Verification of Low-level Crypto-Protocol Implementations Using Automated Theorem Proving*

Jan Jürjens
Software & Systems Engineering, Dep. of Informatics, TU München, Germany
http://www4.in.tum.de/~juerjens

Abstract

Designing and implementing cryptographic protocols is known to be difficult. A lot of research has been devoted to developing formal techniques to analyze abstract designs of cryptographic protocols. Less attention has been paid to the verification of implementation-relevant aspects of cryptographic protocols. This is an important challenge since it is non-trivial to securely implement secure designs, because a specification by its nature is more abstract than the corresponding implementation, and the additional information may introduce attacks not present on the design level. In this paper, we address aspects of crypto protocol implementations close to the hardware level. More concretely, we consider the industrial Cryptographic Token Interface Standard PKCS 11 which defines how software on untrustworthy hardware can make use of tamper-proof hardware such as smart-cards to perform cryptographic operations on sensitive data. We propose an approach for automated security analysis with first-order logic theorem provers of crypto protocol implementations making use of this standard. ***Keywords***: *Cryptographic protocols, hardware, code analysis, verification, automated theorem proving.*

1 Introduction

Automated theorem provers (ATPs) have been successfully applied to the problem of verifying cryptographic protocols for security requirements [Sch97, Wei99, Coh03]. An advantage is the potential for being not only automatic, but also quite efficient and powerful, because of the efficient decision procedures implemented in these tools and because security requirements can be formalized straightforwardly in first-order logic (FOL).

A disadvantage of many of these approaches is that they require developers to construct a formal specification of their protocol in a formal notation. Also, it is not clear whether an implementation of a secure design is still secure, since in general, implementations cannot be automatically generated from formal specifications. For example, [RS98] presents an attack against a protocol implementation whose design had been proven secure using formal logic. Such examples are not just due to careless implementation of specifications: A specification by its nature is more abstract than the corresponding implementation, and the additional information may introduce attacks not present on the design level.

To address this problem, we present an approach for analyzing cryptographic protocol implementations for security requirements using automated theorem provers. Specifically, in this paper, we address aspects of crypto protocol implementations close to the hardware level: We consider the industrial Cryptographic Token Interface Standard PKCS 11 which defines how software on untrustworthy hardware can make use of tamper-proof hardware such as smart-cards to perform cryptographic operations on sensitive data. The C code gives rise to a control flow graph in which the cryptographic operations offered by the cryptographic token 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 automated theorem prover (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. We demonstrate our approach at the hand of a variant of TLS (the current version of the

*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).

security protocol SSL) proposed in [APS99]. We also briefly report on experiences from an ongoing application to a biometric authentication protocol within an industrial project.

One should note that it is not our goal to provide an automated full formal verification of C code. This would not be possible in general, since security requirements are undecidable [DLMS99]. Instead, our goal is to increase trustworthiness of cryptoprotocol implementations in an approach which is 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] in that respect). 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, see for example [HS01, CKL04]). 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-overflow attacks (for which a number of tools already exist). To support our approach, a tool is available over a web-interface and as open-source which, from control flow graphs, automatically generates FOL logic formulas in the standard TPTP notation as input to a variety of ATP's [Jür04b].

Section 2 introduces the Cryptographic Token Interface Standard PKCS 11 of RSA Labs. Section 3 explains the code analysis framework for our approach. In Sect. 4, we demonstrate our approach on a variant of the TLS protocol. In Sect. 5, we shortly report on experiences from an ongoing application to a biometric authentication protocol within an industrial project. After comparing our research with related work, we close with a discussion and an outlook on ongoing research.

2 PKCS 11 Standard

The PKCS 11 standard [Lab04] specifies an ANSI C application programming interface (API) to hardware devices which can store cryptographic information and perform cryptographic functions. It supports resource sharing in the sense of multiple applications accessing multiple devices. The aim of the standard is to isolate an application from the details of the cryptographic device. One or more applications that need to perform certain cryptographic operations are linked to one or more cryptographic devices, on which some or all of the operations are performed. A user may or may not be associated with an application.

