Tools for Traceable Security Verification

Jan Jürjens and Yijun Yu
Computing Department, The Open University, UK
http://mcs.open.ac.uk/{jj2924,yy66}

Andreas Bauer
Computer Sciences Lab, Australian National University
http://users.rsise.anu.edu.au/~baueran

Abstract

Dependable systems evolution has been identified by the UK Computing Research Committee (UKCRC) as one of the current grand challenges for computer science. We present work towards addressing this challenge which focusses on one facet of dependability, namely data security: We give an overview on an approach for model-based security verification which provides a traceability link to the implementation. The approach uses a design model in the UML security extension UMLsec which can be formally verified against high-level security requirements such as secrecy and authenticity. An implementation of the specification can then be verified against the model by making use of run-time verification through the traceability link. The approach supports software evolution in so far as the traceability mapping is updated when refactoring operations are regressively performed using our tool-supported refactoring technique. The proposed method has been applied to an implementation of the Internet security protocol SSL.[a]

[a]This work was partially supported by the Royal Society within the project Model-based Formal Security Analysis of Crypto-Protocol Implementations.

Keywords: Software Engineering, Security Analysis, Dependable Systems Evolution

1. INTRODUCTION

There has been successful research over the last years to provide security assurance tools for the lower abstraction levels of software systems. However, these tools usually search for specific security weaknesses, such as buffer overflow vulnerabilities. What is so far largely missing is automated tool support which would support security assurance throughout the software development process, starting from the analysis of software design models (e.g., in UML) against abstract security requirements (such as secrecy and authenticity), and tracing the requirements to the code level to make sure that the implementation is still secure.

This article presents a tool-supported approach that supports such a software security assurance, which can be used in the context of an approach called Model-based Security Engineering (MBSE) that has been developed over the last few years (see e.g., [7, 8] for details and Fig. 1a for a visual overview). In this approach, recurring security requirements (such as secrecy, integrity, authenticity and others) and security assumptions on the system environment, can be specified either within a UML specification, or within the source code (Java or C) as annotations. One can then formally analyze the models against the security requirements using model-checkers and automated theorem provers for first-order logic (see Fig. 1b) and [11, 17]). The approach has been used successfully in a number of industrial applications (e.g., at BMW [2] and O_2 (Germany) [9]).

We also present an extension of this approach which allows one to use runtime verification to increase one‘s confidence that the implementation securely implements the security properties previously demonstrated at the specification level. Note that our goal is not to provide a full formal verification of the correctness of the implementation against the specification, but to raise one‘s confidence in its security by demonstrating that certain particularly security-relevant parts (such as the checking of cryptographic certificates) are securely implemented. We demonstrate the approach at the hand of an application to an implementation of the Internet security protocol SSL.

In our experience, it is non-trivial to link the model correctly to its implementation to ensure that not only the model but also the implementation satisfies the relevant security requirements. As the implementation or the used libraries evolve, the instrumentation may not anymore guarantee the correct link to the protocol design. It is therefore important to have a way to perform refactoring steps in a traceable way. One goal of our work is thus to maintain traceability between the design

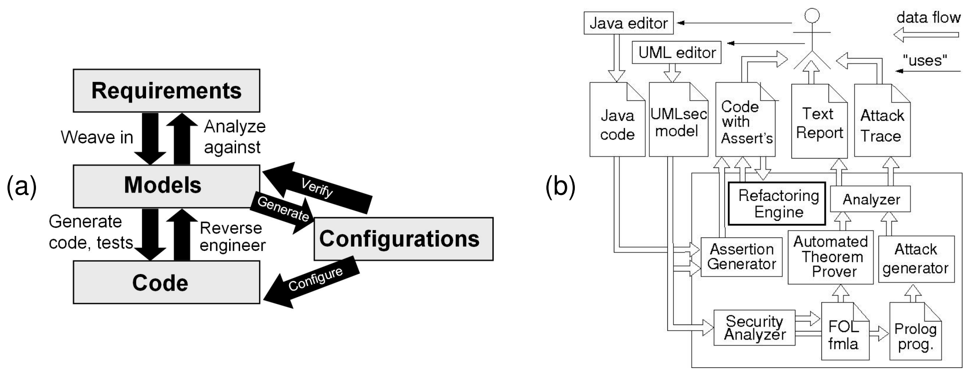


FIGURE 1: a) Model-based Security Engineering; b) MBSE Toolkit

and the implementation of a crypto-based software through a dedicated software refactoring approach. Our approach supports traceability in various stages of development:

- Traceability from security requirements to design: one includes security requirements as annotations into UML models and automatically verifies the models against these requirements (Sect. 2).
- Traceability from secure design to implementation: with the help of refactoring operations, cf. Sect. 3.2.
- Security verification of the implementation: using run-time verification, cf. Sect. 3.3.

Analyzing one implementation of crypto-based software and setting up security monitors are time-intensive. Effort for similar tasks on another implementation of the same design can be greatly alleviated by reusing the traceability established between the design and the previous implementation. Maintaining such traceability using our refactoring tool requires only to adjust parameters from the previous refactoring specifications. Book-keeping changes explicitly makes such program understanding repeatable and regressive.
Specific novel aspects of this approach include a new combination of verification techniques including model-level security analysis, model-code traceability robust under change, and run-time security verification, and applying it to implementations of crypto-protocols.
From a broader point of view, the goal of this work is to allow the use of formally based verification techniques (such as automated theorem provers and run-time verification) in practice by encapsulating them in an industrially accepted development approach (UML). We hope to thus contribute to dealing with the challenges faced when trying to use formal methods in a practical environment (cf. e.g. [6, 4, 16] for relevant discussions).
The approach presented here has to be seen in the context of other approaches to model-based security based on UML developed over the last few years (see [7] for a more complete overview). There are also many other relevant approaches to model-based assurance of security-critical systems which are not based on UML, such as [14, 18]. The work presented here differs from that in that it is based on a modelling notation routinely used in industry today to facilitate uptake in practice, and that it includes a link to implementation level security assurance. Also related are several approaches to formally verifying implementations of crypto protocols developed recently, such as [10, 5, 3]. The current work is different in that it does not verify the implementation directly against security properties, but verifies specification models against security properties, and then verifies the implementation against the models with a focus on the security properties, using techniques including run-time security verification. The idea of this two-step verification process is to facilitate application to complex legacy software.

