Run-Time Security Traceability for Evolving Systems[1]

ANDREAS BAUER[1,2], JAN JÜRJENS[3], YIJUN YU[4]

[1]*National ICT Australia (NICTA)*
[2]*School of Computer Science, The Australian National University*
[3]*TU Dortmund and Fraunhofer ISST (Germany)*
[4]*Computing Department, The Open University, UK*
Email: baueran@rsise.anu.edu.au, j.jurjens@cs.tu-dortmund.de, y.yu@open.ac.uk

Security-critical systems are challenging to design and implement correctly and securely. A lot of vulnerabilities have been found in current software systems both at the specification and the implementation levels. This paper presents a comprehensive approach for model-based security assurance. Initially, it allows one to formally verify the design models against high-level security requirements such as secrecy and authentication on the specification level, and helps to ensure that their implementation adheres to these properties, if they express a system's run-time behaviour. As such, it provides a traceability link from the design model to its implementation by which the actual system can then be verified against the model while it executes. This part of our approach relies on a technique also known as run-time verification. The extra effort for it is small as most of the computation is automated; however, additional resources at run-time may be required. If during run-time verification a security weakness is uncovered, it can be removed using aspect-oriented security hardening transformations. Therefore, this approach also supports the evolution of software since the traceability mapping is updated when refactoring operations are regressively performed using our tool-supported refactoring technique. The proposed method has been applied to the Java-based implementation JESSIE of the Internet security protocol SSL, in which a security weakness was detected and fixed using our approach. We also explain how the traceability link can be transformed to the official implementation of the Java Secure Sockets Extension (JSSE) that was recently made open source by Sun.

Keywords: run-time verification, monitoring, IT security, cryptographic protocols, formal verification, security analysis, software evolution, requirements traceability

Received 08 November 2008; revised 31 March 2009 and 12 September 2009

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 authentication), 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 for Model-based Security Engineering (MBSE) that has been developed in recent years (see e.g., [1, 2] for details and Figure 1 for a visual overview). In this approach, recurring security requirements (such as secrecy, integrity, authentication and others) and security assumptions on the system environment, can be specified either within a UML specification (using the UML extension UMLsec [1]), or within the source code (Java or C) as annotations. One can then formally analyse the UMLsec models against the security requirements using the UMLsec tool suite which makes use of model checkers and automated theorem provers for first-order logic (see Figure 2 and [3, 4]). The approach has been used successfully in a number of industrial applications (e.g., at BMW [5] and O_2 (Germany) [6]).

However, it is not enough that the specification is secure: we must also ensure that the implemented system is secure as well. There are at least two ways to approach this problem: static code verification, or a technique called run-time verification [7, 8, 9] (see Section 3.1 for an introduction). In this paper, we focus on using (online) run-time verification for our purposes. It has an important advantage over static verification: In static verification, one can only verify the implementation on the basis of predefined assumptions. For example, these include assumptions on the behavioural semantics of the programming language, the compiler that will compile the source code to byte code, and/or byte code to machine code, the execution environment (operating system, hardware, physical environment), etc. When trying to apply static verification to complex implementations, one usually needs to make additional simplifying abstractions in order to make an automated formal verification of such implementations feasible in the first place, and one thus needs to make the additional assumption that these abstractions do not limit the scope of the verification. The verification procedure is then only known to be sound where these assumptions are fulfilled, and it is usually not feasible to verify formally whether they are fulfilled for a given execution environment. The advantage of run-time verification is now that the targeted execution environment itself is part of the verification environment (since verification is done at run-time anyway), so by construction the verification will be sound for the execution environment at hand. For this reason, run-time verification also does not suffer from the same scalability issues that static verification does as systems or system models become more complex: run-time verification always considers one concrete behaviour produced by the running system rather than the overall state-space of all the possible states it can be in.

Since run-time verification is a formal yet also *dynamic* technique (i.e., it operates on the running system as compared to a system model) there exist, besides similarities to other formal verification techniques such as model checking, some similarities to testing; however, the context and goals are different: Testing for complex implementations can usually not be applied exhaustively. In contrast to that, run-time verification ensures, by construction, that every system trace that will ever be executed will be verified—while it is executed. In the case of the cryptographic protocols that we consider, it is indeed sufficient to notice attempted security violations at run-time to still be able to maintain the security of the system: The monitor is constructed in such a way that, if it detects a violation, the current execution of the security protocol will be terminated before any secret information is leaked out on the network. Therefore, despite the similarities between testing and run-time verification, run-time verification can provide a level of assurance that goes beyond what testing can usually achieve when applied to highly complex security-critical software.

In practice, however, systems do not remain the same after they are deployed. On the contrary, many systems evolve over their life-time, and usually their life-time is significantly longer than expected when they were implemented (this became very apparent with the year 2000 bug). Manually re-establishing the verified traceability link for a new version of an implementation would be time-intensive. It would therefore be preferable if we enable our security assurance approach to cope automatically with the fact that systems will evolve at run-time, and still provide valid run-time security assurance. This is non-trivial to achieve: As the implementation or the used libraries evolve, the instrumentation may no longer guarantee the correct link to the protocol design. It is therefore important to have a way to perform refactoring steps in a traceable way.

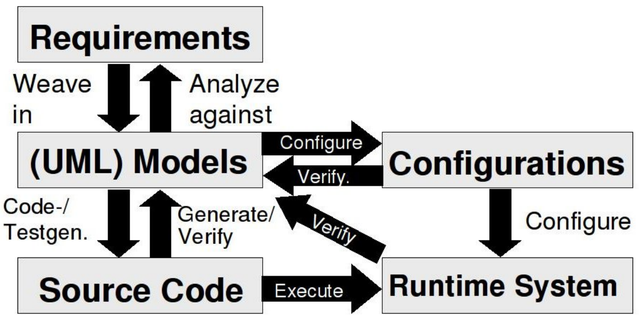

FIGURE 1. Model-based Security Engineering

In addition, we explain how to achieve security hardening through systematic instrumentations. As security vulnerabilities are often scattered throughout the implementation, we choose aspect-oriented programming (AOP) for security hardening.

One goal of our work is thus to maintain traceability between the design and the implementation of a crypto-based software through a dedicated software refactoring approach which supports system evolution.

Note that an alternative approach could aim to generate complete implementations out of cryptographic protocol specifications, rather than establishing a link between the specification and an existing implementation, and hardening that implementation if necessary. If that would be possible, that would automatically also update the link between the specification and the implementation whenever the specification is changed, by just generating a new implementation. However, this is not our goal here. Rather, we would like the approach we develop here to be applicable to existing legacy implementations, rather than generating new implementations. The reason for this is that, in practice, there is often a strong desire to use a particular existing implementation. For example, that implementation might be conformant with certain standards or certifications, or satisfy stringent performance requirements (which an implementation automatically generated from a specification would usually not be able to satisfy). Since legacy implementations are usually too complex to verify statically, this again motivates the use of run-time verification and our approach.

One should note that our approach, at this point, focusses on a certain class of attacks which can be detected when observing the running implementation at a certain degree of abstraction, namely those attacks that rely on an interaction of the attacker with the protocol participants where the passive or active (man-in-the-middle) attacker can read, memorise, insert, change, and delete message parts into the communication between the protocol participants. In each case, we assume the actual cryptographic algorithms and their implementations (such as encryption and digital signature) to be secure, and we aim to detect insecurities in the way they are used in the context of a cryptographic protocol. We do not aim to detect attacks that rely on breaking these assumptions, such as statistical attacks or type confusion attacks.

1.1. A Brief Overview of the Approach

Let us briefly summarise the approach taken in this paper. Our approach supports the following steps:

(1) security protocols are specified and verified using the security extension UMLsec of UML,

(2) an implementation is linked to this UML model,

(3) temporal logic formulae are derived from the UML model,

(4) a security monitor is generated automatically from the temporal logic formulae created in step (3) in order to verify the implementation at run-time,

(5) the relation between the UML model and the code is maintained as the implementation evolves over time,

(6) errors in the implementation can be corrected using AOP, and

(7) the security monitor is updated with respect to the changes arising from step (6).

There are practical considerations why such a process is not fully automated, but rather has to be semi-automated; that is, some manual work is required and may be desired to have full control over the system as it evolves over time. However, steps (4) to (7) can be fully automated and are thus repeatable, given that the specifications from (1) to (3) are established manually. The time and effort spent on steps (1) to (3) can be considered as an overhead to the normal software development process, while steps (4) to (7) save the effort to accommodate changes in the evolving system. Moreover, it helps us in maintaining traceability links as this happens.

Note that our approach is interesting to apply not only to legacy systems (where there is often no alternative to manually re-engineering a specification, and static verification of the software is often not an option because of its complexity). It is also useful to apply our approach in a situation where model-based development techniques are used to develop a system: Experiences from practice indicate that, in such a context, changes are often done on the code level after the development of the model has been finished, which often means that the code becomes inconsistent with the model, which makes it necessary to monitor the code at run-time.

Our approach thus supports verified traceability that is robust under evolution at various stages of the system life-cycle:

- Verified traceability from security requirements to design: one includes security requirements as annotations into UML models and automatically verifies the models against these requirements (see Section 2).
- Verified traceability of security requirements from design to execution time: using run-time verification (see Section 3).
- Verified traceability of security requirements from one version of the implementation to another through system evolution (see Section 4).
- Traceable security hardening for code-level security vulnerabilities (see Section 5).

1.2. Advance over Prior State of the Art

In this section, we explain in which respect the work presented in this paper constitutes an advance over the prior state of the art in this field.

New methodology: We have developed a new integrated methodology for run-time security verification of cryptographic protocol implementations that can handle system evolution.

Prior to our work there existed, to the extent of our knowledge, no approach to security run-time verification of cryptographic protocols, and even less an approach that would be able to handle evolution. There exist other approach for run-time verification of security properties but to the extent of our knowledge they have not been applied to implementations of cryptographic protocols, which pose particular challenges for run-time verification that have thus not been addressed by other approaches.

In particular, we developed a new approach for model-based security assurance that covers properties from the design level all the way down to the implementations.

We validated the new methodology by applying it to several versions of two industrial size applications, the Java Secure Sockets Extension (JSSE) and the open-source implementation JESSIE. To keep the paper readable, we cannot give complete accounts of these whole applications, but can only focus on several examples taken from them. Nevertheless, the size and complexity of the overall application allows us to draw significant conclusions about the applicability of the methodology.

Advance over prior work: Some but not all parts of the methodology build on prior work, although that prior work needed to be further developed significantly in order to be applicable to run-time security monitoring of cryptographic protocol implementations that can handle system evolution, since neither of the prior work was able to deal with this task on the whole.

The work presented in this paper thus constitutes an advance over prior work in the following directions:

- model-based security analysis
- run-time verification
- software evolution

We will shortly discuss the advance over prior work in these directions.

Model-based security analysis The work presented here exceeds the prior work in the area of model-based security analysis in so far that we have developed an approach which allows one to combine automated static verification on the model level (using UML models as a specification approach together with a formalization in first-order logic and the use of automated theorem provers for first-order logic) with run-time verification for monitoring the assumptions on which the static verification is based at run-time.

Run-time verification With regards to run-time verification, because of the particular challenges involved with run-time verification of cryptographic protocols (as opposed to other security-critical software), we base our work on a specialised approach of monitoring temporal logic formulae using 3-valued semantics, whose theoretical foundations were first presented in [10], however, without a detailed evaluation or application to a case study. Our discussion in this paper of which properties are monitorable by this particular run-time verification approach will demonstrate that this technique has significant advantages when applied to run-time verification of cryptographic protocol implementation. The approach has been implemented in terms of [11] by the first author of this paper. The implementation, which we also use in this paper, is available as open source software via a SourceForge web site.

