Understanding Security Goals Provided by Crypto-Protocol Implementations*

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 presence of cryptographic interactions. A lot of research has been devoted to develop formal techniques to analyze abstract specifications of cryptographic protocols. Less attention has been paid to the source code analysis of legacy crypto-protocol implementations, for which specifications are often not available. This is an important challenge since it is non-trivial to determine from a given protocol implementation exactly which security goals are achieved, which is necessary for a reliable maintenance of security-critical systems. In this paper, we propose an approach to determine security goals provided by an implemented protocol based on control flow graphs and automated theorem provers for first-order logic.

Automated theorem provers (ATPs) have been successfully applied to the problem of verifying crypto protocols for security requirements (for example in [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. For legacy systems, this is often not practical or even feasible at all.

To address this problem, we present an approach for analyzing crypto protocol implementations for security requirements using ATPs. Specifically, in this paper, we address the issue of legacy crypto protocol implementations and how to determine the security properties provided by them. The C code gives rise to a control flow graph (CFG) in which the crypto operations are represented as abstract functions. The CFG is translated to formulas in FOL with equality. Together with a logical formalization of the security requirements, they are then given as input into any ATP supporting the TPTP input notation (such as e-SETHEO [SW00]). 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 applied our approach to a variant of the security protocol TLS (the current version of SSL, used in many browsers to set up an https connection) proposed at the conference IEEE Infocom 1999.

It is not our goal to provide an automated full formal verification of C code, but to increase understanding of the security properties enforced by cryptoprotocol implementations in an approach which is as automated as possible. 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 currently investigated through a bounded number of rounds (which is a classical approach in automated software verification, see for example [HS01]). 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). For our approach, a tool is available over a web-interface and as open-source which, from CFGs, automatically generates FOL logic formulas in the standard TPTP notation as input to a variety of ATP's [Jür04].

*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).

© 2005 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/ICSM.2005.97

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

Figure 1. Abstract Crypto Operations

Code Analysis The analysis approach presented here works with the well-known Dolev-Yao adversary model for security analysis [DY83]. The idea is that an adversary can read messages sent other 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.

We explain the transformation from the CFG generated from the C program to FOL, which is given as input to the ATP. 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. 1. Note that 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 atttribute a is modeled by just using that attribute; and generation of keys and random values is formalized by introducing new constants representing the keys and random values.

In that term algebra, one defines the equations $\mathsf{dec}(\mathsf{enc}(\mathsf{E},\mathsf{K}),\mathsf{inv}(\mathsf{K})) = \mathsf{E}$ and $\mathsf{ver}(\mathsf{sign}(\mathsf{E},\mathsf{inv}(\mathsf{K})),\mathsf{K},\mathsf{E}) = \mathsf{true}$ for all terms E,K, and the usual laws regarding concatenation, $\mathsf{head}()$, and $\mathsf{tail}()$. This abstract information is automatically generated from the concrete source code.

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

Figure 2. Some general crypto axioms

The set of predicates defined to hold for a given program is defined as follows. For each publicly known expression E, the statement $\mathsf{knows}(\mathsf{E})$ 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 crypto operations, formulas are generated for these operations for which some examples are given in Fig. 2. We use the TPTP notation for the FOL formulas, which is the input notation for many ATPs including the one we use (e-SETHEO [SW00]). Here & means logical conjunction and ![E1, E2] forall-quantification over E1, E2.

The CFG is transformed to consist of transitions of the form $\mathsf{trans}(\mathsf{state}, \mathsf{inpattern}, \mathsf{condition}, \mathsf{action}, \mathsf{truestate})$, 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 $\mathsf{localvar} = \mathsf{value}$ resp. $\mathsf{outpattern}$ 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 $\mathsf{TR1} = \mathsf{trans}(\mathsf{s1}, \mathsf{i1}, \mathsf{c1}, \mathsf{a1}, \mathsf{t1})$ such that there is a second transition $\mathsf{TR2} = \mathsf{trans}(\mathsf{s2}, \mathsf{i2}, \mathsf{c2}, \mathsf{a2}, \mathsf{t2})$ where $\mathsf{s2} = \mathsf{t1}$. If there is no such transition $\mathsf{TR2}$, we define $\mathsf{TR2} = \mathsf{trans}(\mathsf{t1}, [], \mathsf{true}, [], \mathsf{t1})$ to simplify our presentation, where [] is the empty input or output pattern. Suppose that $\mathsf{c1}$ is of the form $\mathsf{cond}(\mathsf{arg_1}, \ldots, \mathsf{arg_n})$. For $\mathsf{i1}$, we define $\bar{\mathsf{i1}} = \mathsf{knows}(\mathsf{i1})$ 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 $\mathsf{localvar} = \mathsf{value}$ and $\bar{\mathsf{a1}} = \mathsf{knows}(\mathsf{outpattern})$ in case $\mathsf{a1} = \mathsf{outpattern}$ (and $\bar{\mathsf{a1}} = \mathsf{true}$ in case $\mathsf{a1}$ is empty). Then for $\mathsf{TR1}$ we define the following predicate:

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

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 localvar = value holds in case of an assignment, or the adversary gets to know outpattern, in case it is send out in a1. Also then the predicate for the succeeding transition TR2 will hold.

To construct the recursive definition above, we assume that the CFG is finite and cycle-free. Since in general there may be unbounded loops in the C program (although in the case of crypto protocols we consider here, these are not so prevalent because the emphasis is 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 cycle up to the transition path length n. This is a classical approach in automated software verification, see for example [HS01]. 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).

