Tools for Model-based Security Engineering

Jan Jürjens[*]
Competence Center for IT Security, Software & Systems Engineering
Dep. of Informatics, TU Munich, Germany
http://www4.in.tum.de/˜juerjens

ABSTRACT

We present tool-support for checking UML models and C code against security requirements. A framework supports implementing verification routines, based on XMI output of the diagrams from UML CASE tools, and on control flow generated from the C code. The tool also supports weaving security aspects into the code generated from the models. Advanced users can use this open-source framework to implement verification routines for the constraints of self-defined security requirements. We focus on a verification routine that automatically verifies crypto-based software for security requirements by using automated theorem provers.

1. INTRODUCTION

Understanding the security goals provided by software making use of cryptography is one of the major challenges with security-critical systems. Any support to aid secure systems development is thus dearly needed. Towards this goal, the security extension UMLsec for the Unified Modeling Language (UML) [3] allows us to include security requirements as stereotypes with logical constraints. In this paper we present automated tool-support for the analysis of UMLsec models against security requirements by checking the constraints associated with the UMLsec stereotypes. Besides presenting a general, extensible framework for implementing verification routines for the constraints associated with security-critical UML stereotypes, we focus on a plug-in that utilizes an automated theorem-prover (ATP) for first-order logic (FOL) to verify security properties of UMLsec models which make use of cryptography (such as cryptographic protocols), which was explained in [4]. To do so, the analysis routine extracts information from behavioral UML diagrams that may contain additional specific cryptography-related information. If the analysis reveals that there is an attack, an attack generation script written in Prolog generates the attack trace.

Next, one can generate code from the models, in which security aspects can be woven in, as explained in [5]. That the weaving process actually results in software that satisfies the intended security requirements is non-trivial to foresee, because of possible aspect intereferences. We therefore further present a tool for analyzing crypto-based C implementations for security requirements also using ATPs for FOL. This analysis tool is not only intended for C code generated by our code generator, but also for example for legacy code. 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 the ATP. Our approach supports a modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. Also, this way our approach can be applied to code that calls libraries even if the code for the libraries is not (yet) available. Note that 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). Note 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.

Since in particular the code verification part is computationally very challenging, we have developed a further tool plugin in our framework, which combines the ATP-based analysis with the use of the model-checker Spin. This allows one to perform abstractions on the state machines generated from the code which are then verified using the ATP, and verify using the model-checker that the abstracted state machines are actually faithful against the original control flow graphs of the code.

The tools are accessible through a web-interface and available as open-source. They have been validated in several industrial projects including OpenSSL and a biometric authentication system, identifying several major security flaws in software during its industrial development.

[*]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).

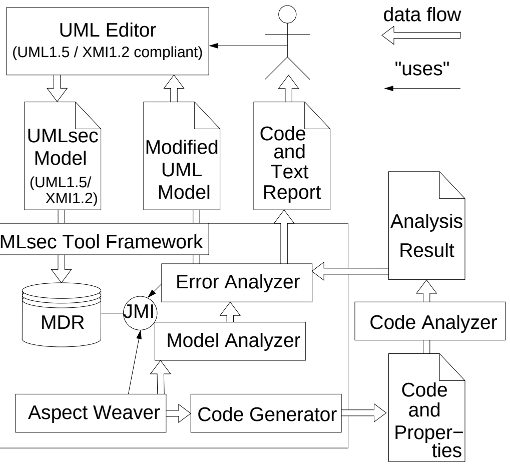

Figure 1: Overall tool-flow of the seCse analysis suite

2. THE VERIFICATION FRAMEWORK

The usage of the framework as illustrated in Fig. 1 proceeds as follows. The developer creates a model and stores it in the UML 1.5/XMI 1.2 file format.[1] The file is imported by the verification framework into the internal MDR repository. MDR is an XMI-specific data-binding library which directly provides a representation of an XMI file on the abstraction level of a UML model through Java interfaces (JMI). This allows the developer to operate directly with UML concepts, such as classes, statecharts, and stereotypes. It is part of the Netbeans project. Each plug-in accesses the model through the JMI interfaces generated by the MDR library, they may receive additional textual input, and they may return both a UML model and textual output. There are two kinds of model analysis plug-ins: The static checkers parse the model, verify its static features, and deliver the results to the error analyzer. The dynamic checkers translate the relevant fragments of the UML model into the input language for example of an ATP. The ATP is spawned by the framework as an external process; its results are delivered back to the error analyzer. The error analyzer uses the information received from the static and dynamic checkers to produce a text report for the developer describing the problems found, and a modified UML model, where the errors found are visualized. Besides the automated theorem prover binding presented in this paper there are other analysis plugins including a model-checker binding and plugins for simulation and test-sequence generation. The developer can then use the aspect weaver to weave in security aspects on the model or into the code that can be generated. The resulting code can then again be analzed for security requirements.