Software evolution The work presented here exceeds the prior work with respect to software evolution in so far that it supports a combined static and run-time security verification approach in the context of software evolution. The work addresses in particular the challenge that an accurate design to implementation traceability is required to make the approach applicable in the context of software evolution. Although refactoring and aspect-oriented programming are two well-known subjects and are well supported by the integrated development environments, this work is to the best of our knowledge the first to use refactoring to create and maintain traceability for secure software development that enables the use of AOP techniques to fix security weaknesses related traceability failures. We also show that our approach improves the precision of the traceability.

An additional contribution of this work is then also to show how these techniques can be combined in an efficient manner, and how they can be used in the context of developing security critical systems.

In this paper, we focus specifically on cryptographic protocols, since these are a compact yet highly security-critical and non-trivial to design part of a secure system, and thus serve as a particularly good example to demonstrate our approach.

Significant practical applications We have applied this new methodology in a significant new application to the SSL protocol implementations JESSIE and JSSE, which are industrial strength implementation with a large user base (particularly in the case of JSSE which is part of the standard Java security architecture).

More precisely, we demonstrate the approach by an application to the Java-based implementation JESSIE of the Internet security protocol SSL. We also explain how the traceability link can be transformed to the official implementation of the Java Secure Sockets Extension (JSSE) that was recently made open source by Sun.

Again, run-time verification of widely used crypto-protocol implementations such as JESSIE and JSSE have to the extent of our knowledge not been attempted so far, and they pose particular challenges since these implementations are significantly complex. In particular, this application allowed us to detect a previously unknown security vulnerability in one of the implementations, which was then hardened using our approach.

In addition to being applications that are interesting on their own, these industrial-size applications also allowed us to validate the new methodology proposed in this paper. In particular, the size and complexity of the overall application allows us to draw significant conclusions about the applicability of the methodology.

Comparison to previous work The work presented in this paper is new: although there has been a lot of work on formally verifying abstract specifications of cryptographic protocols, the only prior work on run-time verification for cryptographic protocols is (to our knowledge) the precursory conference paper [12], which however did not include support for automated security hardening, and for maintaining the verification results when the system evolves.

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 (based on UML models) and apply them to an industrially used programming language (Java). We hope to thus contribute to dealing with the challenges faced when trying to use formal methods in a practical environment (cf. [13, 14, 15] 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 [1] 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 [16, 17]. 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 cryptographic protocols developed recently, such as [18, 19, 20]. 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 motivation for this

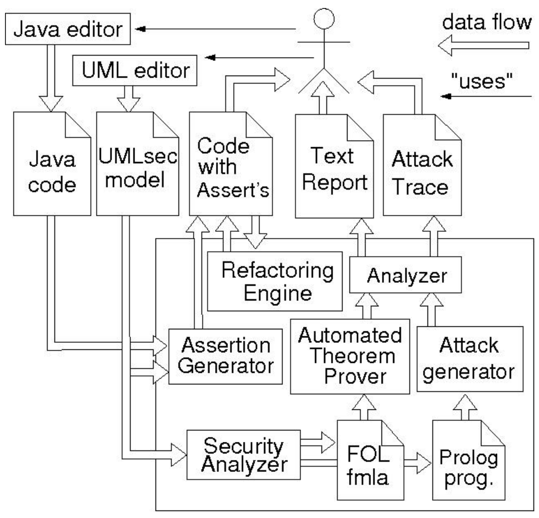

FIGURE 2. MBSE tool framework

two-step verification process is to facilitate application to complex legacy software.

See Section 6 for a more detailed comparison to previous research.

1.3. Outline of the Rest of the Paper

The rest of this paper proceeds as follows. In Section 2, we give an overview of model-based security engineering as a means of analysing *models* of security-critical systems in the UMLsec specification notation at design-time using first-order logic (FOL) theorem proving. We also explain how this technique was applied to the SSL protocol. Then, in Section 3, we discuss run-time verification as a dynamic verification technique in more detail, how to obtain run-time security properties of a system, and apply this approach to our case study, a Java implementation of the SSL protocol, JESSIE. As such we discuss the link between model and code of a system. In Section 4, we give a detailed account on how to maintain the links between models and code, in the face of system evolution, e.g., occurring program changes due to fixing bugs, or extending the functionality of a program. That is, we establish mappings between elements of our system models and code, and use automated refactoring techniques for changing an implementation. Finally, in Section 5, we outline an AOP-based approach that allows us to react to security weaknesses detected in an implementation by *security hardening*.

Related work is discussed Section 6 and we draw conclusions from our work in Section 7.

2. MODEL-BASED SECURITY ANALYSIS

In this section, we give an overview of the part of our approach that applies to the specification level of a cryptographic protocol. We start by giving a general overview of the approach we use there (called model-based security engineering), then explain the relevant part of that approach in technical detail, and finally apply it to our running example, the SSL protocol.

2.1. Model-based Security Engineering

Model-based Security Engineering [1, 2, 21] provides a soundly based approach for developing security-critical software where recurring security requirements (such as secrecy, integrity, authentication 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. Figure 1). Various analysis plug-ins in the associated UMLsec tool framework [22, 4] (Figure 2) generate logical formulae formalising the execution semantics and the annotated security requirements. Automated theorem provers and model checkers are used to try to automatically establish whether the security requirements hold. (Note that security requirements in general are undecidable, so there may be worst-case examples which cannot be decided automatically, although in our experience most practical applications are unproblematic.) 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.

Note that some of the activities contained in Figure 1 are done manually or supported with pre-existing tools outside the UMLsec tool suite, and therefore the relevant workflows do not appear in Figure 2. For example, to generate Java code from UML models (or vice versa) one can use the commercial tool suite Borland Together [23].

Part of the Model-based Security Engineering (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 [24, 25, 1]. 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».

$\mathsf{enc}_{E'}(E)$	(encryption)
$\mathsf{dec}_{E'}(E)$	(decryption)
hash(E)	(hashing)
$\mathsf{sign}_{E'}(E)$	(signing)
$\mathsf{ver}_{E'}(E,E'')$	(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 3. Abstract Cryptographic Operations

In each case, the assumptions, requirements, and policies are defined formally and precisely in [1] on the basis of a formal semantics for the used fragment of UML. We do not repeat these definitions here, since in this paper we will look at one specific security analysis scenario, where the assumptions and requirements are defined precisely at the level of the used formalisation in first-order logic.

The UMLsec tool-support (illustrated in Figure 2) 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 [4, 2]. 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 [1] 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, authentication, and secure information flow are defined. To support stepwise development, one can show secrecy, integrity, authentication, and secure information flow to be *preserved* under refinement and the composition of system components. The approach also supports the secure development of layered security services (such as layered security protocols). See [1] for more information on the above.

2.2. Analysing Cryptographic Protocols

In the current paper, we concentrate on applying model-based security engineering to the special case of cryptographic protocols which are a particularly interesting target since they are compact pieces of highly security-critical software which are nevertheless highly non-trivial to design and implement correctly.

Using UML sequence diagrams, each message in a cryptographic protocol is specified by giving the sender, the receiver, the message, and possibly a precondition (in equational first-order logic (FOL)) which has to be fulfilled so that the message is sent out.

As usual in the formal analysis of cryptographic software, the cryptographic algorithms (such as encryption and decryption) are viewed as abstract functions. Our aim in this paper is not to verify the implementation of these algorithms, but we work on the basis of the assumption that these are correct, and aim to verify whether they are used correctly within the cryptographic protocol implementation.

We assume a set **Keys** of encryption keys disjointly partitioned in sets of *symmetric* and *asymmetric* keys. We fix a set **Var** of *variables* and a set **Data** of *data values* (which may include *nonces* and other secrets). The *algebra of expressions* **Exp** is the term algebra generated from the set $\mathbf{Var} \cup \mathbf{Keys} \cup \mathbf{Data}$ with the operations given in Figure 3. There, the symbols E, E', and E'' denote terms inductively constructed in this way. Note that encryption $\mathsf{enc}_{E'}(E)$ is often written more shortly as $\{E\}_{E'}$, and that we sometimes use a specific notation $\mathsf{symenc}_{E'}(E)$ for symmetric encryption (although these alternative notations are both "syntactic sugar" without impact on the formalisation). In this term algebra, we impose the following equations, formalising the fact that decrypting with the correct key gives back the initial plain-text, and similarly for verification of signatures: $\mathsf{dec}_{K^{-1}}(\mathsf{enc}_K(E)) = E$ (for all $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$) and $\mathsf{ver}_{E'}(E,E'') = \mathsf{true}$ (for all $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$). We also assume the usual laws regarding concatenation, **head**(), and **tail**(), and that $K = K^{-1}$ for any symmetric encryption key K.

A cryptographic protocol can then be verified for the relevant security requirement such as secrecy and authentication using the UMLsec tools presented above, which rely on a translation from the UMLsec sequence diagram to a security-sensitive interpretation in FOL-based on the Dolev-Yao attacker model as explained in [2], which is then verified using automated theorem provers for FOL. The idea here is that an adversary can read messages sent over the network and collect them in his knowledge set. The adversary can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements can then be formalised 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 analyse the UMLsec specification by making use of our translation from cryptographic protocols specified as UML sequence diagrams to FOL formulae which can be processed by the automated theorem prover e-SETHEO [26]. The formalisation 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

$$\begin{aligned}&\forall E_1, E_2.\big(\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathsf{sign}_{E_2}(E_1))\big)\\ &\quad \wedge\big(\mathsf{knows}(E_1 :: E_2) \Rightarrow \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big) \wedge \big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \Rightarrow \mathsf{knows}(E_1)\big)\\ &\quad \wedge\big(\mathsf{knows}(\mathsf{sign}_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1)\big)\end{aligned}$$

FIGURE 4. FOL rules for attacker knowledge generation

$$\begin{aligned}\mathsf{PRED}(l) = \quad & \forall exp_1,\dots,exp_n.\big(\mathsf{knows}(exp_1) \wedge \dots \wedge \mathsf{knows}(exp_n) \wedge\ cond(exp_1,\dots,exp_n)\\ & \qquad\qquad \Rightarrow \mathsf{knows}(exp(exp_1,\dots,exp_n) \wedge\ \mathsf{PRED}(l'))\big)\end{aligned}$$

FIGURE 5. FOL rule for attacker interaction

following therefore has the goal to verify the UML sequence diagram against 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 formalisation will thus compute all scenarios which would lead the attacker to derive $\mathsf{knows}(s)$.

The FOL rules generated for a given UMLsec specification are 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 Figure 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 (where e is a variable of the type **Exp** defined above and e' is a term which evaluates to a value of type **Exp**). Connections are the arrows from the life line of a source object to the life line of a target object which are labelled 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 $\mathsf{guard}(l) \equiv cond(arg_1,\dots,arg_n)$, and $\mathsf{msg}(l) \equiv exp(arg_1,\dots,arg_n)$, 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 Figure 5. If such a connection l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Figure 5.

The formula formalises the fact that, if the adversary knows expressions $exp_1,\dots,exp_n$ validating the condition $cond(exp_1,\dots,exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1,\dots,exp_n)$ 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, this gives a predicate $\mathsf{PRED}(O) = \mathsf{PRED}(l)$ where l is the first connection in the sequence diagram with $\mathsf{source}(l) = O$. The axioms in the overall FOL formula for a given sequence diagram are then the conjunction of the formulae representing the publicly known expressions, the formula in Figure 4, and the conjunction of the formulae $\mathsf{PRED}(O)$ for each object O in the diagram. The conjecture, for which the automated theorem prover 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 $\mathsf{knows}(s)$. An example is given in the next section.

