

# Code Security Analysis of a Biometric Authentication System Using Automated Theorem Provers\*

Jan Jürjens

Software & Systems Engineering, Dep. of Informatics, TU Munich, Germany  
<http://www4.in.tum.de/~juerjens>

## Abstract

*Understanding the security goals provided by cryptographic protocol implementations is known to be difficult, since security requirements such as secrecy, integrity and authenticity of data are notoriously hard to establish, especially in the context of cryptographic interactions. A lot of research has been devoted to developing formal techniques to analyze abstract specifications of cryptographic protocols. Less attention has been paid to the analysis of cryptoprotocol implementations, for which a formal link to specifications is often not available. In this paper, we apply an approach to determine security goals provided by a C implementation to a biometric authentication system in development in an industrial project. Our approach is based on control flow graphs and automated theorem provers for first-order logic.*

## 1 Introduction

While a significant amount of research has been directed to develop formal techniques to analyze abstract specifications of cryptographic protocols, few attempts have been made to apply the results developed in that setting to the analysis of cryptoprotocol implementations. Even if specifications exist for these implementations, and even if these had been analyzed formally, there is usually no guarantee that the implementation actually

conforms to the specification. An example for a protocol whose design had been formally verified for security and whose implementation was later found to contain a weakness with respect to its use of cryptographic algorithms can be found in [RS98]. Even in software projects where specification techniques are used, often changes in the code that become necessary during the implementation phase because of dynamically changing requirements are not reflected in the specifications. In this paper, we therefore propose an approach to determine security goals provided by a protocol implementation on the source-code level.

Our approach uses automated theorem provers (ATPs) for first-order logic. These are not only automatic, but also quite efficient and powerful, because of the efficient proof procedures implemented in these tools and because security requirements can be formalized straightforwardly in first-order logic (FOL). The C code gives rise to a control flow graph in which the cryptographic operations are represented as abstract functions. The control flow graph is translated to formulas in first-order logic with equality. Together with a logical formalization of the security requirements, they are then given as input into any ATP (such as e-SETHEO [SW00]) supporting the TPTP input notation, which is a standard input notation for automated theorem provers (ATPs). If the analysis reveals that there could be an attack against the protocol, an attack generation script written in Prolog is generated from the C code. A tool for our approach is available over a web-interface and as open-source [sec].

The contribution of the current paper consists in applying our approach to a biometric

---

\*This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author(s).

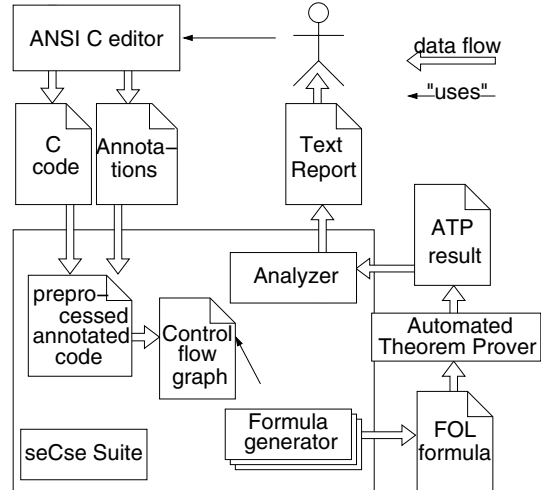
authentication protocol currently in development by an industrial partner in a joint research and development project. Note that this is not just a matter of using well-understood concepts and existing components, and of applying these to a particular problem: Although there is a lot of work on verifying abstract specifications of cryptographic protocol using formal methods, the verification of implementations of cryptographic protocols using first-order logic is a new research topic. Because of the security problems which may be created at the transition from an abstract specification to an implementation, as mentioned above, one cannot naively apply the specification-based verification techniques to the source-code level.

It is not our goal to provide an automated full formal verification of C code. Instead, our goal is to increase understanding of the security properties of cryptoprotocol implementations in a way as automated as possible to facilitate use in an industrial environment. Because of the abstractions used, the approach may produce false alarms (which however have not surfaced yet in practical examples). Also, for space restrictions we cannot consider features such as pointer arithmetic in our presentation here (we essentially follow the approach in [CKY03] there). We do not consider casts, and expressions are assumed to be well-typed. Loops are only investigated through a bounded number of rounds (which is a classical approach in automated software verification). Also, our focus here is on high-level security properties such as secrecy and authenticity, and not on detecting low-level security flaws such as buffer-overflow attacks.

## 2 Code Analysis

We define the translation of security protocol implementations to first-order logic formulas which allows a security analysis of the source code using automated first-order logic theorem provers. The corresponding tool-flow is shown in Fig. 1.