The framework is designed to be extensible: advanced users can define stereotypes, tags, and first-order logic constraints which are then automatically translated to the automated theorem prover for verification on a given UML model. Similarly, new adversary models can be defined.

The user webinterface and the source code of the verification framework is accessible at [12].

3. TRANSLATING UMLSEC DIAGRAMS TO FIRST-ORDER LOGIC FORMULAS

[1]This will be updated to UML 2.0 once the corresponding DTD has been officially released.

We shortly recall from [4] the automated translation of UMLsec diagrams to first-order logic (FOL) formulas which allows automated analysis of the diagrams using ATPs for FOL. Details on the UMLsec notation can be found in [3], although these are not essential here.

We assume that we are given a UML package containing the following kinds of diagrams: A deployment diagram specifies the physical layer of the system, such as system nodes and communication links, and the level of security it provides, using UMLsec stereotypes, such as ≪Internet≫ denoting an Internet communication link. From this, in the security analysis, the adversary model is generated in first-order logic who is able to control certain communication links. Secondly, a class diagram describes the data structure of the system, including the security requirements on the system data, for example using the UMLsec tags {secrecy}, {integrity} and {authenticity} which represent the respective requirements. For the security analysis, from this information the conjecture is derived that is to be checked by the automated theorem prover. The package also contains diagrams specifying the intended behavior of the system, which may include an activity diagram coordinating the components or objects in the package, a sequence diagram specifying interaction between them by message exchange, and statecharts specifying the behavior of single components or objects. The behavioral specifications are compiled to first-order logic axioms giving an abstract interpretation of the system behavior suitable for security analysis.

The translation defines an upper bound for the set of knowledge the adversary can gain using a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain secret as specified in the UMLsec model, one thus has to check whether one can derive $\mathsf{knows}(s)$. For each publicly known expression E, one defines $\mathsf{knows}(E)$ to hold. The fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows (including the use of encryption and decryption) is captured by a logical formula which formalizes the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ expected by one of the protocol participants which validate the condition $cond(exp_1, \ldots, exp_n)$ the protocol participant checks at the receipt of the message, then he can send them to to receive the message returned by that participant in exchange, and then the protocol continues. There are also further formulas formalizing the fact that the adversary may build up knowledge by applying cryptographic algorithms. With this formalization, a data value s is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulas defined by a protocol. This way, the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities and the message order). This means that one will find all possible attacks, but one may also encounter "false positives", although this has not happened yet with any real examples. The advantage is that this approach is rather efficient. The conjecture, for which the ATP will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value s is to be kept secret, the conjecture is $\mathsf{knows}(s)$.

Attack Generation. 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 that 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 durch its search, which can then be queried).

4. CODE ANALYSIS

We explain the code analysis part of the framework. From the control flow graph generated from the source code using the aiCall tool [1], the seCse tool constructs the FOL axioms giving an abstract interpretation of the system behavior suitable for security analysis. Technically, this is realized via the export format GDL of the aiCall tool for the control flow graph. For space restrictions, we explain the translation only for a simplified fragment of C without loops and concurrency. We use standard transformation to simplify the translation from the state machines to logic.

side effects Side effects are transformed away as usual. For example, $i++$ is substituted by $i = i + 1$, the statement $a+ = 20$ is replaced by $a = a + 20$, and $b = ++k$ by the command sequence $k = k + 1; b = k$.

static single assignment The program is transformed to the *static single assignment (SSA)* format as usual. For example, the command sequence $k = k + 1; b = k$ is replaced by $k1 = k0 + 1; b = k1$.

dereferencing pointers To transform away the use of pointers, we use a standard method.

