC protocol (12 protocol steps, 41 properties) [6] and of a Chip-card-based Biometric Identification (CBI) protocol (15 protocol steps, 28 properties) [7]. Typical proofs of these properties consist of 1500 to 2000 proof nodes (steps) represented in a VSE proof tree. A proof of this magnitude would require at least 400 user interactions if the heuristics were not utilized. The heuristics thus allowed us to save up to 90% of user interactions in average.

So far the developed higher-level heuristics are especially tailored for the proof of confidentiality and possibility properties. For this kind of properties an even better automatization degree (up to full automatic proof generation) is reached. In order to increase the automation degree in the verification of the other protocol properties, i.e. of authenticity properties and other structuring lemmata, we plan to define more heuristics: higher-level heuristics tailored for these proofs and probably lower level heuristics when needed.

Although several proof structuring lemmata are formulated automatically, certain proof attempts force the user to define additional lemmata that depend on the remaining open proof goals. We therefore also plan to develop suitable lemma speculation heuristics.

References

1. Advanced Security Mechanisms for Machine Readable Travel Documents – Extended Access Control (EAC) – Version 1.11 Technical Guideline TR-03110, Federal Office for Information Security (BSI)
2. D. Dolev and A. Yao. On the security of public-key protocols. IEEE Transactions on Information Theory, 2(29), 1983
3. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
4. D. Hutter, G. Rock, J. H. Siekmann, W. Stephan, and R. Vogt. Formal Software Development in the Verification Support Environment (VSE). In B. Manaris J. Etheredge, editor, Proceedings of the FLAIRS-2000. AAAI-Press, 2000.
5. G. Denker and J. Millen and H. Rueß. The CAPSL Integrated Protocol Environment. SRI Technical Report SRI-CSL-2000-02, October 2000.
6. Formal Verification of the Cryptographic Protocols for Extended Access Control on Machine Readable Travel Documents. Technical Report, German Research Center for Artificial Intelligence and Federal Office for Information Security
7. L. Cheikhrouhou, G. Rock, W. Stephan, M. Schwan, and G. Lassmann. Verifying a chip-card-based biometric identification protocol in VSE. In J. Górski (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 42–56. Springer, Heidelberg (2006)