The analysis approach presented here works with the well-known Dolev-Yao adversary model for security analysis [DY83, Mea91, Gol03]. The idea is that an adversary can read messages sent over the network and collect them in his knowledge set. The adversary can merge and extract messages in the knowl-



**Figure 1. Tool-flow of the seCse analysis suite**

edge set and can delete or insert messages on the communication links. The security requirements can then be formalized using this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary.

We explain the transformation from the control flow graph generated from the C program to first-order logic, which is given as input to the automated theorem prover. For space restrictions, we restrict our explanation to the analysis for secrecy of data. The idea here is to use a predicate  $\text{knows}$  which defines a bound on the knowledge an adversary may obtain by reading, deleting and inserting messages on vulnerable communication lines (such as the Internet) in interaction with the protocol participants. Precisely,  $\text{knows}(E)$  means that the adversary may get to know  $E$  during the execution of the protocol. For any data value  $s$  supposed to remain confidential, one thus has to check whether one can derive  $\text{knows}(s)$ .

From a logical point of view, this means that one considers a term algebra generated from ground data such as variables, keys, nonces and other data using symbolic operations including the ones in Fig. 2. There, the symbols  $E$ ,  $E'$ , and  $E''$  denote terms inductively constructed in this way. These symbolic operations are the abstract versions of the cryptographic algorithms used in the code. Generating keys and random values is formalized by introducing new variables representing the keys and random values. For keys

- $\text{enc}(E, E')$  (encryption)
- $\text{dec}(E, E')$  (decryption)
- $\text{hash}(E)$  (hashing)
- $\text{sign}(E, E')$  (signing)
- $\text{ver}(E, E', E'')$  (verification of signature)
- $\text{kgen}(E)$  (key generation)
- $\text{inv}(E)$  (inverse key)
- $\text{conc}(E, E')$  (concatenation)
- $\text{head}(E)$  and  $\text{tail}(E)$  (head and tail of concat.)

**Figure 2. Crypto Operations**

and random values that are supposed to be freshly generated for each round of the protocol, one thus has a formula parameterized over these variables, which is then closed by forall-quantification. In that term algebra, one defines the equations  $\text{dec}(\text{enc}(E, K), \text{inv}(K)) = E$  and  $\text{ver}(\text{sign}(E, \text{inv}(K)), K, E) = \text{true}$  for all terms  $E, K$ , and the usual laws regarding concatenation,  $\text{head}()$ , and  $\text{tail}()$ . This abstract information is automatically generated from the concrete source code.

The set of predicates defined to hold for a given program is defined as follows. For each publicly known expression  $E$ , the statement  $\text{knows}(E)$  is derived. To model the fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows, including the use of cryptographic operations, formulas are generated for these operations for which some examples are given in Fig. 3. We use the TPTP notation for the first-order logic formulas [SS01], which is the input notation for many automated theorem provers including the one we use (e-SETHEO [SW00]). Here  $\&$  means logical conjunction and  $![E1, E2]$  forall-quantification over  $E1, E2$ .

We now define how a control flow graph generated from a C program gives rise to a logical formula characterizing the interaction between the adversary and the protocol participants (technically, this is realized via the export format GDL of the aiCall tool [Abs04]).

**Step 1** We observe that the graph can be transformed to consist of transitions of the form  $\text{trans}(\text{state}, \text{inpattern}, \text{condition}, \text{action}, \text{truestate})$ , where  $\text{inpattern}$  is empty and  $\text{condition}$  equals  $\text{true}$  where they are not needed, and where  $\text{action}$  is a logical expression of the form  $\text{localvar} = \text{value}$  respectively  $\text{outpattern}$  in case of a local assignment resp. output command

(and leaving it empty if not needed). If needed, there may be additionally another transition with the negation of the given condition.

**Step 2** Now assume that the source code gives rise to a transition  $\text{TR1} = \text{trans}(s1, i1, c1, a1, t1)$  such that there is a second transition  $\text{TR2} = \text{trans}(s2, i2, c2, a2, t2)$  where  $s2 = t1$ . If there is no such transition  $\text{TR2}$ , we define  $\text{TR2} = \text{trans}(t1, [], \text{true}, [], t1)$  to simplify our presentation, where  $[]$  is the empty input or output pattern and  $\text{true}$  is the boolean condition. Suppose that  $c1$  is of the form  $\text{cond}(\text{arg}_1, \dots, \text{arg}_n)$ . For  $i1$ , we define  $\bar{i}1 = \text{knows}(i1)$  in case  $i1$  is non-empty and otherwise  $\bar{i}1 = \text{true}$ . For  $a1$ , we define  $\bar{a}1 = a1$  in case  $a1$  is of the form  $\text{localvar} = \text{value}$  and  $\bar{a}1 = \text{knows}(\text{outpattern})$  in case  $a1 = \text{outpattern}$  (and  $\bar{a}1 = \text{true}$  in case  $a1$  is empty). Then for  $\text{TR1}$  we define the following predicate:

$$\text{PRED}(\text{TR1}) \equiv \bar{i}1 \& c1 \Rightarrow \bar{a}1 \& \text{PRED}(\text{TR2}) \quad (1)$$