Function	Description
C_GetAttributeValue	obtains attribute value
C_SetAttributeValue	modifies attribute value
C_Encrypt	encrypts single-part data
C_Decrypt	decrypts encrypted data
C_Digest	digests single-part data
C_Sign	signs single-part data
C_VerifyRecover	verifies a signature where the data is recovered
C_GenerateKey	generates a secret key
C_GenerateKeyPair	generates a key pair
C_GenerateRandom	generates random data

Figure 1. PKCS 11 Functions

The standard provides an interface to one or more cryptographic devices through a number of "slots". Each slot (a physical reader or other device interface) may contain a cryptographic token. Applications can connect to tokens in any or all of those slots. The connections are called *sessions* and identified by a *session handle*. For the standard, a token is a device that stores objects (data, certificates, and keys) which can be accessed by *object handles*, and can perform cryptographic functions. Objects may be visible to all applications connected to a token and remain on the token beyond the end of a session (*token objects*), or they may only be visible to the application which created them and destroyed after the relevant session in which they were created is closed (*session objects*). There is a further distinction between private objects for which applications have to log on to view them and public objects otherwise, but for space reasons we have to abstract from this latter aspect in our current treatment.

An application may have one or more sessions with one or more tokens. In general, a token may have multiple sessions with one or more applications. An application may access a token concurrently from multiple threads. All threads of a given application have access to the same sessions and the same session objects. Note that if an application has multiple sessions with a token and creates a session object in one of them, that session object is visible through any of that application's sessions. As soon as the session that was used to create the object is closed, that object is destroyed. Note also that it may go unnoticed by the applications if a token is removed and re-inserted before a PKCS 11 function is executed.

The PKCS 11 functions which we will consider are presented in Fig. 1. For space reasons, here we can

```
/* C_Encrypt encrypts single-part data. */
CK_PKCS11_FUNCTION_INFO(C_Encrypt)
#ifdef CK_NEED_ARG_LIST
(
CK_SESSION_HANDLE hSession,           /*session handle*/
CK_BYTE_PTR       pData,              /*plaintext data*/
CK_ULONG          ulDataLen,          /*plaintext bytes*/
CK_BYTE_PTR       pEncryptedData,     /*gets ciphertext*/
CK_ULONG_PTR      pulEncryptedDataLen /*gets ctext size*/
);
#endif
```

Figure 2. C_Encrypt header

present only a fragment of the PKCS 11 API which is directly relevant to cryptographic operations. As an example, we give the header description for the C_Encrypt function in Fig. 2. We also have to omit our treatment of management functions such as opening and closing sessions (the latter is formalized by setting the session objects to the undefined value ε).

3 Code Analysis

We define the translation of security protocol implementations to first-order logic formulas which allows automated analysis of the source code using automated first-order logic theorem provers. The general approach and framework can be found in [JK04]; here we extend the approach to deal with the PKCS 11 standard. We assume that the following information is given:

- A description of the physical layer of the system, such as system nodes and communication links (for example Internet links), and the level of security it provides. This may be given as a UMLsec deployment diagram (see [Jür04a]).
- Secondly, the data structure of the system, including the security requirements on the system data (such as secrecy, integrity, and authenticity), which may be given in a UMLsec class diagram [Jür04a]. For the security analysis, from this information the conjecture is derived that is to be checked by the automated theorem prover.
- The source code gives the intended behavior of the system. It is extracted as a control flow graph using the aiCall tool [Abs04] which is compiled to first-order logic axioms giving an abstract interpretation of the system behavior suitable for security analysis.

The analysis approach presented here works with the well-known Dolev-Yao adversary model for security analysis [DY83] and is similar to previous approaches using first-order logic such as [Sch97, Wei99, Coh03] (for differences, see Sect. 6). More specifically, we have a broadcast setting simular to that in [Shy02] suitable in particular for the case of Internet security protocols. 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 knowledge 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. Other important data security properties such as integrity and authenticity of data can be analyzed with our method as well, although this cannot be explained here for lack of space.

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 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, $\mathsf{knows}(E)$ 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 $\mathsf{knows}(s)$.

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. 3. There, the symbols E, E', and E'' denote terms inducticely constructed in this way. These symbolic operations are the abstract versions of the cryptographic algorithms offered in PKCS 11, as defined in

• $\mathsf{enc}_S(E,E')$	(encryption)
• $\mathsf{dec}_S(E,E')$	(decryption)
• $\mathsf{hash}_S(E)$	(hashing)
• $\mathsf{sign}_S(E,E')$	(signing)
• $\mathsf{ver}_S(E,E',E'')$	(verification of signature)
• $\mathsf{ext}_S(E,E')$	(recovering content of sig.)
• $\mathsf{kgen}_S(E)$	(key generation)
• $\mathsf{inv}_S(E)$	(inverse key)
• $\mathsf{conc}_S(E,E')$	(concatenation)
• $\mathsf{head}_S(E)$ and $\mathsf{tail}_S(E)$	(head and tail of concat.)