2.3. Application to SSL

We have applied the approach to the core part of the SSL 3.0 handshake protocol given in Figure 6 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 and is therefore an interesting target for a security analysis. It may be interesting to note that early versions of SSL (before becoming a “standard” renamed as TLS in RFC 2246) had been the source of several significant security vulnerabilities in the past [27]. In order to simplify the exposition, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (there is no specific reason why we chose this particular fragment, and we concentrate on a fragment just to simplify the explanations). The protocol participants (here the instances C of class Client and S of class Server) are rep-

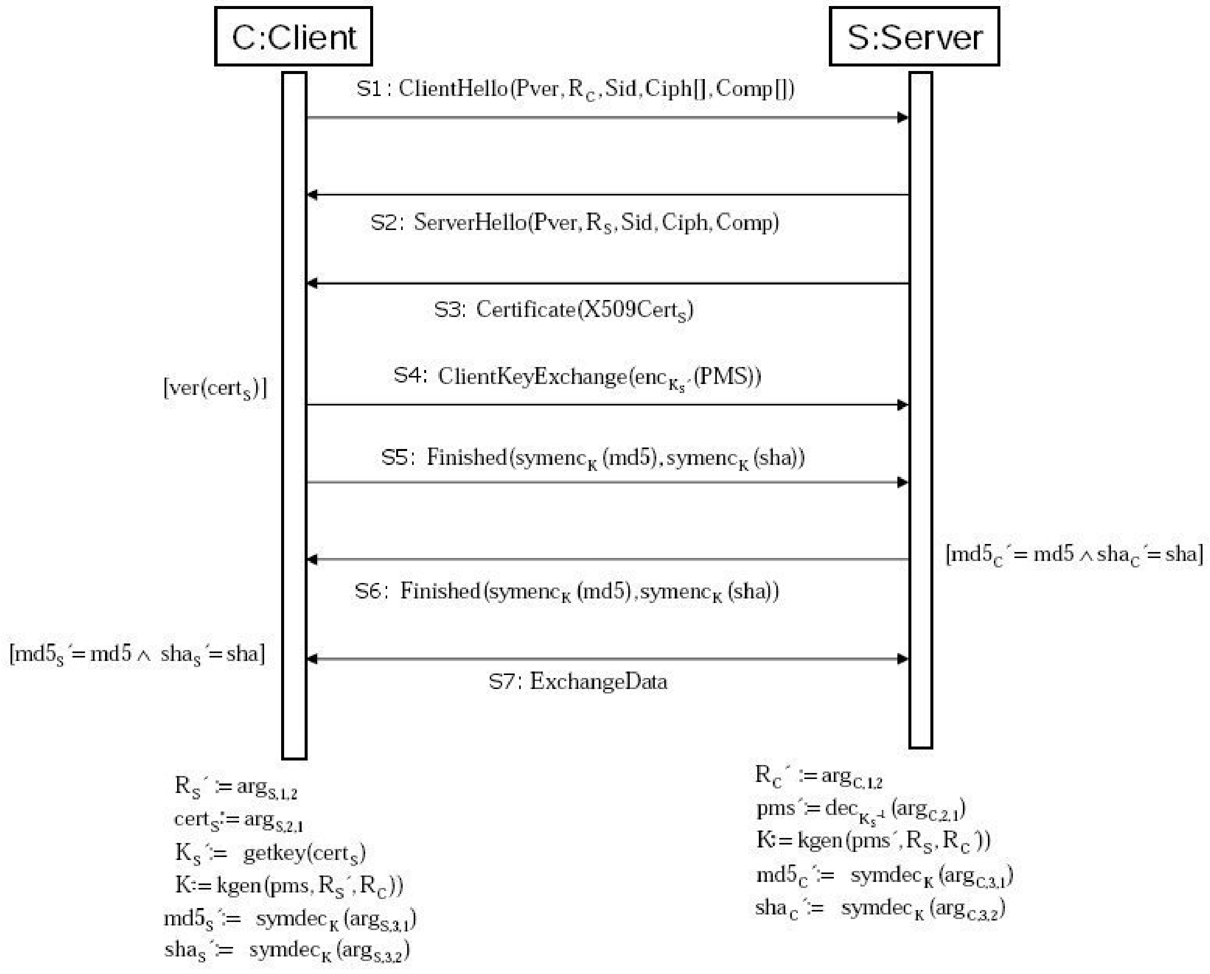
C:Client
S:Server
S1 : ClientHello(Pver, R_C, Sid, Ciph[], Comp[])
S2: ServerHello(Pver, R_S, Sid, Ciph, Comp)
S3: Certificate(X509Cert_S)
[ver(cert_S)]
S4: ClientKeyExchange(enc_{K_S}'(PMS))
S5: Finished(symenc_K(md5), symenc_K(sha))
[md5_C' = md5 ∧ sha_C' = sha]
S6: Finished(symenc_K(md5), symenc_K(sha))
[md5_S' = md5 ∧ 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 6. Handshake protocol of SSL3 using RSA and Server Authentication

resented 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 Figure 6 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 *p*th element of the *n*th 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 [md5$_S$' = md5 $\wedge$ sha$_S$' = sha] and [md5$_C$' = md5 $\wedge$ sha$_C$' = 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 premaster secret PMS. The values md5 and sha used as message arguments are created by the sender of the respective message by using the MD5 respectively 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.

We used the UMLsec tools to verify the UMLsec model of the SSL protocol (cf. Figure 6) against relevant security requirements such as secrecy. Verifying secrecy of a value *s* can be done by checking whether the statement knows(s) is derivable from the FOL formulae generated from the protocol specification. 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 now explain how to link the formally verified specification to a crypto-based implementation which may not be trustworthy (for example, it might have been implemented insecurely from a secure specification, either maliciously or accidentally), in a way that enforces the security of the running system. That is, we use (online) run-time verification (cf. [7, 8, 9]; see also Section 3.1 for a detailed overview of this technique) to check whether or not the implementation conforms to our formal security properties while it executes. We currently focus on Java as the implementation language.

3.1. Run-time Verification using LTL

In a nutshell, run-time verification is a *formal* but *dynamic* technique to establish whether or not an executing system adheres to a predefined property (or a set thereof), by monitoring whether the system satisfies the property while it is used. Properties are typically specified in a temporal logic, such as LTL [28], and the object under scrutiny is the actual system and not its representation in terms of an abstract model or code as is the case with a static technique.

As such, run-time verification bears not only strong resemblance to testing since both techniques are dynamic, as already pointed out in the introduction, but also to rigorous formal verification methods such as (LTL-) model checking (cf. [29]), for instance. The idea of (LTL-) model checking is roughly as follows. A model of the system under scrutiny is checked against a formal correctness property, usually specified in terms of a temporal logic such as LTL, by verifying that all possible executions specified by the model adhere to the specified behaviour by the property. However, depending on the temporal logic used, the complexities of model checking range from polynomial in the size of the system model and property to PSpace-complete in the formula as is the case for LTL, which we are concerned with in this paper. While model checking has been successfully employed, e.g., for checking models of protocols (cf. [30]), using it to verify software in terms of low-level source code abstractions is still an active research subject due to the large state-spaces that result from using low-level models extracted from source code as compared to high-level behavioural models such as sequence diagrams or state machines (cf. [31, 32]). The advantage of model checking, however, is that once correctness has been established, we can be sure that the specified behaviour does, indeed, adhere to the intended behaviour—all possible executions have been checked. If model checking a system fails, then usually the model checker returns a counterexample in terms of a system execution that leads to the violation of the temporal logic property. In such a case, the system can be repaired with respect to the counterexample and perhaps model checked again.

Run-time verification (cf. [7, 8]) is similar to the above in the sense that it also employs, usually, a formal correctness property, specified in temporal logic, to capture either intended behaviour (i.e., when one is interested to detect occurrence of a certain "good" behaviour), or unwanted behaviour (i.e., when one is interested in detecting when something "bad" has happened). However, model checking is a static verification technique as it operates on the model-level, whereas run-time verification operates directly on the system implementation. Moreover from a formal point of view, let $\mathcal{L}(M)$ be the language generated by some system model and $\mathcal{L}(\varphi)$ be the language of some formal property, φ. Then, model checking translates to checking whether or not the formal language generated by the system is *contained* in the formal language generated by the property, i.e., whether $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$ holds. Run-time verification, on the other hand, asks for the answer of a *word problem*: Let u be the prefix of some potentially infinite word w, which resembles the system's behaviour, then we want to know whether or not $w \in \mathcal{L}(\varphi)$ after reading u. Or, in other words, we want to know, after seeing the finite sequence of behaviour u, whether or not for all possible extensions of u, our property will be satisfied, violated, or neither. Note that the last case simply means we have to wait for more behavioural observations until we can give a conclusive answer to this question. (For a more formal account on this form of run-time verification, see Sections 3.1.2 and 3.1.4.).

From a methodological point of view, in run-time verification, a so-called *monitor*, whose task it is to observe the system behaviour as it executes, is automatically generated from a security property (or a set thereof) formalised as an LTL formula, also referred to as an LTL property. This process is somewhat similar to constructing finite automata from regular expressions [33], which are also a formal means to define sequences of actions, i.e., system behaviour. If a monitor detects a violation of the security property it raises an alarm, if it detects that a security property was fulfilled it signals accordance, and otherwise keeps monitoring the executing system. Unlike regular expressions, temporal logic and, in particular, LTL-based temporal logic, has established itself in the area of formal verification and is nowadays frequently used also in industry to define the behaviour of systems (cf. [34]).

3.1.1. Definitions and Notation

In what follows, we briefly recall some formal definitions regarding LTL and introduce the necessary notation. First, let AP be a non-empty set of *atomic propositions*, and $\Sigma := 2^{AP}$ be an *alphabet*. Then infinite words over Σ are elements from Σ^ω and are abbreviated usually as $w, w', \ldots$. Finite words over Σ are elements from Σ^* and are usually abbreviated as $u, u', \ldots$. The notion of infinite words makes sense when we consider the system under scrutiny being a reactive system, where the assumption is that the system is never switched off, and the words as a means to model the observable behaviour of that system. In run-time verification, however, we always observe only the prefix of a potentially infinite behaviour, hence we need a reasonable interpretation for LTL formulae over finite words as well. More specifically, our monitors adhere to the semantics introduced in [10] and realised by the open source monitor generator in [11]. It is explained also in Section 3.1.4.

We will adopt the following terminology with respect to monitoring LTL formulae. We will use the propositions in

AP to represent atomic system *actions*, which is what will be directly observed by the monitors introduced further below. As an example, an action may correspond to a specific function call, or a specific message that is sent or received by a participant in a protocol. This depends somewhat on the property being monitored, and the application at hand. A more comprehensive example is discussed in Section 3.3. Note also that, by making use of dedicated actions that notify the monitor of changes in the system state, one can also indirectly use them to monitor whether properties of the system state hold. Thus, we can use the terms "action occurring" and "proposition holding" synonymously. We will refer to a set of actions as an *event*, denoting the fact that certain actions may have occurred simultaneously, or that a certain state holds, described by a set of actions.

3.1.2. LTL Syntax and Semantics

The set of LTL formulae over Σ, written LTL(Σ), is inductively defined by the following grammar:

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, \quad p \in AP.$$

The *semantics* of LTL formulae is defined inductively over its syntax as follows. Let $\varphi, \varphi_1, \varphi_2 \in$ LTL(Σ) be LTL formulae, $p \in AP$ an atomic proposition, $w \in \Sigma^\omega$ an infinite word, and $i \in \mathbb{N}$ a position in w. Let $w(i)$ denote the ith element in w (which is a set of propositions).