The formula formalizes the fact that, if the adversary knows an expression he can assign to the variable  $i1$  such that the condition  $c1$  holds, then this implies that  $\bar{a}1$  will hold according to the protocol, which means that either the equation  $\text{localvar} = \text{value}$  holds in case of an assignment, or the adversary gets to know  $\text{outpattern}$ , in case it is sent out in  $a1$ . Also then the predicate for the succeeding transition  $\text{TR2}$  will hold.

To construct the recursive definition above, we assume that the control flow graph is finite and cycle-free. Since in general there may be unbounded loops in the C program (although in the case of cryptographic protocols, these are not so prevalent because the emphasis is on interaction rather than computation), this is achieved in an approximate way by fixing a natural number  $n$  (supplied by the user of the approach) and unfolding all cycles up to the transition path length  $n$ . The resulting logical formula is closed by forall-quantification over all free variables contained.

**Step 3** The formulas defined above are written into the TPTP file as axioms. This means that the theorem prover will take these formulas as given. The security requirement to be checked is written into the TPTP file

```

input_formula(construct_message_1,axiom,(
! [E1,E2] :
  ( (   knows(E1)
    & knows(E2) )
  => (   knows(conc(E1, E2))
    & knows(enc(E1, E2))
    & knows(sign(E1, E2)) ) ) ).

input_formula(construct_message_2,axiom,(
! [E1,E2] :
  (   knows(conc(E1, E2))
  => (   knows(E1)
    & knows(E2) ) ) ).

```

**Figure 3. Some general crypto axioms**

as a conjecture. For example, this could be `knows(secret)` in case the secrecy of the value `secret` is to be checked. For authenticity properties, one needs to insert additional correspondence assertions in the formulas defined above at places which are bound by the generation and verification of an authentication certificate. The theorem prover will then check whether the conjecture is derivable from the axioms. In the case of secrecy, the result is interpreted as follows: If `knows(secret)` can be derived from the axioms, this means that the adversary may potentially get to know `secret`. If the theorem prover returns that it is not possible to derive `knows(secret)` from the axioms, this means that the adversary will not get `secret`.

Note that the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities). This means that one will find all possible attacks, but one may also encounter “false alarms”. However, this has not so far happened with practical examples, and the treatment turns out to be rather efficient. Note also that due to the undecidability of first-order logic, one may not always be able to establish automatically that the adversary does *not* get to know a certain data value, but the theorem prover may execute without termination or may break up because resources are exceeded. In our practical applications of our method, this limitation has, however, not yet become observable.

**Step 4** In case the result is that there may be an attack, in order to fix the flaw in the code, it would be helpful to retrieve the attack trace. Since theorem provers such as e-SETHEO are highly optimized for performance by using abstract derivations, it is not trivial to extract this information. Therefore, we also implemented a tool which transforms the logical formulas explained above to Prolog. The translation from the logical formulas to Pro-

log is quite direct, so that no discrepancies can be expected. While the analysis in Prolog is not useful to establish whether there is an attack in the first place (because it is in order of magnitudes slower than using e-SETHEO and in general there are termination problems with its depth-first search algorithm), Prolog works fine in the case where one already knows that there is an attack, and it only needs to be shown explicitly (because it explicitly assigned values to variables during its search, which can then be queried).

### 3 Case-Study

We applied our methods and tools in an industrial application project with a major German company. The goal of the project is the correct development of a security-critical biometric authentication system which is supposed to control access to a protected resource, for example by opening a door or letting someone log into a computer system. In this system, a user carries his biometric reference data on a personal smart-card. To gain access, he inserts the smart-card in the card reader and delivers a fresh biometric sample at the biometric sensor, for example a finger-print reader. Since the communication links between the host system (containing the bio-sensor), the card reader, and the smart-card are physically vulnerable, the system needs to make use of a cryptographic protocol to protect this communication. Because the correct design of such protocols and the correct use within the surrounding system is very difficult, our method was chosen to support the development of the biometric authentication system. Here, we report on experiences we gained when applying our code analysis approach to a prototypical implementation which we constructed from the specification provided by our industrial partner.