2. MODEL-BASED SECURITY ENGINEERING: MODEL ANALYSIS

2.1. Model-based Security Engineering

Model-based Security Engineering [7, 8] provides a soundly based approach for developing security-critical software where recurring security requirements (such as secrecy, integrity, authenticity and others) and security assumptions on the system environment, can be specified either within a UML specification, or within the source code as annotations (cf. Fig. 1a). Various

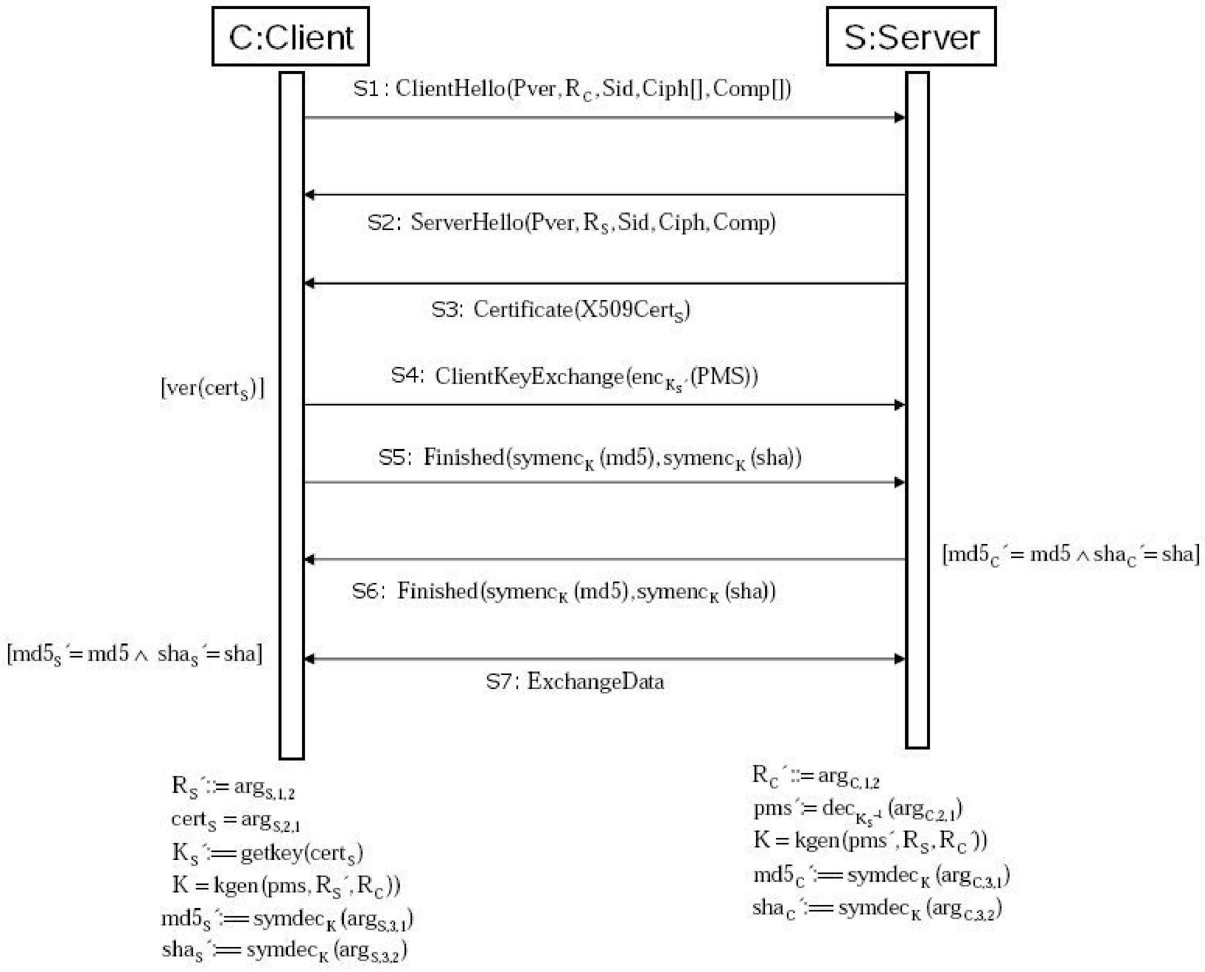

C:Client
S:Server
S1 : ClientHello(Pver, R_C, Sid, Ciph[], Comp[])
S2: ServerHello(Pver, R_S, Sid, Ciph, Comp)
S3: Certificate(X509$Cert_S$)
S4: ClientKeyExchange($enc_{K_S'}$(PMS))
[ver($cert_S$)]
S5: Finished($symenc_K$(md5), $symenc_K$(sha))
[$md5_C' = md5 \wedge sha_C' = sha$]
S6: Finished($symenc_K$(md5), $symenc_K$(sha))
[$md5_S' = md5 \wedge sha_S' = sha$]
S7: ExchangeData
$R_S' ::= arg_{S,1,2}$
$cert_S = arg_{S,2,1}$
$K_S' := getkey(cert_S)$
$K = kgen(pms, R_S', R_C)$
$md5_S' := symdec_K(arg_{S,3,1})$
$sha_S' := symdec_K(arg_{S,3,2})$
$R_C' ::= arg_{C,1,2}$
$pms' := dec_{K_S^{-1}}(arg_{C,2,1})$
$K = kgen(pms', R_S, R_C')$
$md5_C' := symdec_K(arg_{C,3,1})$
$sha_C' := symdec_K(arg_{C,3,2})$