The (infinite word) *semantics* of LTL formulae is then defined inductively by the following logical statements.

$$\begin{array}{lll} w,i \models true & & \\ w,i \models \neg\varphi & \Leftrightarrow & w,i \not\models \varphi \\ w,i \models p & \Leftrightarrow & p \in w(i) \\ w,i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w,i \models \varphi_1 \vee w,i \models \varphi_2 \\ w,i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w,k \models \varphi_2 \wedge \\ & & \quad \forall i \leq l < k.\ w,l \models \varphi_1 \\ & & \quad (\text{``}\varphi_1 \text{ until } \varphi_2\text{''}) \\ w,i \models \mathbf{X}\varphi & \Leftrightarrow & w,i+1 \models \varphi \quad (\text{``next } \varphi\text{''}) \end{array}$$

Here w,i denotes the ith position of w. We also write $w \models \varphi$, if and only if $w,0 \models \varphi$, and use $w(i)$ to denote the ith element in w which is a set of propositions, i.e., an event. (Notice the difference between w,i and $w(i)$.)

Intuitively, the statement $w,i \models \varphi$ is supposed to formalise the situation that the event sequence w satisfies the formula φ at the point when the first i events in the event sequence w have happened. In particular, defining $w,i \models true$ for all w and i means that *true* holds at any point of any sequence of events.

Further, as is common, we use $\mathbf{F}\varphi$ as short notation for $true\mathbf{U}\varphi$ (intuitively interpreted as "eventually φ"), $\mathbf{G}\varphi$ short for $\neg\mathbf{F}\neg\varphi$ ("always φ"), and $\varphi_1\mathbf{W}\varphi_2$ short for $\mathbf{G}\varphi_1 \vee (\varphi_1\mathbf{U}\varphi_2)$, which is thus a weaker version of the **U**-operator. For brevity, whenever Σ is clear from the context or whenever a concrete alphabet is of no importance, we will use LTL instead of LTL(Σ).

3.1.3. Examples

We give some examples of LTL specifications. Let $p \in AP$ be an action (formally represented as a proposition). Then $\mathbf{GF}p$ asserts that at each point of the execution of any of the event sequences produced by the system, p will afterwards eventually occur. In particular, it will occur infinitely often in any infinite system run.

For another example, let $\varphi_1, \varphi_2 \in$ LTL be formulae. Then the formula $\varphi_1\mathbf{U}\varphi_2$ states that φ_1 holds until φ_2 holds and, moreover, that φ_2 will eventually hold. On the other hand, $\mathbf{G}p$ asserts that the proposition p always holds on a given trace (or, depending on the interpretation of this formula, that the corresponding action occurs at each system update).

3.1.4. Finite-Word Monitor Semantics

To see how our monitors cope with the situation that at run-time only prefixes of potentially infinite words are observable, we also outline the semantics employed by the monitors, which is slightly different from the above LTL semantics, but based on it. Notably, it is a 3-valued semantics and defined as follows. Let $\varphi \in$ LTL, and $u \in \Sigma^*$. Then, a monitor for φ returns the following values for a processed u, written $[u \models \varphi]$:

$$[u \models \varphi] := \begin{cases} \top, & \text{if for all } v \in \Sigma^\omega \text{ we have } uv \models \varphi \\ \bot, & \text{if for all } v \in \Sigma^\omega \text{ we have } uv \not\models \varphi \\ ?, & \text{otherwise.} \end{cases}$$

The $[\cdot]$ is used to separate the 3-valued monitor semantics for φ from the classical, 2-valued LTL semantics introduced above.

In other words, this definition says that a monitor which was generated for a formula φ will, upon reading some prefix u, return $\top$ if for all possible extensions of u the infinite word semantics is fulfilled (i.e., $uv \models \varphi$), and $\bot$ if for all possible extensions of u the infinite word semantics is violated (i.e., $uv \not\models \varphi$). Moreover, if there exists an extension v' to u such that $uv' \in \varphi$, and there exists another extension v'' such that $uv'' \notin \varphi$, then the monitor returns ? and keeps monitoring until u is long enough to allow for a conclusive answer (i.e., $\top$ or $\bot$).

Such conclusive prefixes are also referred to as good (respectively bad) prefixes (cf. [8]) with respect to the monitored language that is given by φ. From that point of view, we can say that a monitor detects good (respectively bad) prefixes for the monitored property. Note, however, that not all properties that can be formalised in LTL necessarily have such a good or a bad prefix. Therefore, monitoring is often restricted to so called safety properties, where violations can be detected via bad prefixes (cf. [35]). In contrast, our monitoring procedure is not restricted to safety properties alone, but also to properties that lie outside this language-theoretic categorisation. For a more detailed comparison between the approach discussed in [35] and our monitoring framework, see [8].

3.2. Linking Cryptographic Protocol Models to Code

In this section, we explain how to approach the problem of creating a link between the cryptographic protocol model and its implementation.

Note that our aim is not to provide a fine-grained formal refinement from the specification to the code level. Such a refinement would require a formal behavioural semantics both of the model and the implementation of the protocol. Although such semantics exist in principle for our modelling notation (UMLsec) as well as the implementation language (Java), their treatment requires several chapters (in [1]) respectively, even an entire book on its own (such as [36]). Such a treatment would exceed the goals of the current work.

Fortunately enough, for our purposes it is not necessary to construct a fine-grained refinement relation, but it is sufficient to create a link between the points in the specification and the code where a message is received, where the required cryptographic check is performed, and where the next message is sent out (as explained below). Since our goal is to use run-time verification, rather than static verification of the code, we only need to consider these points in the code, and therefore do not depend on a full formal semantics for all of Java: instead of referring to a static semantics of Java, we will refer to a given, concrete execution trace at run-time, and with respect to that, we only need to consider the messages that are received and sent out, and make sure that the necessary checks are performed in between. Indeed, this is one of the advantages in using run-time verification compared with static verification. We only need to know which library functions need to be called to receive or send messages from or to the network, and be able to determine whether the required cryptographic checks have been performed in between. The link between the relevant points in the model and the implementation is defined formally, although, as argued above, it is sufficient to do this on a syntactic (rather than semantic) level. An example for that is given in the next section in Table 1.

There is a distinct advantage, from a practical point of view, to work with a relatively abstract specification model, which is directly linked by a mapping to the implementation level: when the implementation changes (which usually happens quite frequently during the lifetime of a piece of software like a cryptographic protocol), this minimises the amount of changes that have to be done at the model level, but as far as possible localises the necessary changes to the model-code mapping itself. This is a practical advantage, in so far as the problem of keeping a model in synch with the changing code base is one of the major impediments to a larger update of rigorous model-based development approaches in practice.

As explained above, the cryptographic algorithms are viewed as abstract functions. In our application here, 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(), and generatePrivate() methods which correspond to the abstract operations $\mathsf{hash}(E)$, $\mathsf{sign}_{E'}(E)$, $\mathsf{ver}_{E'}(E,E'')$, $\mathsf{kgen}(E)$ from Figure 3. Encryption and decryption are implemented in the JCA using the functions nextBytes() (encrypting or decrypting the next bytes of a message, depending on context), and doFinal() (finalising the encryption or decryption process). As mentioned above, our goal is not to provide a precise representation of the cryptographic generation process from the code level on the model level, but only to compare the values that were created at the points where they are received from or sent to the network.

First, we 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 creation of a mapping to the relevant elements in 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, with the goal to relate it to the model level. Afterwards, in step 2, a meaning is assigned to this data. The interpreted data elements of the individual messages are then linked to 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 with the goal to ensure that the guards specified in the sequence diagram are correctly implemented in the code.

To be able to determine the data that is sent and received, it first needs to be identified at which points in the implementation messages are received and sent out, and which messages these exactly are. To be able to do this, we exploit the fact that in many implementations of cryptographic protocols, message communication is implemented in a standardised way (which can be used to recognise 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. We assume that each message is represented by a message class (as done in many implementations such as JESSIE or JSSE). It stores the data to be written in the communication buffer. Conversely, this class can also read messages from the communication buffer (this communication principle is visualised in Figure 7). We found that this mechanism is implemented at the class level using the methods write() (for sending messages), and

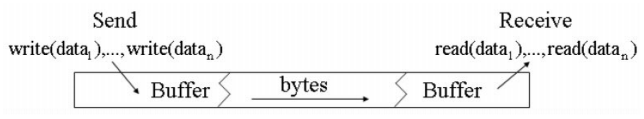

FIGURE 7. Communication in the SSL protocol

read() (for receiving them). Furthermore, the occurrences of the method write() (respectively, read()) which are called at the class java.io.OutputStream (respectively, java.io.InputStream) are used to identify the individual message parts within the communication procedure in the form of parameters that are delivered or the assignments made.

In the next subsection, we will explain how the ideas explained above were used in the application to the SSL Implementation JESSIE.

3.3. Security Monitoring the SSL Implementation JESSIE

We now explain how we applied run-time verification 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). JESSIE 1.0.1 has 27271 lines of uncommented code in Java (measured using the `sloccount` utility).

First, we explain how we applied the approach for linking cryptographic protocol models to code (as explained in the previous section) to the case of JESSIE.

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 Figure 6. Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake(), respectively doServerHandshake(), methods.

As explained above, communication is implemented as follows: With the method call msg.write(dout, version), the message msg is written into the output buffer dout. Each occurrence of such a method call can be identified and associated with the specification of sending a message in a UMLsec sequence diagram (by an outgoing arrow from the life line of the sender). The method call dout.flush later flushes the buffer. The assignment msg = Handshake.read reads a message from the buffer during the handshake part of the protocol. As an example, the code fragment for initialising and sending the ClientHello message is given in Figure 8.

In order to be able to construct a link between the implementation with the abstract model, we must first determine for the individual pieces of data how they are implemented on the code level. For example consider 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 how exactly the value of randomBytes is defined. In particular, 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.

TABLE 1. Mapping messages from symbols to program entities

Symbols	Program entities
1. C	clientHello
2. S	serverHello
3. P_{ver}	session.protocol version
4. R_C R_S	clientRandom serverRandom
5. S_{id}	sessionId
6. Ciph[ ]	session.enabledSuites
7. Comp[ ]	comp
8. Veri	Lines 1518–1557

9. D_{nb}
D_{na}
	getNotBefore()
	getNotAfter()

In the case of the SSL protocol, we had to link the symbols in its UMLsec specification in Figure 6 to their implementation in JESSIE version 1.0.1. To illustrate this, Table 1 presents nine example instances of this mapping. The first column shows the names of symbols as used in the cryptographic protocol model. The second column shows the names of corresponding program entities in the JESSIE library. Here one can also see that in general there does not need to be a one to one correspondence between the design and the code. For example, the design symbol Veri is implemented by a code fragment spread out over several lines of the code.

We now explain in particular how one can use run-time verification to increase one's confidence that the implementation adheres to the security properties previously demonstrated at the model and code levels. 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, as discussed at

```
ClientHello clientHello = new ClientHello(session.protocol , clientRandom ,
        sessionId , session.enableSuites ,comp, extensions );
Handshake msg = new Handshake(Handshake.TYPE.CLIENT_HELLO, clientHello );
msg.write(dout , version );
```

FIGURE 8. Initialising and sending the CLIENT_HELLO message

the beginning of the last section, run-time verification can provide a higher level of assurance for crypto-based software than for example model-based testing, since full test coverage is in general not achievable for highly interactive and complex software like cryptographic protocols.

According to the information that is contained in a sequence diagram specification of a cryptographic protocol, the run-time verification needs to keep track of the following information:

(1) *Which data is sent out?* and
(2) *Which data is received?*

The run-time 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.*

An example of such a property in the case of the SSL-protocol specified in Figure 6 is given by the following requirement that arises from the above discussion:

> "ClientKeyExchange$(enc_K, (PMS))$ is not sent by the client until it has received the Certificate$(X509Cer_s)$ message from the server, has performed the validity check for the certificate as specified in Figure 6, and this check turned out to be positive."