Although the construction of the control flow graph from a C program is essentially standard, the *input* and *output patterns* are somewhat special. They are needed because of the emphasis on interaction when verifying cryptoprotocols. An *input pattern* consists of a message name msg and a list of variables which will be assigned values when a message with name mgs is received over the network. We use code annotations (defined below) to define which input variables store the incoming arguments of which messages, and which functions are used to receive them. Similarly, an output pattern consists of a message name msg and a list of expressions, that are at run-time evaluated to values which are sent on the network as arguments of the message msg. Again we use code annotations defined below to specify which functions take care of sending out the messages. Lastly, we may use other kinds of annotations to map an assignment assgmt of an expression to a variable in C to a logical predicate p_{assgmt} on the corresponding logical variable. inpattern may be empty and condition equal to true where they are not needed. The state machine constructed in this way can now be translated to a FOL formula as in the previous section.

Abstraction through Annotations. An annotation, as defined in Fig. 2, starts with the key word //@C2SM_ANN followed by the name of the function or variable. Then the keyword //@C2SM_TRANS follows (optionally) which

```
//@C2SM_ANN (<<function name>>)
//@C2SM_TRANS (<<trigger>>; <<guard>>; <<effect>>)
//@C2SM_INSERT (<<value>>)
//@C2SM_AXIOMS
// <<FOL axioms>>
//@C2SM_AXIOMS_END
```

Figure 2: Code Annotations

specifies the trigger, the guard, and the effect of a transition which should be inserted in the statemachine where the function is called or variable is used. Here one can refer to the arguments of the function which appear at the occurence which should be replaced by identifying them as functn_i where functn is the name of the function as specified with the key word //@C2SM_ANN and i the number of the argument. The keyword //@C2SM_INSERT specifies an expression that should be inserted at the place of the function call as its return value, or in place of the variable, respectively. The definition ends with the optional keyword //@C2SM_AXIOMS which allows one to insert FOL formulas axiomatizing the expressions used in the state machine transition and the inserted value.

Standard function annotations map functions in the standard libraries to their representations in the state machine. For example, memset(), memcpy(), strcpy(), and strncpy() are each mapped to an effect at a transition which assigns the value given as their second argument to the variable given as their first argument (and abstracts from the third argument, giving the size of the variable value). The C function memcmp() is mapped to the two-argument state machine function equal_num() which returns 0 if its two arguments evaluate to the same value and which abstracts from the third argument of memcmp() defining the size of the arguments. The C functions !memcmp(), strcmp(), and strncmp() are mapped to the function equal() returning true if its two arguments are the same and false otherwise (and also abstracts from the argument sizes). To be feasible, one also needs to completely abstract away irrelevant parts of the code. For example, since in our security analysis on the given level of abstraction we are not concerned with memory allocation, the C commands malloc(), calloc(), realloc(), and free() are abstracted away using our annotations.

Modular Verification. We explain how one can perform a modular security analysis by including security assertions in the program parts generated during the security analysis. 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 that the set of adversary knowledge is contained in the set of expressions E constructed by instantiating the variables from L with values that themselves can be derived this way for p and which fulfill the condition C. To analyze a program fragment p carrying a set of assertions $\mathcal{L}$ one takes the formulas generated from the approach in Sect. 4 and adds for each the assertion of the form derived(L, C, E) an axiom of the form

```
![v1,...,vn]: knows(v1) & ... & knows(vn)
          & C(v1, ..., vn) => knows (E)
```

(where L is the list of variables v1, ..., vn and C(v1, ..., vn) the instantiation of C with the variables v1, ..., vn).