FIGURE 2: Handshake protocol of SSL3 using RSA and Server Authentication

analysis plugins in the associated UMLsec tool framework [17] (Fig. 1b) generate logical formulas formalizing the execution semantics and the annotated security requirements. Automated theorem provers and model checkers automatically establish whether the security requirements hold. If not, a Prolog-based tool automatically generates an attack sequence violating the security requirement which can be examined to determine and remove the weakness. Thus we encapsulate knowledge on prudent security engineering and make it available to developers who may not be security experts. Since the analysis that is performed is too sophisticated to be done manually, it is also valuable to security experts. Part of the MBSE approach is the UML extension UMLsec for secure systems development which allows the evaluation of UML specifications for vulnerabilities using a formal semantics of a simplified fragment of the UML. The UMLsec extension is given in form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate the security requirements and assumptions. *Constraints* give criteria that determine whether the requirements are met by the system design, by referring to a precise semantics of the used fragment of UML. The security-relevant information added using stereotypes includes security-relevant information covering the following aspects:

- Security assumptions on the physical system level, for example the stereotype «encrypted», when applied to a link in a UML deployment diagram, states that this connection has to be encrypted.
- Security requirements on the logical level, for example related to the secure handling and communication of data, such as «secrecy» or «integrity».
- Security policies that system parts are required to obey, such as «fair exchange» or «data security».

The UMLsec tool-support in Fig. 1b can then be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [17, 8]. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes. The semantics for the fragment of UML used for UMLsec is defined in [7] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces and UML-type communication mechanisms. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined. To support stepwise development, one can show secrecy, integrity, authenticity, and secure information flow to be *preserved* under refinement and the composition of system components, and the approach also supports the secure development of layered security services (such as layered security protocols). See [7] for more information on the above.

2.2. Analyzing Cryptographic Protocols

For the remainder of this paper, we concentrate on applying model-based security engineering to the special case of crypto protocols, which is well suited to present our approach in a short overview paper since this is a compact piece of highly security-critical software which is nevertheless non-trivial to design and implement correctly.
We applied the approach presented in this paper to the core part of the SSL 3.0 handshake protocol given in Fig. 2 together with the open-source Java implementation JESSIE (http://www.nongnu.org/jessie) of the Java Secure Socket Extension as will be presented as a running example throughout this paper. SSL is the de facto standard for securing http connections, which however has been the source of several significant security vulnerabilities in the past and is therefore an interesting target for a security analysis. In this paper, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (cf. Fig. 2).
Using UML sequence diagrams, each message in a crypto-protocol is specified by giving the sender, the receiver, the message, and possibly a precondition (in equational first-order logic) which has to be fulfilled so that the message is sent out. An example can be seen in Fig. 2 which shows version 3 of the SSL protocol, the de facto standard for securing http connections, which however has been the source of several significant security vulnerabilities in previous versions and is therefore an interesting target. We concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication. The protocol participants (here the instances C of class Client and S of class Server) are represented by vertical boxes, and the messages between them are represented by arrows. A logical expression next to an outgoing arrow is the guarding constraint that needs to be checked by the relevant protocol participant before the message is sent out. The assignments specified below the model in Fig. 2 describe how the data that is received should be used by the receiving instance. Here the expression $\mathsf{arg}_{i,n,p}$ corresponds to the pth element of the nth message sent by the object instance i. For example, $R_S`{:}{=}{=}\mathsf{arg}_{S,1,1}$ means that the random number $R_S$, which was sent by the server in the message ServerHello, is stored in the variable $R_S`$ at the Client, after receiving the message. In the guards, the local designations are used. The guard [ver(cert$_S$)] means that the certificate X509Cert_s previously received from the server must be verified. The guards $[\mathsf{md5}_S' = \mathsf{md5} \wedge \mathsf{sha}_S' = \mathsf{sha}]$ and $[\mathsf{md5}_C' = \mathsf{md5} \wedge \mathsf{sha}_C' = \mathsf{sha}]$ express the condition that the hash values of the instance which receives a Finished message have to agree with the hash values of the other instance. K is the symmetric session key which is created separately at each of the protocol partners, making use of the pre-master secret PMS. The values md5 and sha used as message arguments are created by the sender of the respective message by using the MD5 resp. SHA hash algorithm over the message elements received so far. ExchangeData represents the communication of data over the established channel once the handshake protocol is finished and also has an associated guard. For simplification, we specify the encryption of a compound message as the concatenation of the encryptions of the separate message elements (for example Finished(symenc$_K$(md5), symenc$_K$(sha)) rather than Finished(symenc$_K$(md5::sha))); we assume that type or message confusion attacks are ruled out using the usual protocol design rules not under investigation here.

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

FIGURE 3: Abstract Crypto Operations

As usual in the formal analysis of crypto-based software, the crypto algorithms are viewed as abstract functions. The messages that can be created from these algorithms are then as usual formally defined as a term algebra generated from ground data such as variables, keys, nonces and other data using symbolic operations including those in Fig. 3. There, the symbols E, E', and E'' denote terms inductively constructed in this way. Note that the key and random generation methods are not part of the crypto term algebra in Fig. 3 but are formalized implicitly in the logical formula by introducing new constants representing the keys and random values (and making use of the inv(E) operation in the case of generateKeyPair()). In the 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, head(), and tail().
A crypto-protocol like the one in Fig. 2 can then be verified for the relevant security requirement such as secrecy and authenticity using the UMLsec tools presented above, which rely on a translation from the UMLsec sequence diagram to a security-sensitive interpretation in first-order logic based on the Dolev-Yao attacker model as explained in [8], which is then verified using automated theorem provers for FOL. The idea is here 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.