Next, we explain how to capture such a requirement using LTL. Together with Figure 6, this requirement gives rise to the following set of atomic propositions:

$$AP := \{\mathsf{ClientKeyExchange}(enc_K, (PMS)), \mathsf{Certificate}(X509Cer_S)\},$$

whose names correlate with the ones displayed in Figure 6. Notice that LTL as introduced above does not cater for parameters. Therefore, parameters in an action's name are not a semantic concept, but merely syntactic sugar to ease readability and establish a link with the names used in Figure 6. The link from symbol names to the actual names used in the monitor, and finally in the implementation code of JESSIE, is also exemplified by Table 5. Based on AP we can now formalise the required property in LTL as follows:

$$\varphi := \neg\mathsf{ClientKeyExchange}(enc_K, (PMS)) \; \mathbf{W} \mathsf{Certificate}(X509Cer_S).$$

The formula uses the "weak until" operator, which in particular allows for the fact that if the certificate is never received, then the formula is satisfied if in turn the message ClientKeyExchange$(enc_K, (PMS))$ is never sent. This meets our intuitive interpretation of the "until" in the natural language requirement because if, for example, a man-in-the-middle attacker deletes any certificate message sent by the server, we cannot possibly demand that ClientKeyExchange$(enc_K, (PMS))$ should be eventually sent by the client. The derived monitor will later signal the value $\top$ ("property satisfied") once the certificate was received and checked, $\bot$ ("property violated") if the client sends the key without a successful check, and it will signal the value ? ("inconclusive") as long as neither of the two conditions holds. Recall that the stream of events that is processed by the monitor consists of elements from 2^{AP} (i.e., the powerset of all possible system actions). That is, at each point in time, the monitor keeps track of *both* events: the sending of ClientKeyExchange$(enc_K, (PMS))$ and the receiving of Certificate$(X509Cer_S)$. Hence, as long as none of the events is observed, the monitor basically processes the empty event.

Once we have formalised the natural language requirements in terms of LTL formulae as above, we can then use the tool from [11] to automatically generate finite state machines (FSMs) from which we derive the actual (Java) monitor code. The FSMs obtained from the tools are of type Moore, which means that, in each state that is reached, they output a symbol (i.e., ?, $\top$ (TOP), $\bot$ (BOT), or ?). States are changed as new system actions become visible to the monitor. In that sense, the states keep track of the context in which new actions are to be interpreted. For example, there may be an action, such as the sending of a secret key, which constitutes a security violation in one context, but is a necessary and desired action in another context, e.g., as part of a protocol. The monitor's states keep track of the current context, and the reaching of a new state means reaching a new context in which to interpret future actions. The FSM generated for the run-time security property φ is given in Figure 9. The initial state is $(0,0)$ whose output is ?. If event $\{cert\}$ occurs, short for $\{\mathsf{Certificate}(X509Cer_S)\}$, then the monitor takes a transition into state $(1,-1)$ and outputs $\top$ to indicate that the property is satisfied. On the other hand, if neither $cert$ nor cke, short for ClientKeyExchange$(enc_K, (PMS))$, occurs, then the automaton remains in $(0,0)$ and outputs ? anew, indicating that so far φ has not been violated, but also not been satisfied. A violation would be the reaching of $(-1,1)$, if event $\{cke\}$ occurs (before $cert$), such that the monitor would output $\bot$. Here is an example run of the client

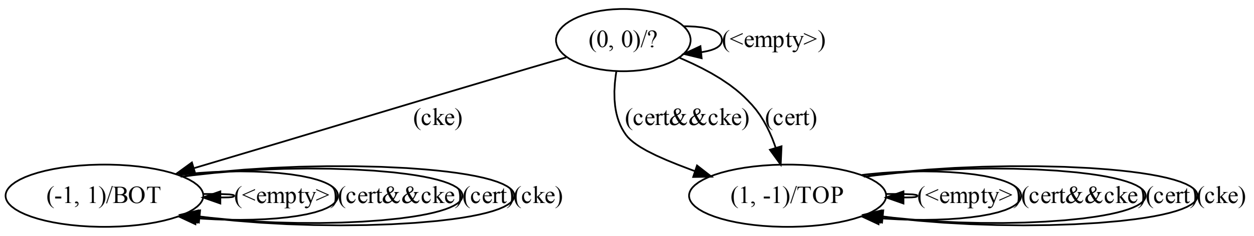
(0, 0)/?
(<empty>)
(cke)
(cert&&cke)
(cert)
(-1, 1)/BOT
(<empty>)(cert&&cke)(cert)(cke)
(1, -1)/TOP
(<empty>)(cert&&cke)(cert)(cke)

FIGURE 9. Automatically generated FSM for the property

which first yields ? as output for three time steps until, finally, $\top$ is returned in the fourth because the message was sent but the certificate also received and checked:

$$u := \langle \{\}, \{\}, \{\}, \{cert, cke\} \rangle.$$

At this point this particular monitor may stop monitoring for the remaining session. On the other hand, consider the following run:

$$u' := \langle \{\}, \{\}, \{\}, \{cke\} \rangle.$$

This run is indicative that the client has attempted to send $\mathsf{ClientKeyExchange}(enc_K, (PMS))$ prematurely, resulting in the monitor returning $\perp$ in the fourth time step. In formal terms, for all $v \in \Sigma^\omega$ we have $uv \models \varphi$ (i.e., $[u \models \varphi] = \top$), and for all $v \in \Sigma^\omega$ we have $u'v \not\models \varphi$ (i.e., $[u' \models \varphi] = \perp$). On the other hand, if we shortened u and u' by one observation, we obviously would have $[u \models \varphi] = ?$ and $[u' \models \varphi] = ?$.

Further properties as the ones above, which are monitorable in this particular application, are also discussed in [8]. The relationships between symbol names used in these specifications, the monitor FSMs, and the code are then given in Table 5 on page 21.

A Note on Efficiency The monitors we generate for each security property are *minimised* in a sense that we find a smallest possible state machine which corresponds exactly to the language of the security property by exploiting the well-known Myhill-Nerode equivalence relation between states of a finite automaton (cf. [37]). The latest version of our monitor generation tools [11] perform this minimisation and thus return the smallest monitor possible for a given language. In other words, it is not possible to find a smaller monitor without altering the monitored language, i.e., the generated monitors are *optimal*. Therefore, the efficiency of the proposed method solely depends on the respective security property chosen, i.e., the formal language it gives rise to and the means by which the state machines are implemented for the application at hand. Specifically, efficiency depends on

(1) the number of states in the monitor, which is the smallest number possible by the Myhill-Nerode equivalence,
(2) the time it takes to accept a system action and to change state in the monitor, and
(3) the time it takes for the monitor to emit the corresponding output symbol.

However, if the monitor contains only a very small number of states, as was the case in our examples, then it is very difficult to effectively measure items 2 and 3 in the above list of items, because they require only microseconds (or less) and exact measurements in these ranges can only be obtained reliably using real-time operating systems. However, due to the monitors being optimal in the above sense, we have a guarantee that the run-time overhead is minimal, which, indeed, resulted in no noticeable performance changes in our application. It may, however, be the case that for very involved specifications to be monitored in other application domains, that there is a noticeable overhead and that, indeed, additional resources are necessary to facilitate this technique. After all, the monitors are of worst-case exponential size with respect to the specification, and sometimes the worst case cannot be avoided, which is particularly limiting when the formula was already of a large size to begin with. Our experiences with the given application, however, did not reveal such cases, which leads us to believe that in many practical situations the worst-case behaviour can be avoided. Moreover, after minimalisation, the state-space of the generated monitors was ≤ 10 states, which is a good indication for how efficiently this method can be implemented.

Notice also that run-time verification is a method which scales well, in a sense that the size of the system under scrutiny does not impact on the efficiency of the method. Run-time verification operates on a concrete behaviour, whereas static verification techniques like model checking or ones which try to establish correctness with the help of a theorem prover, as we have laid them out in Section 2, explore the overall state-space of a system imposed by a model representation of it. Naturally, as system sizes increase, it affects the efficiency of such techniques, whereas run-time verification stays constant.

4. MAINTAINING TRACEABILITY UNDER EVOLUTION

There are two kinds of traceability associated with our use of run-time verification. First, as discussed in Section 3, we use it as a tool to trace high-level security properties beyond code and down to the actual execution level. Second, we have to tackle traceability within our run-time verification framework itself as security properties and the code change, which may affect the generated monitors. In this section, we focus mainly on the second kind of traceability with regard to run-time verification; that is, we examine how our LTL properties and monitors are affected by changes of the high-level security properties and of the implementation code.

We now explain how to maintain the traceability link constructed using the approach explained in the previous section in the presence of code evolution. We then explain how to apply the approach for run-time security verification explained in the previous section in this situation.

4.1. Evolution as Code Refactoring

Software *refactoring* [38] by definition changes the internal structure of an implementation without changing its externally observable behaviour. By this definition, refactoring transformations are program transformations that preserve externally observable program behaviour. Note that therefore transformations that change the externally observable behaviour of a program are beyond the definition of refactoring, and thus not considered in the following. Modification of the behaviour due to refactoring would be considered a bug of the refactoring engine that needs to be fixed eventually.

In practice, the refactoring engine in programming IDEs implements a subset of possible refactoring transformations which are commonly used in programming activities, such as renaming, extracting methods, etc.

For example, the general refactoring engine in Eclipse is provided by a set of plug-ins called the refactoring Language Toolkit (LTK)[39], which allows one (1) to perform refactoring operations, (2) to save the history of refactoring operations into an XML-based script, and (3) to apply a refactoring script automatically. The plug-ins are applicable to any programming or specification language. The Java Development Tool (JDT), for example, instantiates LTK with a number of Java-specific refactoring operations. Rather than refactoring Java source, another refactoring tool in the Plugin Development Environment (PDE) instantiates LTK with a number of refactoring operations specific for the plug-in metadata.

We use refactoring scripts to maintain traceability between a design and its evolving implementations. Modern IDEs 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 externally observable behaviour of the program. Otherwise, Eclipse would reject the execution of an operation that might change the behaviour. For example, renaming class field x to y will change behaviour if there is already a local variable y in some method(s), because the renamed references to x will now become references to the local variable. This is carefully excluded by Eclipse.

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* (see below) 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.

An Illustrative Example To illustrate Java refactoring, Figure 10 shows a running example specific to Eclipse JDT, where a series of refactoring operations are applied to a small "Hello World" program.

Assume that initially the source file abc.java is located at a source folder src in the project abc. A series of refactoring operations are applied as follows. Step 1: The class abc is renamed to hello and abc.java is also renamed to hello.java, accordingly. This refactoring operation is called *rename.type*. Step 2: The statement System.out.println is extracted into the body of a new method print_ hello(). This operation is called *extract.method*. Step 3: The expression "Hello" is explicitly assigned to a new local variable string. This operation is called *extract.temp*. Finally, Step 4: The method main2 is renamed to a new method name main. This last operation is called *rename.method*.

After performing the above refactoring operations in Eclipse one can save the history into a refactoring script. Such a script can be automatically applied on the original code again to replay the changes. Figure 11 shows a snippet from the refactoring script in XML format. It briefly specifies the *rename.type* and *extract.method* operations used in the first two steps.