To generate a FOL formula to be analyzed, one needs to consider the level of security provided by the physical layer of the system, and

formulate security goals on the execution of the system and on the protection of particular data values. Then the security of the protocol is analyzed using the automated tool support described in the previous section. This is done with respect to the threat model which is derived from the information about the physical security of the system and the security goals, as explained in the previous section.

We give a description of the authentication protocol, which on the request of our industrial partner has to remain relatively generic. A high-level specification of the control flow in the biometric authentication system is given in Fig. 4. The system components are the smart-card, the host system and the biosensor. The smart-card is personalized for each user. To prevent an attack where an attacker simply repeatedly tries to match a forged biometric sample, for example, using an artificial finger, with a forged or stolen smart-card, the protocol contains misuse counters which are decreased from an initial value of 3 to 0, when the card will be disabled. The data stored on the card includes the card identifier, the misuse counters, the biometric reference data, a corresponding signature and a key shared with the host system. The host system determines whether the identity of the user can be verified given the biometric reference data on the card and he should thus be granted access.

In the first messages, the card is reset and asked for its ID which the host stores in a variable. Then, the host retrieves a freshly generated random number from the card. The following messages perform a bidirectional authentication between the host and the smart-card using random numbers that are exchanged and the identities of the card and host, encrypted with a key shared between the card and the host. This is done only if the first card misuse counter – decremented with each unsuccessful attempts of the card to authenticate to the host – is greater than 0. After successful authentication, the misuse counter is set back to the default value 3. The biometric authentication is then started with the exchange of the next messages. In the following messages, the session key is generated at the smart-card and sent to the host. The confidentiality and integrity of the communication is protected using encryption and MAC using a shared key. Next, the current value of the sec-

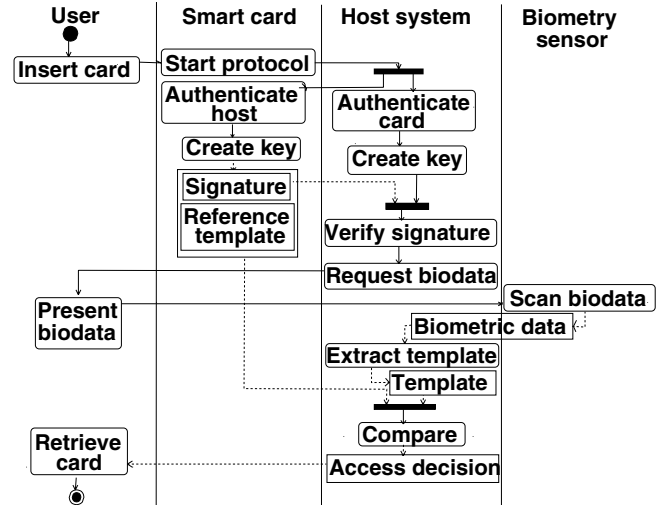


Figure 4. Biometric authentication

ond misuse counter is retrieved from the smart-card. If it is larger than 0, it is decremented at the host and the decremented value is sent back to the smart-card. The smart-card is then queried again for the new value of its second misuse counter to see whether it actually stored the decremented value. The integrity of the last messages starting with the retrieval of the second misuse counter from the card (including the message names) is protected using MACs using the session key. Then, the biometric reference data is retrieved from the smart-card which is signed with the private administrator key that was created when personalizing the smart-card. After that, the current data is requested from the biosensor and compared with the reference data. The user has up to three attempts to present a biometric sample which is accepted as valid (defined as the degree of coincidence between sample and reference being above the given threshold). If the biometric match is successful, the second misuse counter is sent to the default value and the session closed.

For space restrictions, we can only show the main call graph node of the smart-card side of the protocol in Fig. 5 and one of the message exchanges in Fig. 6.

## 4 Security Analysis

The threat scenario which we consider here is that the adversary somehow obtains possession of a legitimate smart-card and can manipulate the communication link between the

smart-card reader and the host system, since it is not assumed to be physically secure.

We have to assume that the adversary can use different (sequential or parallel) executions of the protocol in his attack (with the same or different smart-cards or hosts). This can be achieved by parameterizing the FOL formula generated from the protocol description using a session parameter, using variables for the smart-card, biosensor, and host names, and closing the open formula obtained with a for-all quantification over these parameters.

We have to verify that the protocol provides the intended security guarantees, in particular, that the misuse counter indeed registers any failed attempt to present a false biometric sample to the biosensor. Here we focus on this security requirement, which turned out to be particularly interesting in the case of the given protocol.