Figure 3. Abstract Crypto Operations

Fig. 1. Note that the goal is to have all unencrypted, sensitive data stored and processed on the crypto token, since the hardware employed otherwise may not be trustworthy. Therefore, all operations in our abstract model are parameterized over a session handle S, since they may make use of data stored at the token during a particular session. Note that C_Digest from Fig. 1 is written as $\mathsf{hash}_S(E)$ here and that the functions C_GetAttributeValue, C_SetAttributeValue, C_GenerateKey, C_GenerateKeyPair, and C_GenerateRandom are not part of the crypto term algebra in Fig. 3 but are formalized implicitly in the logical formula: setting an attribute a to a value v is formalized as the logical constraint $\mathsf{a} = \mathsf{v}$ on the models (which any valid model of the axioms will have to fulfill, whereby it amounts to an assignment); getting the value from the attribute a is modeled by just using that attribute; and C_GenerateKey, C_GenerateKeyPair, and C_GenerateRandom are formalized by introducing new constants representing the keys and random values (and making use of the $\mathsf{inv}_S(E)$ operation in the case of C_GenerateKeyPair). Note also that we introduce an additional function $\mathsf{ext}_S(E,E')$ here which lets one conveniently recover the plaintext content of a signature.

In that term algebra, one then defines the equations $\mathsf{dec_S(enc_{S'}(E, inv_{S''}(K)), K) = E}$, $\mathsf{ver_S(sign_{S'}(E, inv_{S''}(K)), K, E) = true}$, and $\mathsf{ext_S(sign_{S'}(E, K), inv_{S''}(K)) = E}$ for all terms $\mathsf{E, K}$ and session handles $\mathsf{S, S', S''}$, and the usual laws regarding concatenation, $\mathsf{head}_S()$, and $\mathsf{tail}_S()$. 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 $\mathsf{knows(E)}$ 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. 4. 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). We observe that the graph can be transformed to consist of transitions of the form $\mathsf{trans(state, inpattern, condition, action, truestate)}$, where $\mathsf{inpattern}$ is empty and $\mathsf{condition}$ equals true where they are not needed, and where action is a logical expression of the form $\mathsf{localvar = value}$ respectively $\mathsf{outpattern}$ 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.

Now assume that the source code gives rise to a transition $\mathsf{TR1 = trans(s1, i1, c1, a1, t1)}$ such that there is a second transition $\mathsf{TR2 = trans(s2, i2, c2, a2, t2)}$ where $\mathsf{s2 = t1}$. If there is no such transition TR2, we define $\mathsf{TR2 = trans(t1, [], true, [], t1)}$ to simplify our presentation, where [] is the empty input or output pattern and $\mathsf{true}$is the boolean condition. Suppose that $\mathsf{c1}$ is of the form $\mathsf{cond(arg_1, \ldots, arg_n)}$. For $\mathsf{i1}$, we define $\bar{\mathsf{i1}} = \mathsf{knows(i1)}$ in case $\mathsf{i1}$ is non-empty and otherwise $\bar{\mathsf{i1}} = \mathsf{true}$. For $\mathsf{a1}$, we define $\bar{\mathsf{a1}} = \mathsf{a1}$ in case $\mathsf{a1}$ is of the form $\mathsf{localvar = value}$ and $\bar{\mathsf{a1}} = \mathsf{knows(outpattern)}$ in case $\mathsf{a1 = outpattern}$ (and $\bar{\mathsf{a1}} = \mathsf{true}$ in case $\mathsf{a1}$ is empty). Then for TR1 we define the following predicate:

$$\mathsf{PRED(TR1)} \equiv \bar{\mathsf{i1}} \& \mathsf{c1} \Rightarrow \bar{\mathsf{a1}} \& \mathsf{PRED(TR2)} \qquad (1)$$

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

```
%---- Basic Crypto Axioms  ----
input_formula(construct_message_1,axiom,(
! [E1,E2,S,S',S''] :
  equal(head(S,conc(S',E1,E2)),E1)
  & equal(tail(S,conc(S',E1,E2)),E2)
  & equal(dec(S,enc(S',E1,E2),inv(S'',E2)),E1)
  & equal(ext(S,sign(S',E1,inv(S'',E2)),E2),E1)
  & equal(ver(S,sign(S',E1,inv(S'',E2)),E2,E1),true)
)).
```

```
%---- Basic Relations on Knowledge  ----
input_formula(construct_message,axiom,(
! [E1,E2,S] : ((knows(E1) & knows(E2))
=> (knows(conc(S,E1,E2))
   & knows(head(S,E1))
   & knows(tail(S,E1))
   & knows(enc(S,E1,E2))
   & knows(dec(S,E1,E2))
   & knows(sign(S,E1,E2))
   & knows(ext(S,E1,E2))
   & knows(hash(S,E1)))))).
```