Every refactoring is recorded as an XML element refactoring, whose attributes specify the operation. Every operation has an identifier ID, indicating the type of the operation. Here, org.eclipse. jdt.ui.rename.type is the internal name used by JDT for *rename. type* refactoring. For readability, we omit the common prefix in the following and call it rename.type. The target of a refactoring operation for rename.type is a new class name, whereas the target for extract.method is a new method name. They are completely specified by the name attribute. On the other hand, the source of a refactoring operation is suggested by attributes including project, input and optionally selection. The values of these attributes typically indicate the context of an operation. The project attribute specifies the subject project of the refactoring operation; the input attribute specifies the source folder, package and class name in which the source element is refactored;

```
/* $workspace/abc/src/abc.java */
public class abc {
    public void main2(String args[]) {
        System.out.println("Hello");
    }
}
———— Step 1. rename.type ————
/* $workspace/abc/src/hello.java */
public class hello { ... }
———— Step 2. extract.method ———
public void main2(String args[]) {
   print_hello();
}
private void print_hello() {
   System.out.println("Hello");
}
———— Step 3. extract.temp ————
   String string = "Hello";
   System.out.println(string);
———— Step 4. rename.method ———
public class hello {
   public void main(String args[]) {
     print_hello();
   }
   private void print_hello() {
     String string = "Hello";
     System.out.println(string);
   }
}
```

FIGURE 10. A running example illustrates refactoring

```
<?xml version="1.0"?>
<session version="1.0">
<refactoring comment="..."
   id="org.eclipse.jdt.ui.rename.type"
   description="Rename type 'abc'"
   project="abc" input="/src&lt;abc.java[abc"
   name="hello" ... />
<refactoring comment="..."
  description="Extract method 'print_hello'"
  id="org.eclipse.jdt.ui.extract.method"
  project="abc" input="/src&lt;{hello.java"
  name="print_hello" selection="64 28"
  ... />
 ...
</session>
```

FIGURE 11. Operations of Eclipse refactoring script (cf. Figure 10)

the selection attribute, when used, specifies the exact offset and length of the string selected for the refactoring.

In our example the extract.method refactoring is applicable only if the selection of a substring of 28 characters starting from the offset 68 in hello.java matches the statement to extract, character by character. Given such strict specifications of refactoring contexts in Eclipse, we can see that existing refactoring scripts are inadequate if source code has been modified by evolution or by previously applied refactoring operations, or when source code from a different library implementation is used. For example, it is required to modify the offset/length value if an extract.temp operation was applied earlier.

4.2. Maintaining Model-Code Traceability

In this subsection, we explain how to maintain traceability between a UMLsec specification of a cryptographic protocol and its implementation while the code evolves (cf. Figure 12).

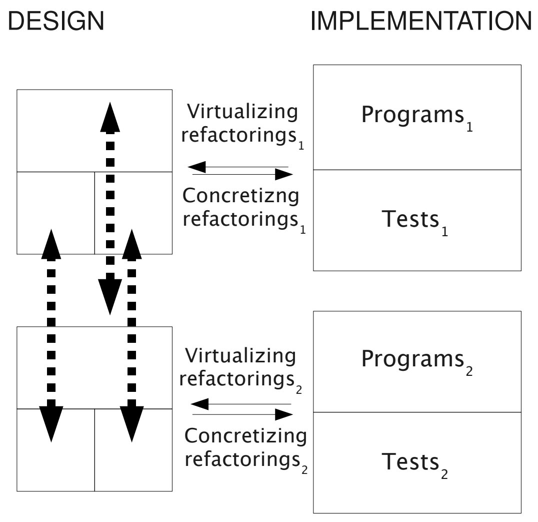
DESIGN
IMPLEMENTATION
Virtualizing refactorings$_1$
Concretizng refactorings$_1$
Programs$_1$
Tests$_1$
Virtualizing refactorings$_2$
Concretizing refactorings$_2$
Programs$_2$
Tests$_2$

FIGURE 12. Traceability for reuse

We present our new refactoring engine that overcomes the limitation of the native Eclipse JDT refactoring operations, while making the refactoring operations reusable for maintaining design traceability in different legacy code.

Specifically, we need to map any symbolic name *S* that appears in the design model to an identifier *I* on the implementation level.

Refactoring scripts are used for maintaining such traceability: they guarantee that the externally observable behaviour 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 program entities to names on the design level and
(2) to convert the names on the design level to names in the implementation.

When a relation between a symbol *S* and an entity *I* in the program is established, it will be maintained through a number of refactoring operations that transform every occurrence and update every reference of *I* into *S*.

When the program entity already has an identifier in a form of class, method, field or local variable, renaming operations such as *rename.type*, *rename.method*, *rename.field* and *rename.local.variable* can be used; when the program entity does not have an associated identifier, then extracting operations such as *extract.method*, *extract.temp* and *extract.field* can be used to directly extract an identifier named by *S*.

The renaming operations, when applied in low granularity (e.g., *rename.local.variable*), are typically change sensitive as a selection offset/length is required to specify the exact context of source. The extracting operations, by definition, always need to specify the context of the source explicitly.

The mapping between symbols and program entities is not one to one. The same symbol from the design model may be implemented differently in different contexts. Therefore

```
SPECIFICATION := OPERATION SPECIFICATION
OPERATION := '@' '{' NAME ',' FIELDS '}'
NAME := Identifier [ '.' NAME ]
FIELDS := FIELD [',' FIELDS]
FIELD := KEY '=' VALUE
KEY := Identifier
VALUE := String
```

FIGURE 13. The extended BNF for the syntax of our refactoring language

```
@{org.eclipse.jdt.ui.rename.type,
  project="abc", source="src", package="",
  class="abc",  name="hello"
}
@{org.eclipse.jdt.ui.extract.method,
  project="abc", source="src", package="",
  class="hello", method="main",
  toclass="hello", name="print_hello",
  regexp="S.*(\"Hello\");",
  count="1"
}
```

FIGURE 14. Our specification for refactoring (cf. Figure 11)

more than one refactoring operation can be applied to resolve the symbol names. Since a symbol may even be called differently in different parts of the program, the actual program entities have to be checked to find out whether they are the same throughout the design. Such checks must in particular ensure that the name can be differentiated by using the context of the messages.

If *S* is a complex design element, such as a message in a message sequence chart, its mapping may at the same time require a mapping from its arguments to their corresponding identifiers. In order to create such a mapping, a sequence of basic refactoring operations needs to be performed. Therefore, such dependencies among refactoring operations need to be respected.

As we have mentioned, the refactoring of *I* to *S*, when applied after other editing/refactoring operations, have to be carried out independently of the previous changes.

4.3. Reuse Support for Traceability Refactoring

One can reuse the traceability information discovered when linking the implementation to the UML model. For example, this can be done 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 plug-in that can apply parameterised refactoring operations[1]. Our refactoring tool is implemented on top of LTK refactoring plug-ins, which support languages beyond Java. In order to limit the changes to the existing refactoring engine, we invoke the context-specific refactoring operations in JDT by instantiating a scripting template with the parameters derived from our specifications.

In [40], Krueger classified software reusability as five connected facets: abstraction, classification, selection, specialisation and integration. Our traceability refactoring engine supports this view.

Abstraction. An extended BNF grammar of the abstract refactoring language is given in Figure 13.

A specification consists of one to many refactoring operations. Every operation has a name indicating the class that handles the refactoring and one to many fields. A field is a pair of a key identifier and a value string. It is up to the refactoring class to decide the concrete list of fields to be used.

Our declarative specification language abstracts away context-sensitivity of existing refactoring operations and can describe generally any refactoring operation supported by LTK, beyond Java. The refactoring context is parameterised to remove certain change-resisting dependencies (e.g., selection in Figure 11). Similar in format to that of BibTeX, a specification consists of a list of entries. Each entry is made of a list of fields, separated by a comma. The first field is a key, which matches one type of the existing refactoring operations (as in JDT). The remaining fields are in the form of name="string" pairs, where the quoted string can span multiple lines. As in Java, every quotation in the string must be escaped. Depending on the type of refactoring operations, the number of required fields may vary. The main reason why we chose this format is to support variability for recording refactoring operations.

Corresponding to Figure 11, the snippet in Figure 14 lists two refactoring operations in our specification language.

Selection and Specialisation. Most fields have evident meaning and usage as they correspond to the attributes in the Eclipse refactoring scripts. We introduce the new fields to compute the context (input) of the source element, such as source, package. The fields regexp and count in this specification indicate a selection to be refactored that is matching a regular expression, counted from the beginning. Our regular expression-based selection for context-sensitive refactoring operations increases the chance of reusability when changes happen to the code. In the implementation, we can actually construct a regular expression from a normal one by replacing white spaces with an arbitrary number of white spaces. In this way, even if a programmer or a code formatter inserted some indentation, the selection can still be matched. Introducing count is done mainly to be able to selectively refactor some instances of matching selection rather than the first one. When unspecified, the first matching selection will be chosen. The selection parameter is specialised from the other parameters by parsing the Java source file, searching for the method name in the given class to obtain the offset to the method in the source range, and then searching for the local variable in the source of the selected method and adding its relative offset to the method to obtain the absolute offset to the file.

Classification. As refactoring consists of a sequence of operations, we *classify* existing refactoring operations by

[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 [4].

TABLE 2. Refactoring operations parameterised by our refactoring tool

ID	change resistant?	context	source selection	specified in Eclipse	our specification
org.eclipse.jdt.ui.rename.project	no	workspace	project	project	project
org.eclipse.jdt.ui.rename.folder	no	project	folder	folder	folder
org.eclipse.jdt.ui.rename.package	no	folder	package	package	package
org.eclipse.jdt.ui.rename.type	no	package	class	class	class
org.eclipse.jdt.ui.rename.method	no	class	method	method	method
org.eclipse.jdt.ui.move.method	no	class	method	method	method
org.eclipse.jdt.ui.extract.method	yes	class	statements	(offset, len)	(regexp [, count])
org.eclipse.jdt.ui.rename.local.variable	yes	method	variable	(offset, len)	(regexp [, count])
org.eclipse.jdt.ui.extract.local.variable	yes	method	expression	(offset, len)	(regexp [, count])
...	...	...	...	...	...

context-sensitivity and discuss its impact on exchangeability and invertibility.

Context-free operations are more reusable whereas context-resistant or sensitive ones require more care. Since it is more likely to have the other parts of the code changed rather than the pattern of regular expressions, our new refactoring operation becomes less sensitive to code changes. According to our experience, when relaxed patterns are used in the regular expression, the context specification of refactoring operation is more tolerant to changes.

In practice we have found that if one performs larger-granularity refactoring operations (say, a) earlier than smaller-granularity ones (say, b), the reusability of refactoring operation sequences can be increased. To allow reordering operations, side-effects of an operation on the context of another must be captured by changes to their parameters, i.e. $a \otimes b = b' \otimes a'$. For example, the two operations in our running example are not exchangeable. If one were to swap their order, one needs to accordingly apply the latter refactoring to the refactoring script of the former one. If one would apply the *extract.method* operation first, then the rename.type operation should be applied to the specification such that it is a method in the class abc rather than the class hello being extracted.

In Table 2, we list some JDT refactoring operations that have been parameterised in our refactoring engine. We also show which JDT operations are considered change resistant and a brief description on how such limitations are resolved.

Integration. After selection and specialisation, 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 a refactoring if they choose to, and to avoid manually constructing the specification from the saved refactoring history in Eclipse.

The implementation of our refactoring plug-in 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 a refactoring. This allows us to verify if there are any potential maintenance problems arising from the operation. For example, when renaming a variable to R_C, we can see a warning message from the Eclipse IDE that, by programming convention, it is not recommended to let the name of a variable start with capital letters. However, since our purpose is to facilitate the reuse of traceability in security analysis, such a renaming does not affect programmers because they can always edit the original source code.