We now explain how to analyze the UMLsec specification by making use of our translation from cryptographic protocols specified as UML sequence diagrams to FOL formulas which can be processed by the automated theorem prover e-SETHEO. The formalization automatically derives an upper bound for the set of knowledge the adversary can gain. The usage of the FOL generation explained in the following is complementary to the model-level security analysis mentioned above: Although using the approach described earlier one can make sure that the specification is secure, this does not imply that the implementation is secure as well, since we cannot make any assumptions on how it was constructed (as we would like to deal in particular with legacy implementations such as openSSL). The FOL-based approach described in the following therefore has the goal to verify the UML sequence diagram against for the given security requirements such as secrecy.
The idea is to use 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, the FOL formalization will thus compute all scenarios which would lead the attacker to derive $\mathsf{knows}(s)$. The FOL rules generated for a given UMLsec specification is defined as follows. 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 the formula in Fig. 4.
For our purposes, a sequence diagram is essentially a sequence of command schemata of the form *await event e – check condition g – output event e'* represented as *connections* in the sequence diagrams. Connections are the arrows from the life-line of a source object to the life-line of a target object which are labeled with a message to be sent from the source to the target and a guard condition that has to be fulfilled. Suppose we are given a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ in a sequence diagram with

$$\begin{aligned}\forall E_1, E_2.\Big(&\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\Big)\\ &\wedge\Big(\mathsf{knows}(E_1 :: E_2) \Rightarrow \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\Big) \wedge \Big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \Rightarrow \mathsf{knows}(E_1)\Big)\\ &\wedge\Big(\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1)\Big)\end{aligned}$$

FIGURE 4: FOL rules for attacker knowledge generation

$$\mathsf{PRED}(l) = \forall exp_1, \ldots, exp_n.\Big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n) \wedge\ cond(exp_1, \ldots, exp_n) \Rightarrow \mathsf{knows}(exp(exp_1, \ldots, exp_n) \wedge\ \mathsf{PRED}(l')\Big)$$

FIGURE 5: FOL rule for attacker interaction

$\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$, where the parameters arg_i of the guard and the message are variables which store the data values exchanged during the course of the protocol. Suppose that the connection l' is the next connection in the sequence diagram with $\mathsf{source}(l') = \mathsf{source}(l)$. For each such connection l, we define a predicate $\mathsf{PRED}(l)$ as in Fig. 5. If such a connection l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Fig. 5. The formula formalizes the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. This way, the adversary knowledge set is approximated from above (e.g. one abstracts away from the message sender and receiver identities and the message order). In particular, one will find all possible Dolev-Yao type attacks on the protocol, but execution traces may also be generated that are not actually executable for a valid implementation. This has however not been a problem in practical applications of the approach. For each object O in the sequence diagram, the generated FOL formulas will now compute scenarios which, if successful, may constitute an attack on the protocol against the security properties required of the protocol.
We used the UMLsec tools to verify the UMLsec model of the SSL protocol (cf. Fig. 2) against the relevant security requirements such as secrecy and authenticity. Verifying secrecy of a value s can be done by checking whether the statement knows(s) is derivable from the FOL formulas generated from the protocol specification. Verifying authenticity involves additional correspondence predicates which ensure that certain authenticity checks are carried out correctly (details have to be omitted here for space restrictions). In each case, the properties were proved within less than a minute. E.g., the verification of the secrecy of the master secret communicated in the SSL protocol took 2 seconds.

3. LINKING MODELS TO CODE

We explain how to link the formally verified specification to a crypto-based implementation which may not be trustworthy (for example, which might have been implemented insecurely from a secure specification, either maliciously or accidentally). The approach is currently focusing on Java as the implementation language. We then explain how to use the link to perform run-time verification of the implementation.

3.1. The SSL Implementation JESSIE

We applied the approach sketched below to the implementation of the Internet security protocol SSL in the project JESSIE, which is an open-source implementation of the Java Secure Sockets Extension (JSSE). The whole JESSIE project currently consists of about 5 MB of code, but the part directly relevant to SSL consists of less than 700 KB in about 70 classes.
As explained above, the crypto algorithms are viewed as abstract functions. In our application, these abstract functions represent the implementations from the Java Cryptography Architecture (JCA). The messages that can be created from these algorithms are then as usual formally defined as a term algebra generated from ground data such as variables, keys, nonces, and other data using symbolic operations. These symbolic operations are the abstract versions of the cryptographic algorithms. Note that the cryptographic functions in the JCA are implemented as several methods, including an object creation and possibly initialisation. Relevant for our analysis are the actual cryptographic computations performed by the digest(), sign(), verify(), generatePublic(), generatePrivate(), nextBytes(), and doFinal() methods (together with the arguments that are given beforehand, possibly using the update() method), so the others are essentially abstracted away. As noted above, the key and random generation methods generatePublic(), generatePrivate(), and nextBytes() are not part of the crypto term algebra but

Message name	Class of Message Type	Message Type
ClientHello	ClientHello	CLIENT_HELLO
ServerHello	ServerHello	SERVER_HELLO
Certificate*	Certificate	CERTIFICATE
ClientKeyExchange	ClientKeyExchange	CLIENT_KEY_EXCHANGE
Finished	Finished	FINISHED

FIGURE 6: Data for the Handshake message

are formalized implicitly in the logical formula by introducing new constants representing the keys and random values (and making use of the inv(E) operation in the case of generateKeyPair()).