Figure 4. Some general crypto axioms

on interaction rather than computation), this can only be 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. This is a classical approach in automated software verification, see for example [HS01, CKL04]. The analysis process can also be iterated with n as the iteration variable to approximate the unbounded loops as far as possible (within the limits of tool performance). Although this works fine with our practical examples, we are currently working towards providing limits after which a further security analysis becomes redundant, similar to ideas in [Sto02].

We explain how we deal with concurrent threads sharing objects on the crypto token. First, we identify maximal transition paths in the control flow graph between synchronization points (that is, where shared variables are written or read). We have to consider all possible interleavings between these maximal transition paths. This is done by constructing a formula ϕ constructing of nested implications of the form like formula 1 but containing predicates $\mathsf{PRED}(\mathsf{Pi})$ where i ranges from 1 to the number of paths n. We then consider the all-quantification of the formula $\psi \Rightarrow \phi$ over the possible interleavings of the paths (represented as ordered lists of the numbers 1 through n), where ψ is an equational formula assigning to the predicate $\mathsf{PRED}(\mathsf{Pi})$ the values from the predicate formalizing the path numbered j, where j is the ith element in the ordered list. This way we can detect security flaws arising from concurrent access to shared objects on the token (for example one threads storing a confidential value to an object and then another sending out the content of that object unencrypted).

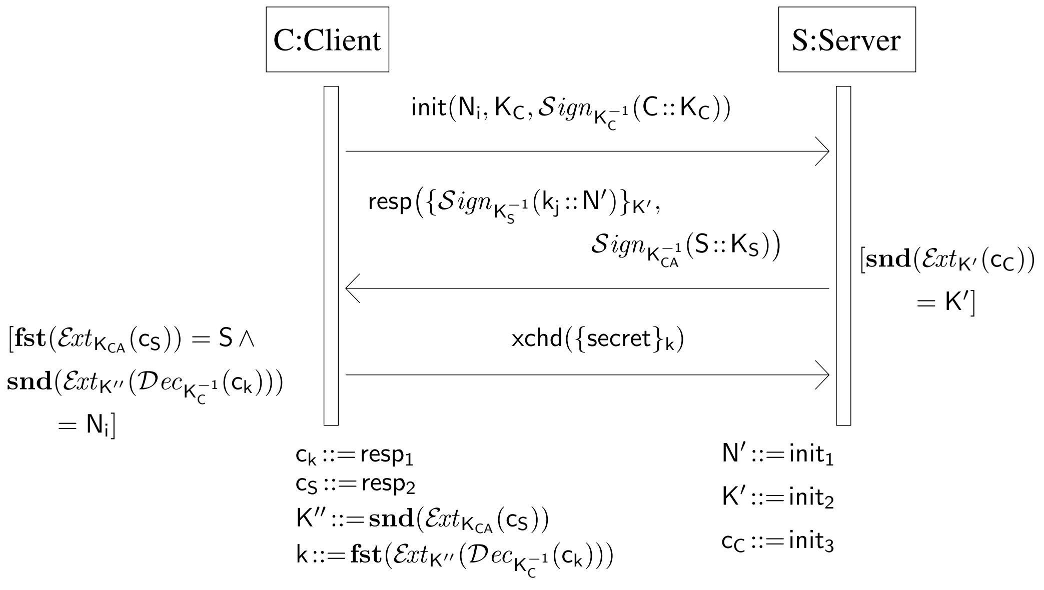
The predicates $\mathsf{PRED}(\mathsf{TR})$ for all such transitions TR are then joined together using logical conjunctions. The resulting logical formula is closed by forall-quantification over all free variables contained.

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 as a conjecture (for example, $\mathsf{knows}(\mathsf{secret})$ in case the secrecy of the value secret is to be checked). 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 $\mathsf{knows}(\mathsf{secret})$ 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 $\mathsf{knows}(\mathsf{secret})$ 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 that due to the undecidability of Horn formulas with equations, 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.