Another utility program we implemented is a transformation that converts an XML-based refactoring script from the Eclipse IDE into our own specification language. By such a conversion, a string selected by offset and length is replaced with a regular expression and its count of its matching occurrence. For the string selected, the utility generates a regular expression with wildcards and an occurrence count such that it could match precisely with the selection string in the refactoring context (e.g., a method body), while being agnostic to the change to other parts of the program. For example, if the *extract.method* is applied to a set of statements, they will be remembered by the generated regular expression so that the method can be matched even when the other part of the method is changed.

After translation, the resulting specification is still further customisable. We also implemented a headless tool to invoke the functionality of the automated button as an RCP command. The argument of the command provides the name of a refactoring specification file.

4.4. The SSL Case Studies

In general, the run-time verification of a protocol like SSL should be invariant to implementation changes if the properties that are to be monitored are derived from the specification of the protocol (unless, of course, the specification of the protocol changes). In other words, if we have a security property that we want a correctly operating implementation of the SSL-protocol to adhere to at run-time, we want a modified implementation to also adhere to this property regardless of how it achieves it internally. This view asserts that run-time verification considers the system under scrutiny as a "black box".

4.4.1. *Evolution in the* JESSIE *Case Study*

To perform the model-based security analysis as explained above on a different version of JESSIE, one only needs to modify the specifications of the refactoring operations that provide the traceability of the model to the implementation level, without making any other adjustments to our refactoring engine. In particular, we considered the two versions JESSIE 1.0.0 (released on June 9, 2004 according to its CVS repository) JESSIE 1.0.1 (released on October 12, 2005 according to its CVS repository).

However, in the case of monitoring JESSIE, we linked the run-time verification directly to code elements and, therefore, changes to code may affect an existing integration of a monitor. For example, in Section 3, we have defined an abstract run-time security property in LTL and, in doing so, have performed a mapping from elements in the design model (and therefore also in the LTL formula) to elements used in the monitor code. Moreover, there also exists a mapping from elements used in the monitor code to elements used in JESSIE's code. Table 5 exemplifies these mappings for three different run-time security properties (where the first is our running example, discussed in Section 3).

Evolution Inside the org.metastatic.jessie.provider package in JESSIE the 1.0.1 version has got 24 code block differences compared to that of 1.0.0 version. These changes cause that the selection-sensitive operations in the refactoring history script saved from Eclipse cannot be applied to JESSIE 1.0.0. After converting the script into our specification language, all of the refactoring operations (some of which are listed in Table 3) become reusable in our enhanced refactoring engine (cf. the column JESSIE 1.0.0). The only necessary change made to our original refactoring specification for JESSIE 1.0.1 was a global substitution of the project attribute for all operations from jessie-1.0.1 to jessie-1.0.0.

As part of the library release, two model-based unit tests for the message sequences in JESSIE 1.0.1 were provided: testclient.java and testserver.java. After refactoring, we were able to reuse them for the two other implementation libraries as well.

Since the "hooks" required in the code are different but conceptually the same, we focussed on only the first of the three properties in this paper given again in Table 5. All of the mentioned code can be found inside SSLSocket.java. The first column of Table 5 shows the name of an entity on the design model level as well as its symbol name in the corresponding LTL formula, the second column shows the action symbol as it is used within the generated monitor, and the last column displays the corresponding entity in the JESSIE source code. Notably, some properties share symbol names, which has to be respected by potential refactoring steps. That is, when we apply refactoring steps that affect elements in the given table, we have to make the according changes there as well to notify the run-time verification framework of the occurring changes. Internally, our refactorings reflect the links represented by that table and its crucial symbol names and code segments. This gives us a straightforward, but manageable, means to evolve our monitors along with code changes. As the properties that we monitor are fairly generic properties that are derived from the SSL-protocol specification itself, they have not changed between versions 1.0.1 and 1.0.0 of the SSL-protocol implementation JESSIE. However, this type of book-keeping does not in general give us any indication when refactorings or other code changes do *not* affect our monitors, because code which has been identified relevant to the monitors may have become "dead code" by a modification outside the scope of our book-keeping. Note however that dead code detection can typically also be automated, e.g. using static analysis (cf. [41]).

Evolution Beyond Simple Refactoring In order to preserve externally observable behaviour, the refactoring steps defined in previous sections represent relatively small and simple changes on the code base (e.g. consistent renaming of identifiers). However, in practice, systems often undergo more significant evolutions which may in particular not be behaviour-preserving. In these cases, the definition of the LTL formula to be monitored may have to be adapted manually to account for the system evolution. In this section, we demonstrate this in terms of an example.

We consider the situation where an initial version of a protocol implementation does not provide for dedicated error handling in the case that one of the cryptographic checks in the protocol is violated. We investigate how a monitor for such an implementation will have to evolve if the implementation is adjusted to provide dedicated error handling, to make sure that the error handling leads to a fail-safe system state. This will prevent the protocol implementation from proceeding with an insecure protocol execution, e.g. by sending out secret information even though the cryptographic checks were violated.

We therefore distinguish between the following cases:

(1) The system fails and does not reach a fail-safe state (monitor returns $\bot$).
(2) The system succeeds *or* reaches a fail-safe state, assuming an error occurred (monitor returns $\top$).
(3) Neither of the two conditions holds (monitor returns ?).

What we have done is, basically, added an *exception* to our rule, and thereby mapped two different events to one truth value, namely $\top$. This, however, is not uncommon in specifying behaviour of software and systems. For example, exceptions are incorporated into many different specification languages that are based on LTL and regular languages (cf. [34, 42, 43]). The one presented in [42], SALT, introduced the **accepton** *x* directive for this purpose, where *x* represents the exception.

As we are using LTL directly in this paper, we give a straightforward extension of our previously used specification (see Section 3) that caters for such an exception and demonstrates the concept:

$$\begin{aligned}\varphi_{fs} := \quad & \neg\mathsf{ClientKeyExchange}(enc_K, (PMS)) \\ & \mathbf{W}(\mathsf{Certificate}(X509Cer_S) \\ & \quad \vee \mathsf{failsafe}).\end{aligned}$$

Using our monitor generator [11], it is easy to verify in terms of the resulting monitor FSM that this extension has the

TABLE 3. Refactorings for the traceability to the protocol (cf. Figure 6)

Messages in sequence	op.	diff	Time (sec)
S1: $C \to S : (P_{\text{ver}}, R_C, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$	7	31	13.891
S2. $S \to C : (P_{\text{ver}}, R_S, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$	5	20	9.437
S3. $S \to C$: Certificate[X509Cert$_s$]	2	2	1.474
S4. C : Veri(X509Cert$_s$)	2	2	3.854
...	...	...	...
Total of 7 messages and 3 checks	27	86	40.303

desired effect. Had we used the LTL meta-language SALT as introduced in [42], we could have simply added an exception as follows

$$\begin{aligned}\varphi'_{fs} := \; & (\neg \mathsf{ClientKeyExchange}(enc_K, (PMS)) \\ & \mathbf{W} \mathsf{Certificate}(X509Cer_S)) \\ & \mathbf{accepton}\ \mathsf{failsafe},\end{aligned}$$

which would then have been translated into the above LTL formula. While with short formulae such as the above, it does not seem to make any difference as to whether meta-level constructs like **accepton** are employed, which subsequently "weave" the exception into all subformulae of a given specification, more comprehensive formulae may be difficult to specify in LTL alone and without such directives. Semantically, however, a language such as SALT is equally expressive to LTL. We therefore abstain from discussing it further in this paper as the resulting monitors are the same.

For all the 19 symbols, 7 messages and 3 checks in Figure 6, in total we have defined 27 refactoring steps in the specification to maintain the traceability between the protocol design and the JESSIE 1.0.1 code. The third column of Table 3 shows the count of changed segments by the refactoring steps. Using `diff`, each block of changes, even when they contain multiple lines, is counted as one. When the number of changed blocks is larger than the number of steps, changes have happened to more than one places on average. The last column shows the performance, i.e., how much time in seconds it took to perform the refactoring steps using our tools. Note the time required for the refactoring steps varies depending on its type and the number of occurrences in the code. For example, renaming a sessionId field into S_id took only 0.141ms whereas renaming a local variable sessionId into S_id took 1.484ms. The automatic execution of all the steps took about 40 seconds running our plug-ins inside Eclipse SDK 3.3 on a dual-core laptop (with a CPU running at 2 x 1.8GHz). Given the significant pay-off provided by the fact that the externally observable behaviour of the code is preserved during the complex refactoring steps, such performance figures do not impose a bottleneck within the overall process. On the contrary, much more time is spent on the security analysis and the manual creation of the refactoring steps, which will be paid back by reusing the scripts on different implementations.

4.4.2. Maintaining Monitor-Code Traceability

To illustrate this refactoring mapping, Table 4 presents some instances of such a mapping for our example

TABLE 4. Refactoring program entities in a traceable way

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

implementation. The first column shows the names of symbols as used in the cryptographic 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 externally observable behaviour of the program is preserved (e.g. renaming identifiers in a way that is ensured not to create any conflicts). 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.

4.4.3. Reusing JESSIE *Refactoring Transformations for* JSSE

We also investigated how to reuse the model-code traceability links for SSL from the JESSIE project for JSSE, another implementation of SSL. JSSE is part of Sun's Java Secure Sockets Extension (JSSE), a library in the standard JDK since version 1.4, released by Sun from version 1.6 onwards as an open source project called OpenJDK. Specifically, we considered JSSE 1.6 (released on May 8, 2007). The source code of the JSSE library can be checked out from its Subversion repository: https://openjdk.dev.java.net/svn/openjdk/jdk/trunk/j2se/src/share/classes/sun/security/ssl. In this case, we found that most of the refactoring operations cannot be applied as is. The doHandshake protocol is mainly implemented in the class SSLSocket of the JESSIE 1.0.1 library, whereas in the JSSE library implementation in the OpenJDK 1.6 (hereafter called JSSE 1.6), the protocol is mainly implemented in the class sun.security.ssl.HandshakeMessage. Nevertheless, the naming of the symbols can be traced to the implementation.

Table 6 lists the mappings from the symbols in Table 1 to their naming in the JSSE library. To reuse the existing refactoring operations, we have to

TABLE 5. Mapping model elements to monitor code and JESSIE (SSLSocket.java)

Model / LTL symbol	Monitor	Concrete representation in JESSIE
1: $\neg\mathsf{ClientKeyExchange}(enc_K,(PMS))\mathbf{W}\mathsf{Certificate}(X509Cer_S)$		

$\mathsf{ClientKeyExchange}(enc_K,(PMS))$	cke	`ProtocolVersion v = (ProtocolVersion)`
		`    session.enabledProtocols.last();`
		`byte[] b = new byte[46];`
		`session.random.nextBytes (b);`
		`preMasterSecret = Util.concat(v.getEncoded(), b);`
		`EME_PKCS1_V1_5 pkcs1 =`
		`    EME_PKCS1_V1_5.getInstance((RSAPublicKey)`
		`    serverKex);`
		`BigInteger bi = new BigInteger(1, pkcs1.encode`
		`    (preMasterSecret, session.random));`
		`bi = RSA.encrypt((RSAPublicKey) serverKex, bi);`
		`ClientKeyExchange ckex = new ClientKeyExchange`
		`    (Util.trim(bi));`

$\mathsf{Certificate}(X509Cer_S)$	cert	`Certificate serverCertificate = (Certificate)`
		`    msg.getBody();`
		`X509Certificate[] peerCerts =`
		`    serverCertificate.getCertificates();`

2: $(\neg\mathsf{Finished}(HashMD5(\ldots\mathbf{W}\mathsf{Arrayequal}(md5_s,md5_c))\wedge(\mathbf{F}\mathsf{Arrayequal}(md5_s,md5_c)\Rightarrow\mathbf{F}\mathsf{Finished}(HashMD5(md5_s,ms,\ldots)))$		

$\mathsf{Finished}(HashMD5(md5_s,ms,PAD1,PAD2))$	finished	`finis = generateFinished(version, (IMessageDigest)`
		`    md5.clone(), (IMessageDigest) sha.clone(), true);`
		`msg = new Handshake(Handshake.Type.FINISHED, finis);`

$\mathsf{Arrayequal}(md5_s,md5_c)$	equal	`if (!Arrays.equals(finis.getMD5Hash(),`
		`    verify.getMD5Hash()) ||`
		`!Arrays.equals(finis.getSHAHash(),`
		`    verify.getSHAHash()))`
		`...`

3: $\neg\mathsf{Data}\mathbf{W}\mathsf{Arrayequal}(md5_s,md5_c)$		

$\mathsf{Arrayequal}(md5_s,md5_c)$	equal	`if (!Arrays.equals(finis.getMD5Hash(),`
		`    verify.getMD5Hash()) ||`
		`!Arrays.equals(finis.getSHAHash(),`
		`    verify.getSHAHash()))`
		`...`

Data	data	(Various stream read and write methods.)

instantiate their specifications with different parameters for its source (i.e., project, folder, package, class) and its context (i.e., regexp, count). In some cases even the type of refactoring operation needs to be changed. For example, Veri(X509Cert$_s$) is refactored by the *extract.method* operation in JESSIE (Table 4). However, to obtain the same symbol, a *rename.method* operation in JSSE is required (Table 6).