In our particular protocol, setting up the connection is done by two methods: doClientHandshake() on the client side and doServerHandshake() on the server side, which are part of the SSLsocket class in jessie − 1.0.1/org/metastatic/jessie/provider. After some initialisations and parameter checking, both methods perform the interaction between client and server that is specified in Fig. 2. Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake() rp. doServerHandshake() methods. The associated data is given in Fig. 6.

We must now determine for the individual data how it is implemented on the code level, to then be able to verify that this is done correctly. We explain this exemplarily for the variable randomBytes written by the method ClientHello to the message buffer. By inspecting the location at which the variable is written (the method write(randomBytes) in the class Random), we can see that the value of randomBytes is determined by the second parameter of the constructor of this class (see Fig. 7).

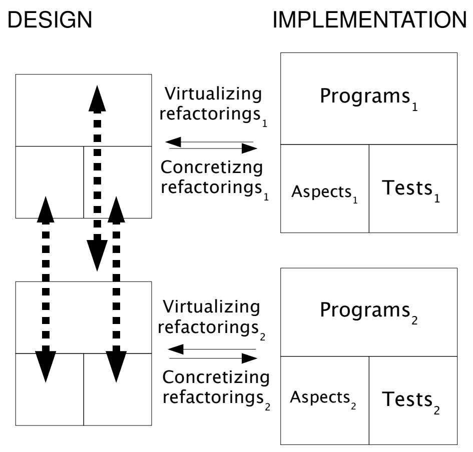
Therefore the contents of the variable depends on the initialisation of the current random object and thus also on the program state. Thus we need to trace back the initialisation of the object. In the current program state, the random object was passed on to the ClientHello object by the constructor. This again was delivered at the initialisation of the Handshake object in SSLSocket. doClientHandshake() to the constructor of Handshake. Here (within doClientHandshake()), we can find the initialisation of the Random object that was passed on. The second parameter is generateSeed() of the class SecureRandom from the package java.security. This call determines the value of randomBytes in the current program state. Thus the value randomBytes is mapped to the model element R_C in the message ClientHello on the model level. For this, java.security.SecureRandom.generateSeed() must be correctly implemented. To increase our confidence in this assumption of an agreement of the implementation with the model (although a full formal verification is not the goal of this paper), all data that is sent and received must be investigated. This will be considered in the next section.

```
Random(int gmtUnixTime, byte[] randomBytes) {
  this.gmtUnixTime = gmtUnixTime;
  this.randomBytes = (byte[])randomBytes.clone(); }
```

FIGURE 7: Constructor for random

3.2. Tools for Maintaining Model-Code Traceability

Software refactoring [13] changes the internal structure of an implementation without changing its external behavior. In this work, we use refactoring scripts to maintain traceability between a design and its evolving implementations. Modern IDE's such as Eclipse support refactoring by automated scripts, allowing users to perform, record and replay refactoring steps as if they were basic editing operations. The advantage over traditional editing scripts is that refactoring scripts preserve the behavioral semantics of the program. However, such basic refactoring support is inadequate for our purpose, namely to maintain traceability between changing code bases. For example, adding or deleting a single space can make the Extract Method operation inapplicable. To enhance reusability of refactoring operations regarding such kind of code changes, we extended the Eclipse Refactoring Language Toolkit (LTK) using a new approach to make the operating context of refactoring more tolerant to changes. To ease specifying these refactoring operations, we also implemented a utility to convert refactoring scripts saved from Eclipse into our specification language.
Conceptually, two kinds of mappings are defined to maintain traceability. The first one maps any


DESIGN
IMPLEMENTATION
Virtualizing refactorings$_1$
Concretizng refactorings$_1$
Programs$_1$
Aspects$_1$
Tests$_1$
Virtualizing refactorings$_2$
Concretizing refactorings$_2$
Programs$_2$
Aspects$_2$
Tests$_2$

Symbols	Program entities	Identif.	Refactoring op.
1. C	clientHello	C	rename.type
2. S	serverHello	S	rename.type
3. P_{ver}	session.protocol version	P_ver	extract.temp
4. R_C	clientRandom	R_C	rename.local.variable
R_S	serverRandom	R_S	rename.local.variable
5. S_{id}	sessionId	S_id	rename.field
	sessionId	S_id	rename.local.variable
6. Ciph[ ]	session.enabledSuites	Ciph	extract.temp
7. Comp[ ]	comp	Comp	extract.temp
8. Veri	Lines 1518–1557	Veri	extract.method

FIGURE 8: a) Traceability for reuse; b) Mapping symbols to program entities

symbolic name that appears in the design model to an identifier at the implementation level. The second maps any identifier to a method, that can be pointcut by a security aspect. We aim to use refactoring scripts for maintaining such traceability: they guarantee that the behavior of the program is preserved as far as expressed in the traceability links to the model level. We can apply the mapping in a round-trip fashion (1) to convert the identifiers/methods to names at the design level and (2) to convert the names on the design level to identifier/method names in the implementation.
Though refactoring, the first mapping is set up between the symbolic names in a design model and the identifier names in the program. A sequence of refactoring operations can be used to transform every occurrence of a selection of program elements into their counterpart design elements. In general there may not be a straightforward one-to-one mapping between the abstract symbols and the way they are implemented. We make use of refactoring actions to refactor the relevant code fragments in a semantics-preserving way to provide a program entity that can be linked to the given model-level symbol, in order to facilitate constructing the traceability mapping. To illustrate this, Fig. 8 b) presents some instances of such a mapping for our example implementation. The first column shows the names of symbols as used in the crypto protocol model. The second column shows the names of the corresponding program entities in the implementation. The third column shows the identifiers that are the target names of the refactoring operations. The type of the refactoring operation is shown in the last column. The implementation and execution of these refactoring operations is done using refactoring scripts. These scripts only allow a limited kind of refactoring which guarantees that the behavior of the program is preserved while facilitating the construction of the traceability links to the model level (e.g. renaming identifiers in a way that is ensured not to create any conflicts). Using the refactoring scripts, we can apply the mapping in a round-trip fashion (1) to convert the program entities to names on the design level and (2) to convert the names on the design level to names in the implementation. Each refactoring operation is declared as a transformation from a program entity (a collection of executable statements or declarations) to a symbolic entity which is named after the corresponding symbol in the design model. For example, the clientRandom variable is mapped to the symbol R_C in the protocol.