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. 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 vari-



C:Client
S:Server
$\mathsf{init}(\mathsf{N_i}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$
$\mathsf{resp}\big(\{\mathcal{S}ign_{\mathsf{K_S^{-1}}}(\mathsf{k_j} :: \mathsf{N'})\}_{\mathsf{K'}}, \mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}(\mathsf{S} :: \mathsf{K_S})\big)$
$[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K'}}(\mathsf{c_C})) = \mathsf{K'}]$
$\mathsf{xchd}(\{\mathsf{secret}\}_{\mathsf{k}})$
$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S})) = \mathsf{S} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k}))) = \mathsf{N_i}]$
$\mathsf{c_k} ::= \mathsf{resp_1}$
$\mathsf{c_S} ::= \mathsf{resp_2}$
$\mathsf{K''} ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))$
$\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))$
$\mathsf{N'} ::= \mathsf{init_1}$
$\mathsf{K'} ::= \mathsf{init_2}$
$\mathsf{c_C} ::= \mathsf{init_3}$

Figure 5. Variant of the TLS handshake protocol

ables during its search, which can then be queried).

4 A Variant of TLS

We will analyze a variant of the handshake protocol of TLS[1] proposed in [APS99]. To show applicability of our approach, we demonstrate that we can detect the flaw observed in [Jür04a]. The goal of the protocol is to let a client send a secret over an untrusted communication link to a server in a way that provides secrecy and server authentication, by using symmetric session keys.

As shown in the specification in Fig. 5, the protocol proceeds as follows. The symbols msg_n for each message name msg and number n, represent the nth argument of the message msg_n. received by an object according to the protocol specification. The client C initiates the protocol by sending the message $\mathsf{init}(\mathsf{N_i}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$ to the server S. If the condition $[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K'}}(\mathsf{c_C}))=\mathsf{K'}]$ holds, where $\mathsf{K'} ::= \mathsf{init_2}$ and $\mathsf{c_C} ::= \mathsf{init_3}$ (that is, the key K_C contained in the signature matches the one transmitted in the clear), S sends the message $\mathsf{resp}\big(\{\mathcal{S}ign_{\mathsf{K_S^{-1}}}(\mathsf{k_j} :: \mathsf{N'})\}_{\mathsf{K'}}, \mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}(\mathsf{S} :: \mathsf{K_S})\big)$ back to C (where $\mathsf{N'} ::= \mathsf{init_1}$). Then if the condition

$$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))=\mathsf{S} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))=\mathsf{N_i}]$$

holds, where $\mathsf{c_k} ::= \mathsf{resp_1}$ and $\mathsf{c_S} ::= \mathsf{resp_2}$, and $\mathsf{K''} ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))$ (that is, the certificate is actually for S and the correct nonce is returned), C sends $\mathsf{xchd}(\{\mathsf{s_i}\}_{\mathsf{k}})$ to S, where $\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

[1]TLS is the current successor of the Internet security protocol SSL.

Figure 6 gives a simplified C implementation of the client side of the protocol, where the cryptographic operations from Fig. 1 are already substituted by the abstract operations in Fig. 3 and the cryptographic data is represented by strings. From this abstracted code, we can then generate the control flow graph. Although the complete graph cannot be shown here we show as examples the fragments representing the main function in Fig. 7 and the s_xchd_3_message function in Fig. 8.

The main part of the transformation to the e-SETHEO input formal TPTP is given in Fig. 9. The protocol itself is expressed by a for-all quantification over the session handles used and the pieces of messages which are transferred over the communication channel. The variable S represents the session handle which is included during the translation from the control flow graph to the FOL formula. The message variables Init_1, Init_2, Init_3, and Xchd_1 stand for the messages received by the server. The message variables Resp_1 and Resp_2 stand for the client receiving messages parts. The protocol example includes three messages (cf. Fig. 5): the first one sent from the client, the second one from the server and the third one sent again from the client. Each message is expressed by an implication.