We note that each possible instantiation of the message argument variables in the formula corresponds to one execution of the protocol, assuming that each protocol participant accepts only one copy of a given protocol message per protocol execution (and ignores a second message with the same message name).

Note that an automated theorem prover such as SPASS or E-SETHEO considers every possible model satisfying the given axioms to see whether it satisfies the given conjecture, not only the quotients of the free algebra under the formula (as Prolog does). This means that in the models considered, additional properties not following from the given axioms may hold. In the case of cryptographic protocols, this may mean that a secret key coincides with a public value and therefore becomes known to the adversary. This is of course something which one would assume an implementation of the protocol to avoid, and therefore one would like to analyze the protocol under the assumption that this does not happen. Therefore, we formulate the conjecture in a negated way so that a proof of the conjecture corresponds to an attack, and the absence of a proof (equivalently, by soundness and completeness of FOL, a counter-example to the formula) corresponds to the security of the protocol. This makes sure that, when considering a given protocol execution (i.e., a given instantiation of the message variables), all models of the formulas have to fulfill the attack conjecture in order for an

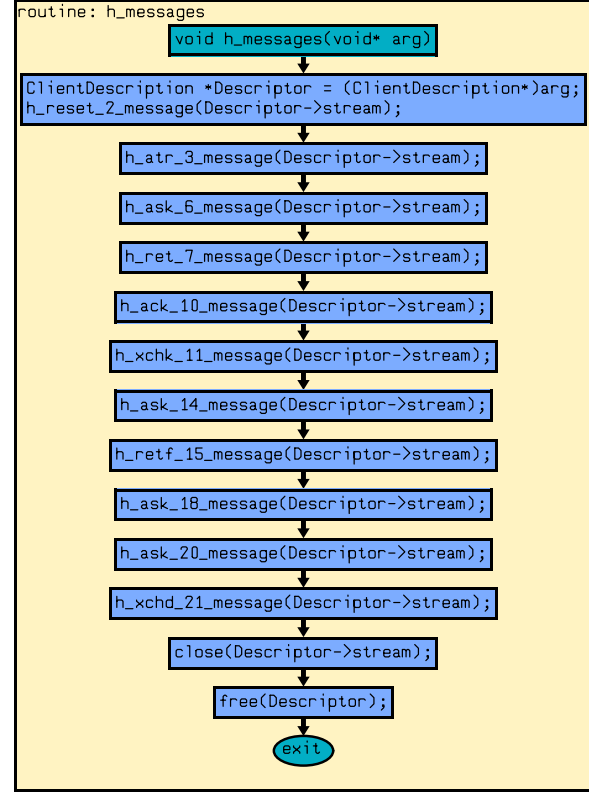


Figure 5. Main graph node

attack to be detected, in particular also any model of the formula which does not satisfy any equalities that would be assumed not to hold in an implementation of the protocol (for example, between a secret key and a public value). That way, false positives arising in this way can be avoided.

We would like to ensure that in each execution of the protocol in which the biometric match is performed, the second misuse counter is decremented. The security conjecture is formulated by inserting the predicate `match_performed` in the protocol where the biometric match is performed and predicate `fbz2written` where the decremented second misuse counter has been written to the smart-card. To formulate the security conjecture (where  $\sim$  represents logical negation), we assume that in the protocol session between the card and the host which is uniquely determined by the value of the session counters and the messages exchanged in that session (which are given as arguments to the predicates but left out here for readability), the misuse counter FBZ2 is *not* decremented, but the biometric match is still performed:

