Code Security Analysis with Assertions[*]

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

Mark Yampolskiy
Software & Systems Engineering
Dep. of Informatics
TU Munich
http://www4.in.tum.de/~secse

ABSTRACT

Designing and implementing cryptographic protocols is known to be difficult. A lot of research has been devoted to develop 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.

We propose an approach to determine security goals provided by a protocol implementation based on control flow graphs and automated theorem provers for first-order logic. More specifically, here we explain how to make use of assertions in the source code for a practical and efficient security analysis.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Security,Verification

Keywords

Code security analysis, assertions, first-order logic, automated theorem proving, C programs, cryptographic protocols, cryto-based software.

[*]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).

1. INTRODUCTION

Understanding the security goals provided by cryptographic protocol implementations is one of the major challenges with security-critical systems. Since security requirements such as secrecy, integrity and authenticity of data are always relative to an unpredictable adversary, they are difficult to even define precisely, let alone to determine in a complex software system making sophisticated use of cryptographic operations for example to authenticate communication partners over an untrusted network. 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 source code 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 (of the kind our approach can detect) can be found in [10]. Even in software projects where specification techniques such as the UML 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 an implemented protocol on the source-code level. Our approach is based on control flow graphs and automated theorem provers for first-order logic (FOL). Automated theorem provers (ATPs) have been successfully applied to the problem of verifying cryptographic protocols for security requirements (for example in [11]). An advantage is the potential for being not only automatic, but also efficient and powerful due to their efficient proof procedures and because security requirements can be formalized straightforwardly in 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. 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 the issue of how to make use of assertions in the source code for a practical and efficient security analysis. The C code gives rise to a control flow graph (CFG) in which the crypto operations are represented as abstract functions. 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 automated theorem prover supporting the TPTP input notation (such as e-SETHEO [12]). If the analysis reveals that there could be an attack against the protocol, an attack generation script written in Prolog is generated.

The assertions can also be seen as some kind of formal and semantically meaningful documentation which can be adapted for a changing system later. Also, this way a provider of a software component can increase trustworthiness of this product for the client by including these assertions which can be automatically verified by the client (similar to the case of proof-carrying code).

Our goal is not to provide an automated full formal verification of C code but to increase understanding of the security properties enforced by cryptoprotocol implementations in a way as automated as possible. Because of the abstractions, the approach may produce false alarms (which however have not surfaced yet in practical examples). For space restrictions we cannot consider features such as pointer arithmetic in our presentation here (we essentially follow the approach in [1] in that respect). We do not consider casts, and expressions are assumed to be well-typed. Loops are currently only investigated through a bounded number of rounds[1] (which is a classical approach in automated software verification, see for example [6]). Note also that our focus here is on high-level security properties such as secrecy, and not on detecting low-level security flaws such as buffer-overflow attacks (for which a number of tools already exist). We provide a tool over a webinterface 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 ATPs [7].

2. 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 [9].

The analysis approach presented here works with the well-known Dolev-Yao adversary model for security analysis [4] and is similar to previous approaches using first-order logic such as [11]. 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 $\mathsf{knows}(s)$. 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.

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 [12]). Here & means logical conjunction and $![\mathsf{E1},\mathsf{E2}]$ forall-quantification over $\mathsf{E1}$, $\mathsf{E2}$.

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:

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

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 $\mathsf{localvar} = \mathsf{value}$ holds in case of an assignment, or the adversary gets to know $\mathsf{outpattern}$, in case it is sent out in $\mathsf{a1}$. Also then the predicate for the succeeding transition $\mathsf{TR2}$ will hold. To construct the recursive definition above, we assume that the CFG is finite and cycle-free. Since in

[1]This restriction is being dropped in ongoing work.

general there may be unbounded loops in the C program, this can only be achieved in an approximate way by fixing a natural number n and unfolding all cycle up to the transition path length n. This is a classical approach in automated software verification, see for example [6]. (Note that this restriction is being dropped in ongoing work.)

The formulas defined above are written into the TPTP file as axioms. 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 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 [8].

3. ASSERTIONS

We explain how one can perform a security analysis with our approach by including security assertions in the program parts to be composed which are generated during the security analysis of these parts. Again, here we focus on confidentiality of data.

A set of security assertions for a program part p consists of statements derived(L, C, E) where L is a list of variables, C is a condition over the variables in L, and E is an expression which may contain free variables from L.