When given the formulas generated from the source

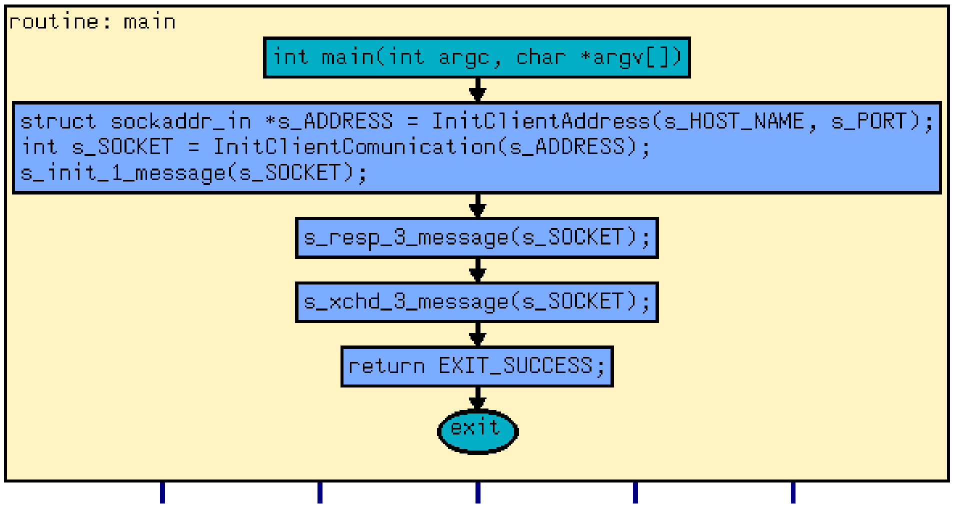

routine: main
int main(int argc, char *argv[])
struct sockaddr_in *s_ADDRESS = InitClientAddress(s_HOST_NAME, s_PORT);
int s_SOCKET = InitClientComunication(s_ADDRESS);
s_init_1_message(s_SOCKET);
s_resp_3_message(s_SOCKET);
s_xchd_3_message(s_SOCKET);
return EXIT_SUCCESS;
exit

Figure 7. Control graph for main function

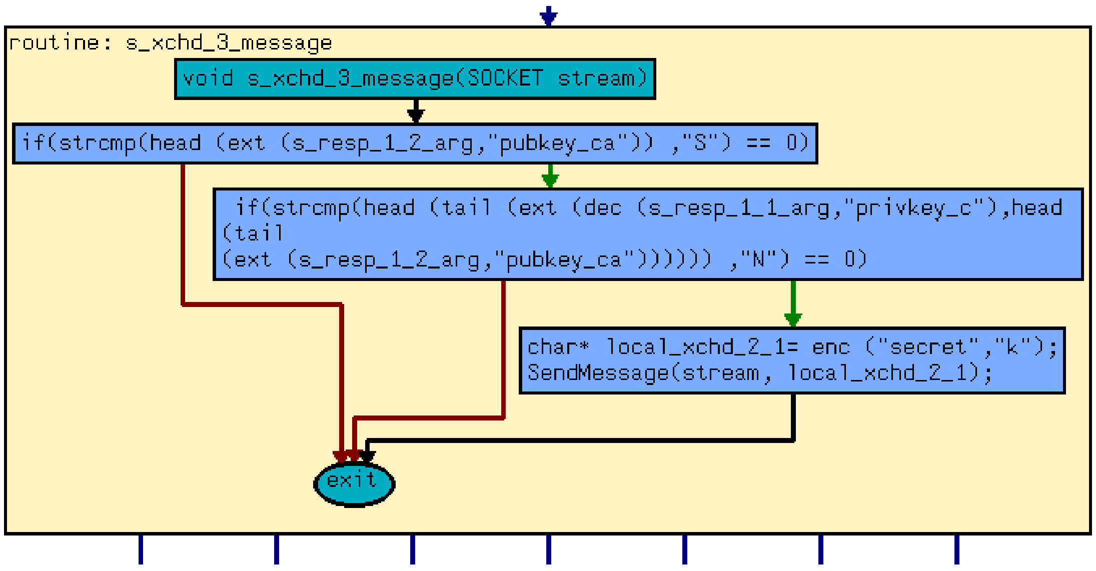
code together with the conjecture known(secret), the prover returns Proof found (within a few seconds on an AMD Athlon processor with 1533 MHz. tact frequency and 1024 MB RAM), which means that the secrecy requirement on the value secret is not fulfilled. Using the Prolog attack generation script we could then generate the attack scenario to see that it is not a false positive. Details about the attack can be found in [Jür04a] as mentioned above.

5 Industrial Application

We are applying our method in an industrial project with a major German company. The goal is the correct development of a security-critical biometric authentication system which is supposed to control access to a protected resource. 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. Our approach has already been applied at the specification level [Jür05]. Since the implementation is created manually from the specification, and therefore is open to implementation bugs and introduced additional security flaws, we are currently continuing to apply our approach on the implementation level. So far, we have already found several severe security flaws which allow an adversary to disable the misuse counters which are supposed to detect attempts to authenticate using forged biometric samples.