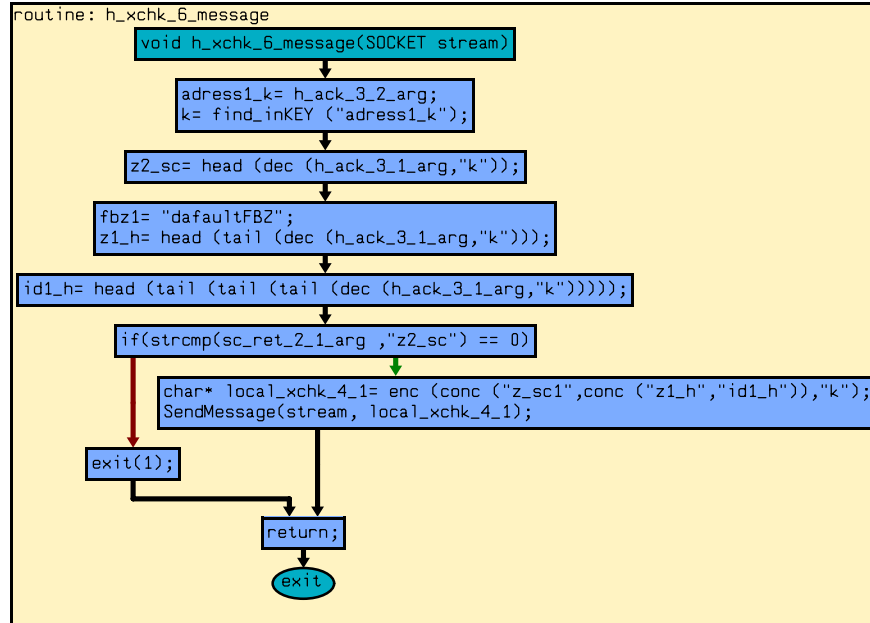


Figure 6. sc\_h\_xchk\_6\_message

`~fbz2written&& match_performed`

If this conjecture is found to be provable from the axioms, that means that in any possible run of the protocol, the adversary can make the host system perform the biometric match without decrementing the misuse-counter. This would break the security function of the misuse counter.

When applying this analysis to our prototypical implementation of the specifications provided by our industrial partner, this turned out to be in fact true. This means, in all models satisfying the set of axioms generated from the protocol description (or equivalently, in the quotient model which satisfies only those formulas which follow from the axioms and which is therefore not “degenerated”), there exists a protocol execution in which the adversary sends certain messages to the protocol participants, so that the biometric match is performed, although the second misuse counter is not decremented. Thus, the second misuse counter does not fulfill its purpose and the protocol implementation has to be seen as insecure (since an adversary can run arbitrary many tests with fake biometric samples until she succeeds in getting access with the stolen card). The result was obtained with SPASS within less than a minute computing time on an AMD Athlon processor with 1533 MHz.

clock frequency and 1024 MB RAM.

After receiving this result from the ATP, we ran the attack generator implemented in Prolog to actually exhibit the attack (by determining the valuations of the message variables, possibly of different protocol sessions). We now explain the result. It turns out that in our implementation of the specifications provided by our industrial partner, the authentication phase of the protocol is not sufficiently bound to the part where firstly the session key is exchanged and then the misuse counter decremented and the biometric match performed, as shown in Fig. 7.

The attack proceeds as follows: First, the attacker runs one execution of the protocol using the stolen smart-card and breaks up before the biometric match is performed, e.g. by pulling the smart-card out of the card-reader. Note that at this point, the second misuse counter is decremented, since the biometric authentication has not been successfully completed. The first misuse counter is unchanged, since the smart-card is legitimate and was therefore successfully authenticated at the host.

One should note that when the smart-card is pulled out (and therefore the power cut off), the smart-card returns to its initial state (except that the misuse counters are saved) and is again ready to start another fresh execution