Having the first mapping between symbols and implementation established through refactoring, the second mapping is used to define joinpoints in terms of the symbol names and the joinpoint model in aspectJ (methods and fields). Such joinpoints must be aware of the context of the method invocations or field accesses. When identifiers are methods or fields, then they can already be matched by pointcut expressions in the aspects. Otherwise, more refactoring operations need to be performed to prepare for AOP instrumentation. As the joinpoint model in aspectJ does not support the instrumentation of a group of statements inside a method, for example, it is necessary to apply more refactoring operations such as extract.method to group these statements into a method. Having the joinpoints symbols refactored as methods and fields, they can now be used to define aspect pointcut expressions. As long as program changes are captured by changing the refactoring scripts, one can maintain the pointcut expression unchanged. Similarly, if one wants to apply the same aspect to a different library where the symbols are implemented differently, the reusability of such aspects eliminates the need to change the definition of the aspects. This effort for maintaining the traceability has a payoff only when a mapping can be used to express aspects which otherwise would be non-reusable. Since refactoring operations are semantics-preserving program transformations, these mappings can be performed selectively on the joinpoints that are immediately useful for the aspects.

One can reuse the traceability information discovered when linking the implementation to the UML model for example if one wants to apply the refactoring operations defined for one version of the implementation to a different version of that implementation, or to a different library. To this end, we create a refactoring plugin that can apply parameterized refactoring operations[1]. Our refactoring tool is implemented on top of LTK refactoring plugins, which supports languages beyond Java. In order to limit the changes to existing refactoring engine, we invoke the context-specific refactoring operations in JDT by instantiating a scripting template with the parameters derived from our specifications.

Integration. Our tool delegates the domain-specific (here Java) refactoring integration tasks to LTK in Eclipse. We also support both *interactivity* and *transparency* for programmers to preview the effects of refactoring if they choose to, and to avoid manually constructing the specification from the saved refactoring history in Eclipse. The implementation of our refactoring plugin adds two command buttons to the Eclipse GUI, one of them performs all refactoring operations automatically, while the other brings up a dialogue for each operation to preview the effects of refactoring. This allows us to check for any potential maintenance problems of the operation. We also implemented a headless tool to invoke the functionality of the automated button as a RCP command. The argument of the command provides the name of a refactoring specification file. In this way, our tools are integrated into a customized continuous integration (CI) process using cruisecontrol[2]. In the process, our security analysis using the automated refactoring tool (ART) and UMLsec analysis tool is parallel to the typical software development process while two synchronizations happen for each iteration of development. When a programmer commits changes to the code repository, the CI monitor detects the change and triggers our automated refactoring. When a designer commits a change to the design artifacts, the traceability links established on basis of the previous design might break, therefore a warning will be issued to report a check on the traceability refactoring operations. If such design-level changes happened inside Eclipse-based UML design tools, another utility transformation program we implemented converts an XML-based refactoring script from Eclipse IDE into our own specification language.

3.3. Monitoring security properties

As explained above, a crypto-protocol like the one specified in Fig. 2 can be verified at the specification level for the relevant security requirement such as secrecy and authenticity using the UMLsec tools [8]. We now explain how one can then use runtime verification to increase one‘s confidence that the implementation securely implements the security properties previously demonstrated at the specification level. Note that our goal is not to provide a full formal verification of the correctness of the implementation against the specification, but to raise one‘s confidence in its security by demonstrating that certain particularly security-relevant parts (such as the checking of cryptographic certificates) are securely included into the implementation context. However, run-time verification can provide a higher level of assurance for crypto based software than for example model-based testing: For maximal assurance we would have to aim for full test coverage for the system traces that might lead to an attack. Full test coverage is in general not achievable for highly interactive and complex software like cryptographic protocols. Also, cryptography is required to be secure against brute force searching attacks, which also prevents full test coverage when testing crypto based software.

To our knowledge, this is the first approach to run-time verification of crypto protocol implementations, although there has been other work on run-time security verification using LTL, such as [15, 12].

[1]These automated refactoring tools (ART), including their source code and examples in the paper, can be downloaded from the project subversion repository linked from [17].
[2]See Martin Fowler, Continuous integration, http://www.martinfowler.com/articles/continuousIntegration.html

We first need to determine how important elements at the model level are implemented at the implementation level. This can be done in the following three steps:

- Step 1: Identification of the data transmitted in the sending and receiving procedures at the implementation level.
- Step 2: Interpretation of the data that is transferred and comparison with the sequence diagram.
- Step 3: Identification and analysis of the cryptographic guards at the implementation level.