6 Related Work

There are other approaches to using FOL automated theorem provers for cryptoprotocol analysis, so far applied mainly on the specification level. [Sch97] formalizes the well-known BAN logic in first-order logic and uses the ATP SETHEO to prove statements in the BAN logic. BAN logic is a modal belief logic used to formulate the beliefs of protocol participants during protocol execution. It has been successfully applied to the analysis of authenticity properties of protocols. It is less suitable for reasoning about secrecy and not particularly close to an execution model of a protocol. In that sense, it is different from our approach which is based on the knowledge of the adversary, instead of the beliefs of the protocol participants. [Wei99] analyzes the Neuman-Stubblebine key exchange protocol using FOL and the ATP Spass. The protocol is translated into first-order monadic Horn fragments and analyzed for attacks against the secure key establishment between the two protocol participants. This approach differs from ours for example in that in general we also use non-monadic Horn formulas (and even non-Horn formulas), to be able to consider unbounded state when



routine: s_xchd_3_message
void s_xchd_3_message(SOCKET stream)
if(strcmp(head (ext (s_resp_1_2_arg,"pubkey_ca")) ,"S") == 0)
if(strcmp(head (tail (ext (dec (s_resp_1_1_arg,"privkey_c"),head (tail (ext (s_resp_1_2_arg,"pubkey_ca")))))) ,"N") == 0)
char* local_xchd_2_1= enc ("secret","k"); SendMessage(stream, local_xchd_2_1);
exit

Figure 8. Control graph for s_xchd_3_message function

necessary to express a security property.

[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 are therefore only used once the protocol is found to be insecure by the ATP). In so far, the cited approach and ours are complementary.

In other approaches to automated software engineering for security, [KAH99] uses the Software Cost Reduction method (SCR) to analyze a cryptographic system for various security properties. Lowlevel security properties have so far received comparatively little attention; examples are [ALP03], analyzing security policies for Security-Enhanced Linux, and [GRS99], securely refining architectural descriptions down to implementations.

There has been a substantial amount of research regarding program analysis. In [HS01] model checking and an automated model extraction in combination with a lookup table is used to verify large software applications. [CKL04] presents a tool for the automated formal verification of ANSI-C programs using Bounded Model Checking. The tool supports most ANSI-C language features, such as pointer constructs, dynamic memory allocation, recursion, and the float and double data types. Both approaches differ from ours in that they do not consider security properties.

We are not aware of any existing work formally verifying crypto protocol implementations making use of the PKCS 11 crypto API.

7 Conclusion

We presented an approach using automated theorem provers for first order logic to analyze C code implementations of cryptographic protocols making use of the PKCS 11 API for security requirements. The goal is to detect security flaws in code, as opposed to abstract specifications, because of the difficulty in securely implementing a secure specification. Our approach constructs a logical abstraction of the code which can be used to verify predefined 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 trustworthiness of crypto-protocol 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

```
input_formula(protocol,axiom,(
 ![S,Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1] : (
 % C -> Attacker
( ( ( true & true )
      => knows(conc(S, n,  conc(S, k_c,  sign(S, conc(S, c, conc(S, k_c, eol)), inv(S, k_c)) ) ))
     & ( (knows(Resp_1) & knows(Resp_2)
         & equal( fst(S, ext(S, Resp_2, k_ca)), s) & equal(snd(S, ext(S, dec(S, Resp_1, inv(S, k_c)),
              snd(S, ext(S, Resp_2, k_ca)))), n ) )
        => knows(enc(S, secret, fst(S, ext(S, dec(S, Resp_1, inv(S, k_c)), snd(S, ext(S, Resp_2, k_ca))))))
       ) ) )
 &  % S -> Attacker
( ( ( knows(Init_1) & knows(Init_2) & knows(Init_3)
       & equal( snd(S, ext(S, Init_3, Init_2)), Init_2 ) )
      => knows(conc(S, enc(S, sign(S, conc(S, kgen(S, Init_2), conc(S, Init_1, eol)), inv(S, k_s)), Init_2),
            sign(S, conc(S, s, conc(S, k_s, eol)), inv(S, k_ca) ) ))
     & ( ( knows(Xchd_1)
         & true )
        => true ) ) ) ) ) ).
```

Figure 9. Core Protocol Axiom

such as buffer overflows. We demonstrate feasibility of our approach at the hand of a variant of the TLS protocol. Also, we are currently applying it to a biometric authentication protocol currently in development by a large German company. Experiences from that application have been quite encouraging, in so far as several significant security flaws have been identified automatically and corrected in subsequent versions of the protocol. The verification method is automatic and, according to the experiences gained from the presented application, sufficiently powerful to address industrial-size systems.