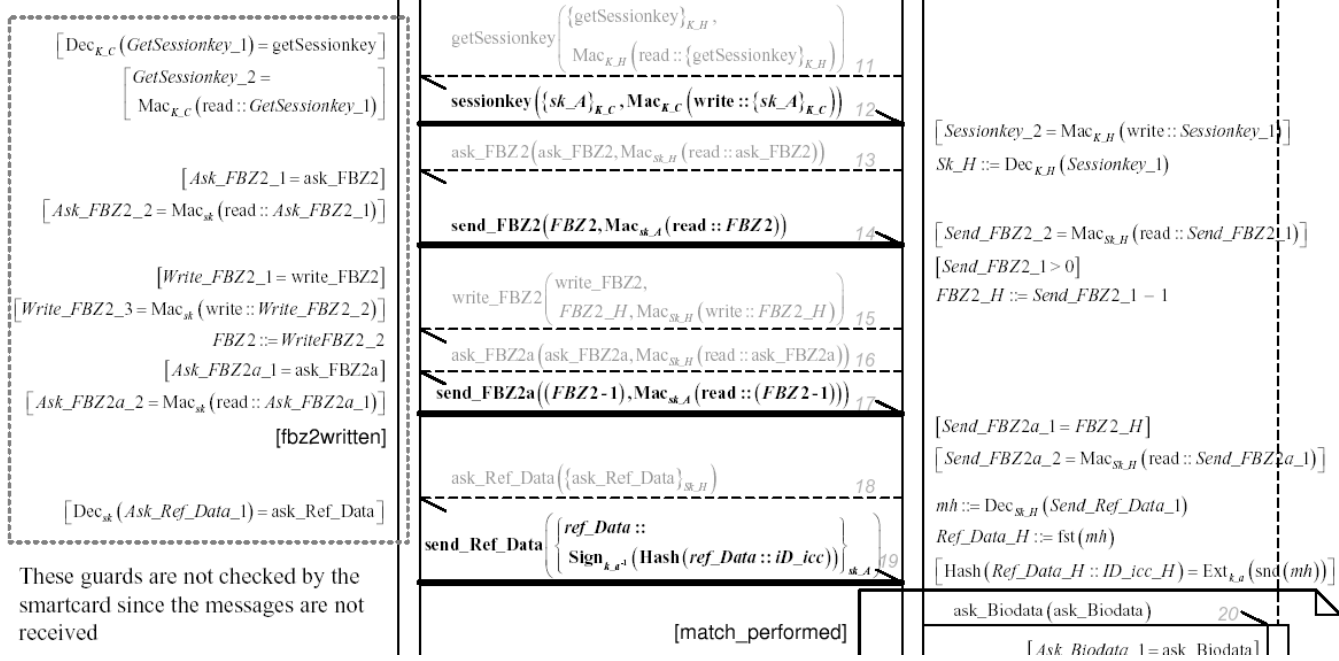


Figure 7. Attack

of the protocol. Thus, the attacker can now start another execution of the protocol with the same card. This time, when the authentication phase is finished, the attacker now manipulates the communication between the host system and the smart card reader so that the smart-card is cut off from the further communication, and she directly communicates with the host system. This way, she aims to make sure that she can now perform tests in order to get to a positive match of the biosensor using fake biometric samples. For that, she now has to perform the interaction with the host system in the messages responsible to the secure update of the second misuse counter herself in order to convince the host system that the it still communicates with the legitimate card and that the card in fact decrements its second misuse counter. However, this can now be done by replaying the messages from the previous protocol execution.

This is due to the fact that in the protocol, the host requests the session key from the smart-card. Although the session key returned by the smart-card is supposed to be protected with a MAC using the private key shared between the host system and the card, and although it is correctly checked by the host that this is actually the case, replay of the session key from the previous session is not prevented. And although the host performs the decremen-

tation of the session key itself, and later checks correctly that the smart-card actually stored the decremented key, the host has to request the current state of the second misuse counter from the smart-card to start with. At that point, the attacker can again replay the corresponding message from the smart-card in the previous protocol run, which again sends the default value of the second misuse counter to the host. As in the previous run, this value is decremented and sent to the smart-card to be stored. The smart-card is then queried whether it actually stored this decremented value. Again, this check is protected by a MAC which is actually correctly verified, but again the exact message from the smart-card in the previous protocol run can be reused. Then, the biometric reference data is requested from the smart-card, and again the response from the previous run is reused here. That way, the attacker proceeds to the point where the biometric match is performed, although the second misuse counter was not again decremented on the card.

The second part of the attack can be iterated arbitrarily many times without decrementation of any of the misuse counters. That way, the biometric match can eventually be tricked with an unacceptable probability of success.

Note that we do not assume that the attacker somehow obtains the session key from a

previous protocol run. In fact, after the successful attack against the protocol the attacker still does not know the session key. Thus, is attack is different from previous session key replay attacks and specific to the way the misuse counter mechanism is implemented which is necessary in biometric authentication protocol, due to the inherent failure rates in biometric matching.

When examining the specifications, it turned out that the security flaw found in our prototypical implementation was made possible since the specification were not sufficiently detailed with respect to the generation of the MACs used. When the specification was specialized to prescribe that the random numbers exchanged in the authentication phase are used in the generation of the MACs, and our implementation changed accordingly, the attack described above was not any more possible. We could in fact show that the protocol is secure in the sense of the security conjecture formulated above. Again, this result was obtained with SPASS within less than a minute computing time on an AMD Athlon processor with 1533 MHz. tact frequency and 1024 MB RAM.

## 5 Lessons Learned

We discuss some of the experiences gained during the application of our approach presented in this paper.

One of the lessons learned was that the amount of work one has to invest in order to apply our method is dependent on how the code was constructed. Our method needs the least amount when it is applied by the software developers in the course of programming the code. It is more effort to apply our approach to legacy systems, since performing the abstractions that are necessary requires some understanding of the software.

With respect to the preciseness of our analysis, we already mentioned that our method does not suffer from *false negatives* in the sense that it finds all attacks which exist relative to our adversary model, provided that the annotations introduced by the user are correct. The method does admit the existence of *false positives* in the sense that attack possibilities found may not have a counter-part in reality, because of the abstractions introduced for efficiency. False positives would have to be ex-

pected for example in a situation where an adversary can construct a secret out of two pieces of data which he can gain in two mutually exclusive conditional branches in the protocol, which in the most abstract application of our approach would both be taken into account. However, during several application case-studies so far, including the one presented here, this problem has not become apparent. Also, the fact that a false positive does not constitute a realistic attack becomes apparent by using the Prolog attack generator.