In step 1, the communication at the implementation level is examined and it is determined how the data that is sent and received can be identified in the source code. Afterwards, in step 2, a meaning is assigned to this data. The interpreted data elements of the individual messages are then compared with the appropriate elements in the model. In step 3, it is described how one can identify the guards from the model in the source code.
To this aim, it first needs to be identified at which points in the implementation messages are received and sent out, and which messages exactly. To be able to do this, we exploit the fact that in many implementations of crypto-protocols, message communication is implemented in a standardized way (which can be used to recognize where messages are sent and received). The common implementation of sending and receiving messages in cryptographic protocols is through message buffers, by writing the data into type-free streams (ordered byte sequences), which are sent across the communication link, and which can be read at the receiving end. The receiver is responsible for reading out the messages from the buffer in the correct order in storing it into variables of the appropriate types. This is done by using the methods write() from the class java.io.OutputStream to write the data to be sent into the buffer and the method read() from the class java.io.InputStream to read out the received data from the buffer. Also, the messages themselves are usually represented by message classes that offer write and read methods and in which the write and read methods from the java.io are called.
According to the information that is contained in a sequence diagram specification of a crypto-protocol, the runtime verification needs to keep track of the following information: 1. *Which data is sent out?* and 2. *Which data is received?* The runtime checks will enforce that the relevant part of the implementation conforms to the specification in the following sense. 1. *The code should only send out messages that are specified to be sent out according to the specification and in the correct order*, and 2. *these messages should only be sent out if the conditions that have to be checked first according to the specification are met*.
Some examples for such properties in the case of the SSL-protocol specified in Fig. 2 are given by the following requirements that arise from the above discussion:

(1) The client will not send out the ClientKeyExchange message until it has received the Certificate message from the server, has performed the validity check for the certificate as specified in Fig. 2, and this check turned out to be positive.
(2) The server will not send the Finished message to the client before the MD5 hash received from the client in the Finished message has been checked to be equal to the MD5 created by the server, and correspondingly for the SHA hash, but will send it out eventually after that has been established.
(3) The client will not send any transport data to the server before the MD5 hash received from the server in the Finished message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash.

Formalization in LTL Below, we explain how to capture to above properties using linear-time temporal logic (LTL) over the alphabet $\Sigma = 2^{AP}$ obtained by mapping a system's behaviour to atomic *actions* represented by the set AP of atomic propositions. A series of actions then corresponds to a string or word w of a formal language $L \subseteq \Sigma^{\infty}$, written $w \in L$. For each formula $\varphi \in$ LTL, we then use a standard construction to obtain a so-called *Büchi automaton* $\mathcal{A}^{\varphi}$, such that the accepted language of the automaton, $\mathcal{L}(\mathcal{A}^{\varphi})$, consists of all the models of φ, i. e. , $\mathcal{L}(\varphi)$. This automaton is then used to generate the actual Java code for the security monitor.
The first property can be formalized as $\varphi_1 = \neg \mathsf{ClientKeyExchange}_S \mathbf{U} \mathsf{Certificate}_R$ where $\{\mathsf{ClientKeyExchange}_S, \mathsf{Certificate}_R\} \subseteq AP$ is the set of atomic propositions that match the sending and receiving events of ClientKeyExchange and Certificate. The stream of events processed by our

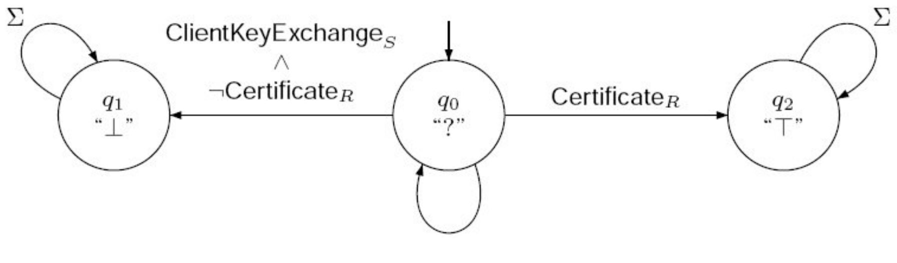


FIGURE 9: Finite state machine for first requirement

monitor consists of elements from 2^{AP}; that is, at each point in time, the application keeps track of both events the sending of ClientKeyExchange and the receiving of Certificate. If none of the events was observed, the according propositions are interpreted as $\perp$, otherwise as $\top$. Note that φ_1 is a classical co-safety property. The monitor for this property, which is obtained by the usual Büchi automaton generation from the LTL formular, is displayed as a finite state machine in Fig. 9. Each reachable state has an output symbol associated, stating whether φ is satisfied, violated, or neither. Labels on transitions indicate which symbols in the monitor's input trigger a transition. Σ on a loop means that any input triggers the corresponding transition. For instance, an observed trace $u = \emptyset\emptyset\ldots\{\text{Certificate}_R\}$ would lead to satisfaction of φ, whereas $u' = \{\text{ClientKeyExchange}_S\}$ would lead to violatation of φ.
Let us examine the second property as given above. It involves comparison of values and function calls. Since this cannot be modelled using LTL directly, we instead adapt the set of atomic propositions as follows. Let
$\{\text{Finished}_S, (MD5(\text{Finished}_R) = MD5(\text{Finished}_S)), (SHA(\text{Finished}_R) = SHA(\text{Finished}_S))\} \subseteq AP$.
In other words, we define two propositions that are interpreted as $\top$, if and only if the equality condition holds, which we have to check in the code of our application in terms of comparing the MD5 and SHA hash values. The corresponding property wrt. AP is then

$$\begin{aligned}\varphi_2 = &(\neg\text{Finished}_S\mathbf{W}(MD5(\text{Finished}_R) = MD5(\text{Finished}_S)))\\ &\wedge(\mathbf{F}(MD5(\text{Finished}_R) = MD5(\text{Finished}_S)) \Rightarrow \mathbf{F}\text{Finished}_S),\end{aligned}$$