These assertions mean, intuitively, that the set of adversary knowledge is contained in the set of expressions E that can be constructed by instantiating the variables from L with values that themselves can be derived this way for p and which fulfill the condition C.

More formally, a program fragment p has an associated set $\mathcal{L}$ of statements derived(L, C, E) if according to the security analysis explained above, an adversary gets to know only those expressions that can be constructed recursively in the following way.

- For all valuations of the variables v in L fulfilling C and such that knows(v) holds, we have knows($\bar{E}$) for the corresponding valuation $\bar{E}$ of E.

Note that for a single protocol run of p, one can construct a finite set of such assertions (details have to be omitted for space restrictions). By making use of the statements derived(L, C, E) included in the source code, one can now reuse earlier verification results automatically and thereby reduce the verification burden. In particular, the generation of the annotations has also been automated.

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

Figure 1: Abstract Crypto Operations

```
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

We explain our method at 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 C code 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 example, to see

```
void TLS_Client (char* secret)
{
    char Resp_1[MSG_MAXLEN];
    char Resp_2[MSG_MAXLEN];

    // C->S: Init
    // Cert_c_ca => sign(conc(c, k_c), inv(k_ca))
    SendMsg (n, k_c, Cert_c_ca);

    // S->C: Receive Server's respond
    RecvMsg (MSG_MAXLEN, Resp_1);
    RecvMsg (MSG_MAXLEN, Resp_2);

    if (// Check Guards
        (IsEqual(fst(ext(Resp_2, k_ca)), s)) &&
        (IsEqual(snd(ext(dec(Resp_1, inv(k_c)),
                    snd(ext(Resp_2, k_ca)))), n))&&
        (IsEqual(thd(ext(dec(Resp_1, inv(k_c)),
                    snd(ext(Resp_2, k_ca)))), k_c))
      )
       SendMsg(symenc(secret, // C->S: Secret to Server
                 fst(ext(dec(Resp_1,inv(k_c)),
                 snd(ext(Resp_2,k_ca))))));
}
```

Figure 3: C code for client

```
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

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.

One can fix the protocol as suggested in [8] and show that the fixed version is actually secure in the sense that the transmitted secret actually remains confidential.

In order to be able to use this fixed protocol in the system context (for example in a layered composition with other protocols providing further security guarantees), one can now insert the abstract protocol formalization from Fig. 4 as assertions into the protocol source code, as explained above. This means that in further security analyses of the composed protocol, one can reuse verification results which were obtained.

Experiences from applying our approach in a number of industrial case-studies are quite promising with respect to its applicability to practical systems. These case-studies include a biometric authentication system currently in development by an industrial partner (a large German company), where several problems were detected during the development using our method. Also, we are currently applying our method to the open-source project OpenSSL. Results so far are also quite promising. In particular, in these practical applications, the false positives which in principle can appear were not a significant problem. Note also that false negatives do not occur because the abstractions we make are complete in the sense that they do not loose any attacks which are present in our model of the source code. Further details have to be omitted for space restrictions.

4. RELATED WORK

There are other approaches to using FOL ATPs for cryptoprotocol analysis, so far applied mainly on the specification level, including [11, 2]. Some early ideas towards our approach were presented in [9]. With respect to general software verification, relevant work includes [3] describing a system for the automated certification of safety properties of avionics software which uses Hoare-style program verification to generate proof obligations processed by an automated first-order theorem prover (ATP) such as Vampire, Spass, and e-setheo. The obligations are preprocessed and simplified using rewriting. [5] addresses the formal semantics of pointers used in programming languages such as C, together with a tool for construction of formal specifications of programs using pointers.

5. 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. 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. In the current paper, we explained how to make use of assertions to support a practical and efficient security analysis. In all, although our approach is not completely automatic, it turned out to find some actual security flaws at realistic cryptoprotocol implementations.

6. REFERENCES

[1] 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.

[2] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[3] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In D.A. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 198–212. Springer, 2004.

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

[5] G.C. Gannod and B.H.C. Cheng. A formal automated approach for reverse engineering programs with pointers. In *ASE*, pages 219–226, 1997.

[6] 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.

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

[8] 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.

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

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

[11] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In *14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *LNCS*, pages 87–100. Springer, 1997.

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