## 6 Related Work

[CDW04] reports on the usage of the model-checker MOPS on security-critical Unix-based applications with respect to low-level security properties such as the proper dropping of privileges, the avoidance of race conditions when accessing files, and the secure creation of temporary files. The approach is applied to applications with over one million lines of code where more than a dozen new security weaknesses in widely-deployed applications were found. Compared to our work, that approach is focussed on low-level implementation details, while we aim to analyze specifically the secure usage of cryptographic operations in implementations.

[DDMP03] presents a tool which automates the detection of high-risk security-critical functions based on the observation validated in an experiment in the paper that functions near a source of input are most likely to contain a security vulnerability. The tool is applied to three open source applications with known vulnerabilities and the privilege separation code in the OpenSSH server daemon. Compared to this approach, our approach is directed more specifically to attacks against certain security requirements such as secrecy of data, against which the code is analyzed in depth.

[DM03] uses a tool which repositions stack allocated arrays at compile time by preserving the semantics of the program with a small performance penalty. The paper considers the semantics-preserving transformation of stack allocated arrays to heap allocated "pointers to arrays". Compared to that work, ours is not concerned with the buffer-overflow type of errors, but with security flaws arising from design errors in the security protocol logic.

There are other approaches to using FOL automated theorem provers for cryptoprotocol analysis, so far applied mainly on the specification level. For example, [Coh03] uses first-order invariants to verify cryptographic protocols against safety properties. For typical protocols, the invariants can be generated automatically from the protocol specification, allowing them to be proved by ordinary first-order reasoning. The approach is supported by the ATP TAPS and has been extensively tested and shown to be efficient on the protocols that were considered. Compared to our approach, the method does not generate counter-examples (that is, attacks) in case a protocol is found to be insecure. In our approach, the attack can be generated from the proof tree. Alternatively, we have implemented our approach also in Prolog which allows one to read the attack off the message variables directly (but is not as efficient as there are only used once the protocol is found to be insecure by the ATP). In so far, the cited approach and ours are complementary.

## 7 Conclusion

We presented an approach using automated theorem provers for first order logic to understand the security requirements provided by C code implementations of cryptographic protocols. Our approach constructs a logical abstraction of the code which can be used to analyze the code for security properties (such as confidentiality) with automated theorem provers. One should note that it is not our goal to provide an automated full formal verification of C code using formal logic but to increase understanding of cryptoprotocol implementations in an approach which is as automated as possible. Note also that our focus here is on high-level security properties such as secrecy and authenticity, and not on detecting low-level security flaws such as buffer overflows. We demonstrate our approach at the hand of a biometric authentication protocol currently in development by a large German company in a joint research and development project. In all, although our approach is not completely automatic, it turned out to be applicable with reasonable effort in industrial practice.

**Acknowledgements** Many thanks go to Mark Yampolskiy for help with constructing the control flow graphs used for this work, and to Matthias Schwan for helpful discussions.

## References

- [Abs04] AbsInt. aicall. <http://www.aicall.de/>, 2004.
- [CDW04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of c code. In *NDSS*. The Internet Society, 2004.
- [CKY03] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, School of Computer Science, Carnegie Mellon University, 2003.
- [Coh03] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.
- [DDMP03] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the ‘security vulnerability likelihood’ of software functions. In *ICSM*, pages 266–. IEEE Computer Society, 2003.
- [DM03] Christopher Dahn and Spiros Mancoridis. Using program transformation to secure C programs against buffer overflows. In A.v. Deursen, E. Stroulia, and M.D. Storey, editors, *WCRE*, pages 323–333. IEEE Computer Society, 2003.
- [DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.
- [Gol03] D. Gollmann. Facets of security. In C. Priami, editor, *Global Computing (GC 2003)*, volume 2874 of *LNCS*, pages 192–202. Springer, 2003.
- [Mea91] C. Meadows. A system for the specification and analysis of key management protocols. In *IEEE Symposium on Security and Privacy*, pages 182–195, 1991.
- [RS98] P. Ryan and S. Schneider. An attack on a recursive authentication protocol. *Information Processing Letters*, 65:7–10, 1998.
- [sec] seCse tool (webinterface and download). <http://www4.in.tum.de/~secse>.
- [SS01] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving, 2001. Available at <http://www.tptp.org>.
- [SW00] G. Stenz and A. Wolf. E-SETHEO: An automated<sup>3</sup> theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *LNCS*, pages 436–440. Springer, 2000.