and we assume some φ_2' where all occurrences of the proposition $(MD5(\text{Finished}_R) = \ldots)$ are replaced by the proposition $(SHA(\text{Finished}_R) = \ldots)$ from AP, respectively. Hence, we create two monitors for this security requirement: one for φ_2 and another one for φ_2'. Notice, φ_2 and φ_2' are neither strictly safety nor strictly co-safety properties. For instance, consider a trace $u = \emptyset\{\text{Finished}_S\}$, which violates the first part of our conjunction since $\text{Finished}_S = \top$, but $(MD5(\text{Finished}_R) = \ldots) = \perp$. On the other hand, the trace $v = \emptyset\{(MD5(\text{Finished}_R) = \ldots)\}$ is a model for φ, since our second observation in v shows that the MD5 checksum was successfully compared, and until then, $\text{Finished}_S = \perp$ held. Recall $\emptyset$ means that all propositions are interpreted as $\perp$. φ_2 is not co-safety since there exists the infinite model $v' = \emptyset\emptyset\ldots$ without a good prefix, i. e. , Finished_S never holds. Moreover, it is not safety since, there exists the infinite counterexample $u' = \{(MD5(\text{Finished}_R) = \ldots)\}\emptyset\emptyset\ldots$ without a bad prefix.
Requirement 3. can be formalised as $\varphi_3 = \neg Data\mathbf{W}((MD5(\text{Finished}_R) = MD5(\text{Finished}_S))$, where AP is as in the previous example, but additionally contains an action, indicating the sending of data, $Data$. Now we have a safety property since all traces of violating behaviour for φ_3 are finite, or in other words: there exist no infinite counterexamples that cannot be recognised with a bad prefix. It is not co-safety since the infinite trace $w = \emptyset\emptyset\ldots$ satisfies φ_3, which would be the case if an intruder has intercepted and kept the Finished message, such that it is never received at the server-side. Again, as in the previous example, we create a second monitor to check the outcome of the SHA-comparison.

Notice that monitoring strict safety properties, such as $\mathbf{G}p$, means that the corresponding monitor would output $?$ as long as no violation occurred, but never $\top$, since all models are infinite traces without good prefixes. However, φ_2 and φ_3 are such that they do have finite models, hence the corresponding monitor can output all three values of $\{\top, \perp, ?\}$, depending on the observed system behaviour. The same holds for φ_1, although it is strictly a co-safety property.

Code instrumentation & realisation Technically, our Java implementation of the SSL-protocol needs to set or unset the atomic propositions as system events are created, e.g. , by the sending and receiving of messages, or by the outcomes of comparing actual values with reference values, and so forth. Our monitors then process the resulting stream of actions, in that the setting or the unsetting of propositions creates a new action, each time this occurs. Once the monitors are generated for all relevant specifications, the only code that needs to be added to the main application is the code to set or unset propositions, and the code for handling communication between the monitors and the application itself.

4. CONCLUSIONS

We gave an overview on an approach for model-based security verification in which a design model in the UML security extension UMLsec can be formally verified against high-level security requirements such as secrecy and authenticity. An implementation of the specification can then be verified against the model by making use of run-time verification. The approach supports software evolution in so far as the traceability mapping used is updated when refactoring operations are regressively performed using our tool-supported refactoring technique. We applied our approach to an implementation of the Internet security protocol SSL. Although run-time verification is quite effective, sometimes it would be preferable to be able to statically verify at least a particularly critical part of the code, to further increase its trustworthiness. In future work we plan to investigate compositional software verification such as [1] to statically verify the relevant parts of a crypto-protocol verification in a localized way.

REFERENCES

[1] S. Abramsky, D. Ghica, A. Murawski, and C.-H. Ong. Applying game semantics to compositional software modeling and verification. In *TACAS*, pages 421–435, 2004.

[2] B. Best, J. Jürjens, and B. Nuseibeh. Model-based security engineering of distributed information systems using UMLsec. In *ICSE*, pages 581–590. ACM, 2007.

[3] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152. IEEE Computer Society, 2006.

[4] M. Calder. What use are formal design and analysis methods to telecommunications services? In K. Kimbler and W. Bouma, editors, *FIW*, pages 23–31. IOS Press, 1998.

[5] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, LNCS. Springer-Verlag, 2005.

[6] C. Hoare. How did software get so reliable without proof? In *Formal Methods Europe (FME'96)*, volume 1051 of *LNCS*, pages 1–17. Springer-Verlag, 1996.

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

[8] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *Int. Conf. on Software Engineering (ICSE)*, pages 322–331. IEEE/ACM, 2005.

[9] J. Jürjens, J. Schreck, and P. Bartmann. Model-based security analysis for mobile communications. In *30th Int. Conf. on Software Engineering (ICSE 2008)*. ACM, 2008.

[10] J. Jürjens and M. Yampolskiy. Code security analysis with assertions. In *20th Int. Conf. on Automated Software Engineering (ASE 2005)*, pages 392–395. ACM, 2005.

[11] J. Jürjens and Y. Yu. Tools for model-based security engineering: Models vs. code. In *Automated Software Engineering (ASE)*. IEEE/ACM, 2007.

[12] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *CCS*, pages 260–269, 2005.

[13] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[14] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, Reading, MA, 2001.

[15] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[16] M. Thomas. Engineering judgement. In A. Cant, editor, *SCS*, volume 47 of *CRPIT*, pages 43–47. Australian Computer Society, 2004.

[17] Security analysis tools, 2001-08. http://computing-research.open.ac.uk/jj/sectracetool.

[18] J. Woodcock, S. Stepney, D. Cooper, J. Clark, and J. Jacob. The certification of the Mondex electronic purse to ITSEC Level E6. *Formal Asp. Comput.*, 20(1):5–19, 2008.