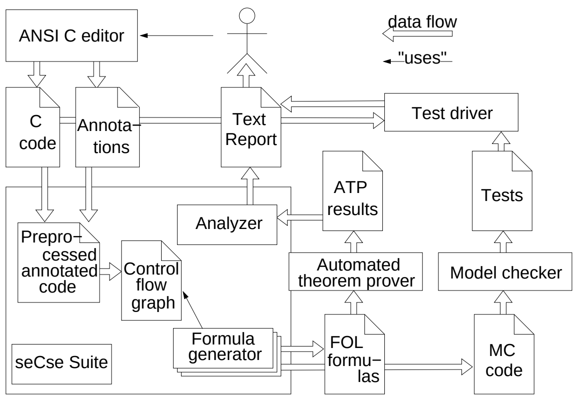

Figure 3: Analysis part of the seCse suite

5. AUTOMATED THEOREM PROVERS VS. MODEL-CHECKERS

Providing automated verification techniques for state machines specifying distributed systems is challenging in several aspects: On the one hand, the non-deterministic interleaving of the states and actions of parallel substates and of different statecharts executed in parallel leads to a state space explosion. On the other hand, complex data structures add further to the complexity. Lastly, state machines derived from a practical context (such as control flow graphs of implementations) can be quite large. At the hand of the verification of security properties, we use automated theorem provers for first-order logic for automated verification of state machines generated from code in a combination with using model-checkers for security-sensitive model-based testing to ensure the abstractions that have to be made are valid. The tool-flow for this extension of our framework is given in Fig. 3.

6. RELATED WORK

For space restrictions, we can each only point to a few exemplary works in the fields of tool support for security, UML, and software verification.

[10] formalizes the well-known BAN logic in FOL and uses the ATP SETHEO to proof statements in the BAN logic. BAN logic is a modal belief logic used to formulate the beliefs of protocol participants during protocol execution. [8] uses the OTS/CafeOBJ tool to verify the security protocol TLS by means of equational reasoning.

In terms of UML verification, [11] presents a tool which uses the FDR model-checker to verify a variant of sequence diagrams via a translation to the process algebra CSP.

In the field of software model-checking, [2] presents the Bandera toolset using model-checkers to reason about correctness requirements of Java programs. More precisely, the tool "provides tool support for defining and managing collections of requirements for a program, for extracting compact finite-state models of the program to enable tractable analysis, and for displaying analysis results to the user through a debugger-like interface". Further developments in this line of work include the Bogor model-checking framework which provides "an extensible input language for defining domain-specific constructs and a modular interface design to ease the optimization of domain-specific state-space encodings, reductions and search algorithms" [9].

With respect to tool frameworks in general, related work includes [7], which offers a tool framework which goes beyond our framework presented here in that it is open to non-UML-based tools.

In a previous tool demo, a small part of the proposed tool demo was presented in [6] (consisting of the model analysis part only).

7. CONCLUSION

We use automated theorem provers for first order logic to understand the security requirements provided by UML models and C code implementations of crypto-based software. Our approach constructs a logical abstraction of the annotated models or code which can be used to analyze them for security properties (such as confidentiality) with ATPs. It supports a modular security analysis of crypto-based implementations using assertions in the source code. Although our approach is not completely automatic and requires some effort for annotating the code to make it scale, it turned out to be applicable with reasonable effort even in large software projects, as demonstrated at the hand of the OpenSSL suite and a biometric authentication system. We keep the annotation effort bounded by providing an annotated standard library (although these do not cover user functions). We are also currently exploring ideas from automated discovery of proof invariants in order to partially automate annotating the code.

8. REFERENCES

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

[2] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *22th International Conference on Software Engineering (ICSE 2000)*, pages 762–765. IEEE Computer Society, 2000.

[3] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[4] 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.

[5] J. Jürjens and S.H. Houmb. Dynamic secure aspect modeling with UML: From models to code. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML 2005)*, LNCS. Springer, 2005.

[6] J. Jürjens and P. Shabalin. Tools for secure systems development with UML: Security analysis with ATPs. In *FASE 2005*, LNCS, Edinburgh, 2-10 April 2005. Springer.

[7] T. Margaria, R. Nagel, and B. Steffen. jETI: A tool for remote tool integration. In N. Halbwachs and L.D. Zuck, editors, *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 557–562. Springer, 2005.

[8] K. Ogata and K. Futatsugi. Equational approach to formal analysis of TLS. In *25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 795–804. IEEE Computer Society, 2005.

[9] Robby, M.B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'03)*, pages 267–276. ACM, 2003.

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

[11] Jun Sun and Jin Song Dong. Model checking live sequence charts. pages 529–538. IEEE Computer Society, 2005.

[12] UMLsec tool, 2004. http://www4.in.tum.de/csduml/interface. Access: User: Tooluser, password: Imayusethis.