Acknowledgements Many thanks go to Mark Yampolskiy for help with constructing the C program examples and control flow graphs used for this work, and to Andreas Gilg for collaboration on implementing the security analysis tool, as well as Gernot Stenz for helpful comments on using e-SETHEO.

References

[Abs04] AbsInt. aicall. http://www.aicall.de/, 2004.

[ALP03] Myla Archer, Elizabeth Leonard, and Matteo Pradella. Analyzing security-enhanced linux policy specifications. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 158–172. IEEE Computer Society, 2003.

[APS99] V. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Conference on Computer Communications (IEEE Infocom)*, pages 717–725. IEEE Computer Society, March 1999.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 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] Ernie Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[DLMS99] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP 1999)*, 1999.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[GRS99] Fred Gilham, Robert A. Riemenschneider, and Victoria Stavridou. Secure interoperation of secure distributed databases. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM 99)*, number 1708 in Lecture Notes in Computer Science, pages 701–717. Springer-Verlag, 1999.

[HS01] G.J. Holzmann and M.H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.

[JK04] J. Jürjens and Thomas Kuhn. Practical security analysis of C programs using automatic theorem provers. Technical Report ITB 51, Verisoft Project, December 2004.

```
#include "common.h"
char*random = "random";
#define s_PORT 127//specify port
#define s_HOST_NAME "127.0.0.1"//specify port

void s_init_1_message();
void s_resp_2();
char * s_resp_1_1_arg=NULL;
char * s_resp_1_2_arg=NULL;
void s_xchd_3_message();

void s_init_1_message(SOCKET stream)
{
 char* local_init_1_1= "N";
 char* local_init_1_2= "pubkey_c";
 char* local_init_1_3= sign ( (conc ("c",
    "pubkey_c"), "privkey_c");
 SendMessage(stream, local_init_1_1);
 SendMessage(stream, local_init_1_2);
 SendMessage(stream, local_init_1_3);
}
void s_resp_2(SOCKET stream)
{
s_resp_1_1_arg= ReceiveMessage(stream);
s_resp_1_2_arg= ReceiveMessage(stream);
}
void s_xchd_3_message(SOCKET stream)
{
 if(strcmp(head (ext (s_resp_1_2_arg,"pubkey_ca")),
    "S") == 0)
{
 if(strcmp(head (tail (ext (dec (s_resp_1_1_arg,
    "privkey_c"),head
(tail
(ext (s_resp_1_2_arg,"pubkey_ca")))))) ,"N") == 0)
{
 char* local_xchd_2_1= enc ("secret","k");
 SendMessage(stream, local_xchd_2_1);}}
}int main(int argc, char *argv[])
{
 struct sockaddr_in *s_ADDRESS = InitClientAddress
    (s_HOST_NAME, s_PORT);
 int s_SOCKET = InitClientComunication(s_ADDRESS);
 s_init_1_message(s_SOCKET);
 s_resp_2(s_SOCKET);
 s_xchd_3_message(s_SOCKET); return EXIT_SUCCESS;
}
```

Figure 6. Fragment of abstracted client code

[Jür04a] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

[Jür04b] J. Jürjens. Security analysis tool (webinterface and download), 2004. http://www4.in.tum.de/csduml/interface.

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[KAH99] J. Kirby, M. Archer, and C. Heitmeyer. Applying formal methods to an information security device: An experience report. In *4th IEEE International Symposium on High Assurance Systems Engineering (HASE 1999)*, pages 81–88. IEEE Computer Society, 1999.

[Lab04] RSA Labs. PKCS 11. http://www.rsasecurity.com/rsalabs/node.asp?id=2133, 2004.

[RS98] P. Ryan and S. Schneider. An attack on a recursive authentication protocol. *Information Processing Letters*, 65:7–10, 1998.

[Sch97] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In W. McCune, editor, *14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Computer Science*, pages 87–100. Springer-Verlag, 1997.

[Shy02] R. K. Shyamasundar. Analyzing cryptographic protocols in a reactive framework. In Agostino Cortesi, editor, *VMCAI*, volume 2294 of *Lecture Notes in Computer Science*, pages 46–64. Springer, 2002.

[SS01] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving, 2001. Available at http://www.tptp.org.

[Sto02] Scott D. Stoller. A bound on attacks on authentication protocols. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *IFIP TCS*, volume 223 of *IFIP Conference Proceedings*, pages 588–600. Kluwer, 2002.

[SW00] G. Stenz and A. Wolf. E-SETHEO: An automated^3 theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 2000.

[Wei99] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328, 1999.