We explain how we deal with concurrent threads. Firstly, we identify maximal transition paths in the CFG 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 PRED(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 PRED(Pi) the values from the predicate formalising 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 resources (for example one threads storing a confidential value to an object and then another sending out the content of that object unencrypted).

The predicates PRED(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 ATP will take these formulas as given. The security requirement to be checked is written into the TPTP file as a conjecture (for example, knows(secret) in case the secrecy of the value secret is to be checked). The ATP 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

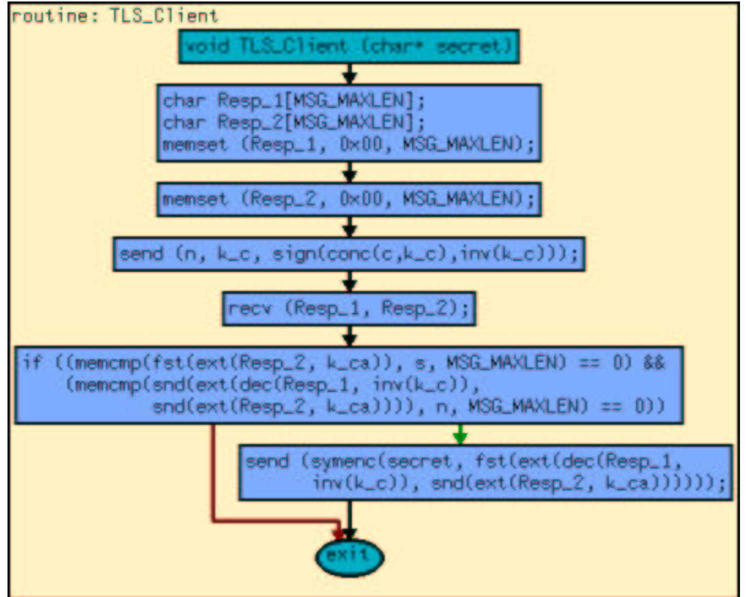

Figure 3. Control flow graph for client

get to know secret. If the ATP returns that it is not possible to derive knows(secret) from the axioms, this means that the adversary will not get secret. More details on how to perform this analysis given the FOL formulas are explained in [Jür05].

We applied our method to a variant of the security protocol TLS (the current version of SSL, used in many browsers to set up an https connection). 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. In Fig. 3, we give the CFG automatically generated for a fragment of the client side of the handshake protocol from this TLS variant. The client C initiates the protocol by sending a message to the server S, containing a random number n, the client public key k_c and a self-signed certificate which binds the public key of the client to its identity. Then the client waits for a message with two arguments sent by the server, which again contain certain certificates which also include the encrypted session key. The client checks the certificates and if this succeeds, sends the secret encrypted under the session key to the server. If any of the checks fail, the client stops the execution of the protocol. The fragment of the transformation to the e-SETHEO input format corresponding to the program fragment in Fig. 3 is given in Fig. 4.

When given the formulas generated from the source code, one can now formulate the security requirements against which the code should be analyzed. For ex-

```
input_formula(protocol,axiom,(
 ![Resp_1, Resp_2] : (((knows(conc(n, conc(k_c,sign(conc(c,conc(k_c,eol)),inv(k_c)))))
         & ((knows(Resp_1) & knows(Resp_2)
         & equal(fst(ext(Resp_2,k_ca)),s) & equal(snd(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca)))),n))
     => knows(enc(secret,fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca)))))))))).
```

Figure 4. Core protocol axiom for client

ample to see whether the data value secret is indeed kept secret, one queries the ATP whether the conjecture known(secret) can be derived from the axioms generated from the source code. That way, one can understand which security properties are provided by the code. In the case of the handshake from the TLS variant, we actually found a security flaw which breaks the secrecy of the secret to be communicated. One should note that this does not concern the actual TLS protocol, but a variant proposed at the conference IEEE Infocom 1999.

Related Work Comparatively little work so far has been devoted to the application of program understanding approaches to the specific case of security-critical software. Most notably, [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. There are other approaches to using FOL ATPs for cryptoprotocol analysis, so far applied mainly on the specification level. For example, [Coh03] uses first-order invariants to verify crypto protocols against safety properties. For typical protocols, the invariants can be generated automatically from the protocol specification, allowing to be proved by ordinary first-order reasoning. Some early ideas towards our approach were presented in [JK04].

Conclusion We presented an approach using automated theorem provers (ATPs) for first order logic to understand the security requirements provided by legacy C code implementations of cryptographic protocols. The goal is to analyze code, even where no specifications or documentation is present. Our approach constructs a logical abstraction of the code which can be used to analyze the code for security properties (such as confidentiality) with ATPs.

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 variant of the security protocol TLS (the current version of SSL, used in many browsers to set up an https connection) proposed at the conference IEEE Infocom 1999. In all, although our approach is not completely automatic, it turned out to find some actual security flaws at realistic cryptoprotocol implementations.

Acknowledgements Help from Mark Yampolskiy on the example is very gratefully acknowledged.

References

[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.

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

[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 T. Kuhn. Practical security analysis of C programs using automatic theorem provers. Technical Report ITB 51, Verisoft Project, Dec. 2004.

[Jür04] 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.

[SW00] G. Stenz and A. Wolf. E-SETHEO: An automated[3] theorem prover. In R. Dyckhoff, editor, *TABLEAUX 2000*, volume 1847 of *LNCS*, pages 436–440. Springer, 2000.