Such changes, however, do not influence the target name attribute for the operations because they are derived from the same protocol design. Modifying the refactoring specifications might seem a lot of work. However, we experienced little difficulty in applying them with the help of automated execution of the declarative refactoring specification. The benefit of such an effort is that we can reuse the model-based security test cases.

TABLE 6. Symbol-code mappings for JSSE

Symbols	JSSE 1.6
1. C	HandshakeMessage.ClientHello
2. S	HandshakeMessage.ServerHello
3. P_{ver}	protocolVersion
4. R_C R_S	clnt_random svr_random
5. S_{id}	sessionId
6. Ciph[ ]	cipherSuites
7. Comp[ ]	compression_methods
8. Veri	CertificateVerify.verify()

9. $D_{\mathrm{notBefore}}$
D_{notAfter}
	cert.getNotBefore()
	cert.getNotAfter()

5. SECURITY HARDENING

Using the approach to run-time security verification explained in the previous sections, one can raise an alarm at run-time in case of a security violation, and terminate the given protocol execution, before the secret is leaked out to the network. In such a situation, it would however be even more useful if one could go a step further, and remove the security vulnerability in the implementation that has been detected in this way to make sure the same problem will not

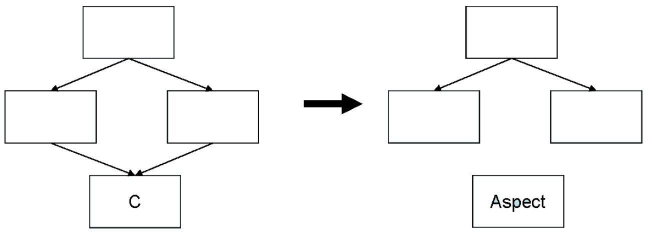

FIGURE 15. Comparing component-based with aspect-oriented systems in light of the inverse of control principle

```
/* HelloWorld.java */
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello␣world!");
  }
}

/* GoodbyeWorld.java */
public class GoodbyeWorld {
  public static void main(String[] args) {
    System.out.println("Goodbye,␣world!");
  }
}

/* HelloFromAspectJ.aj */
public aspect HelloFromAspectJ {
  pointcut mainMethod() :
    execution(public static void
        main(String[]));
  after() returning : mainMethod() {
    System.out.println("Hello␣from␣
        AspectJ");
  }
}
```

FIGURE 16. An illustrative aspectJ program

appear again. In this section, we explain how this is achieved in an approach making use of automated instrumentation techniques.

One often has to fix a vulnerability in multiple places of the code, making it difficult to maintain the changes consistently. In order to automate the vulnerability fix for security hardening, we therefore choose to apply aspect-oriented programming (AOP) techniques.

In the following subsection, we explain the basic concepts of aspect-oriented programming (AOP) in detail.

5.1. Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP, [44, 45]) separates crosscutting concerns that tangle the code into aspect modules. The tangled code at various control flow points (so-called “joinpoints”) are encapsulated into a module when they match with the signatures of pointcut expressions. The functionality of the existing code can be altered by weaving additional statements (so-called “advices”) before, after or around the existing joinpoints. AOP has advantages for maintenance as one can change the crosscutting behaviour of the system without directly modifying the source. AOP is supported in systems such as aspectJ [44] and Hyper/J [45] and fully supported in Java IDEs such as Eclipse through the AJDT project.

AOP Principles In a previous paper, we compared the fundamental difference in the methods reflected by component-based programming and AOP [46]. Figure 15 illustrates two modularisations to divide a problem into subproblems and to compose their solution later. In the component-based manner (left), the composition requires at various points an explicit invocation of the component module, whilst in the AOP manner (right), the aspect module has two parts: *pointcuts* and *advices*. The pointcuts are expression for the aspect module to figure out the various points that would be otherwise scattered in the components; and the advices are instrumentations that need to be weaved into the base system by *automatically* composing the advices at points (i.e., *joinpoints*) that match with the pointcuts expression. One of the major advantages of AOP is that the scattered joinpoints are modularised by the pointcut expressions, thus reducing the complexity in code. This principle is also known as *Inversion Of Control (IOC)*.

AOP can reduce the complexity given that the base system cannot be easily disentangled and the joinpoints scattered among them as crosscuts. As one often sees, similar security vulnerability are often scattered in the code thus making them good candidates for joinpoints. By weaving the advices into these scattered places, AOP can help one harden the security in the design and implementation of the base system.

AOP with aspectJ There are several implementations of AOP, among which `aspectJ` for Java is the most widely used. To convey the basic concepts of AOP, and also to explain the example used in this paper, we illustrate the syntax and semantics of the AOP language using the following illustrative example.

Two Java classes “HelloWorld” and “GoodbyeWorld” serve as the original system which an aspect “HelloFromAspectJ.aj” implements an advice that instrument the original program to print an additional message “Hello from AspectJ” (see Figure 16).

In the aspectJ module, for example, the pointcut expression `mainMethod()` matches the *main* methods in both Java classes according to the interface signature of the method. The specification of the advice introduces the boolean pointcut expression by the keyword “after”. Because it matches with the two joinpoint methods in the Java classes, the statement in the body of the advice will be inserted *after* the invocation of *main* method. According to the semantics of the aspectJ language, one can also specify “before” and “around” advices. Namely, the *before* advice will be executed before the execution of the method at the joinpoint, and the *around* advice will be executed instead of the execution of the method at the joinpoint. Therefore, it is clear that AOP can completely change the behaviour of the original method, making it suitable to fix security vulnerability as opposed to the refactoring transformations.

5.2. Traceability under Evolution in the Presence of Aspects

The joinpoint model of AOP such as aspectJ is powerful: according to the specification an aspect specified in aspectJ can match with methods of classes. On the other hand, it does not include support for loops, super calls, throws clauses, multiple statements, etc. According to the literature [47], a joinpoint model at the method level has an important advantage over one at the lower statement level as it ensures modularity of the code. Using aspectJ, therefore, we cannot base our solution on a statement level joinpoint model. If one wants to alter the behaviour of a group of statements, a necessary step is to perform a refactoring operation such as *extract.method*.

Another issue is that when expressed in an aspect, the pointcuts must match with names in a particular library. If one does not change the function names or naming conventions used in the pointcut expressions, the aspects can be harder to reuse for a different library. In order to improve the reusability of such security aspects, we therefore abstract away the names from the implementation by substituting them with the corresponding symbolic name in the design model. These again require refactoring operations.

Therefore to exploit the refactoring traceability, we need to make sure that the traceability-preserving refactoring also preserves the aspect-oriented joinpoints used in that approach. Thus we need 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 the 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 instrumentations. 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 security 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 security aspects which otherwise would be non-reusable.

Since refactoring operations can improve the internal structures, these mappings can be performed selectively on the joinpoints that are made immediately useful for the aspects.

5.3. The SSL Case study: Fixing a Vulnerability in JESSIE

Since the places that need to get changed to fix a security vulnerability are often scattered across the code, it can be difficult and error-prone for humans to manually and consistently update the code.

We demonstrate how we use aspects for security hardening with an example from the JESSIE project.

In the JESSIE implementation, we found a significant security vulnerability as the certificate verification Veri(X509Cert_s) is not always invoked when the certificate message is received, which is an essential security check according to the protocol specification. It is needed because otherwise a man-in-the-middle attacker could insert a forged certificate containing his own public key into the communication and thereby decrypt the session key that is encrypted using that key, and thus eavesdrop on the encrypted communication in that session without being noticed by the communication partners. Therefore the current implementation of the SSL protocol in the JESSIE project does not enforce its security requirements. Below, we explain how this vulnerability arises and how one can use our approach to insert additional checks into the protocol implementation to harden its security.

Table 7 highlights the vulnerability by showing the execution log of four different test cases. In this table, the eight steps on the handshake protocol message sequence chart are shown by the rows. The second column shows the code corresponding to these steps that has been tested by the test cases. The third column highlights the differences in the instances of the four test cases.

If the certificate was checked at step S4, in Cases 3 and 4, the cheVal should report false in a correct implementation. However, we found they reported true instead.

Additional checks can be inserted into the protocol to harden its security. For example, using an aspect to crosscut every joinpoint of the program where a certificate is received, we found nothing is called by the program to check the issuing date. Therefore we find it is necessary to instrument the program with the functionality to check validity of the certificate against its date range issued by OpenSSL. Interestingly, this functionality was defined in JESSIE as a utility method checkValidity() in X509CertBridge.java. However it was never called, as indicated by a warning message in Eclipse.

Besides fixing the vulnerability by weaving an aspect into the refactored code we can also apply it to the implementation of the original program: After renaming checkValidity to cheVal, the aspect in Figure 17 is enabled to insert an additional check on the validity of certificate date (cheVal). Also, the refactored Veri is called right *after* a certificate is obtained through the pointcut expression certificate(). Without these refactoring operations, this aspect cannot be weaved through the original program.

This aspect whose design is derived from the protocol design model introduced earlier assumes the existence of a method for Veri. This method is created from the given

TABLE 7. Test cases for assessing security

Seq.	Tested Code	Example Test Case

S1	C = **new** ClientHello(P_pre, R_C,
	S_id, Ciph, Comp);
		Case1: ClientHello(TLSv1, clientRandom1,
		[B@b012a558, enabledSuites1, zlib)
		Case2−4: ClientHello(TLSv1, clientRandom2,
		[B@b01b0558, enabledSuites2, zlib)

S2	S.ServerHello(P_ver, R_S, S_id, Ciph, Comp);	Case1: ServerHello(TLSv1, serverRandom1,[B@b0134ed8, TLS_DHE_RSA_WITH_AES_256_CBC_SHA, zlib) Case2−4: ServerHello(TLSv1, serverRandom2,[B@b01baed8, TLS_DHE_DSS_WITH_AES_256_CBC_SHA, zlib)
S3	C.Certificate(X509Cert_s)	Case1−4: Certificate(serverCertificate)
S4	cheVal(D_notBefore, D_notAfter)	Case1,2: cheVal((107,2,2),(108,3,2))==True Case3: cheVal((107,2,1),(107,3,1))!=False Case4: cheVal((107,2,3),(107,3,1))!=False
S5	Ver_K_CA(Sig)	Case1−4: sigVerity((1.2.840.113549.1.1.5 Signature))
S6	clientKeyExchange(ckex)	Case1−4: ClientKeyExchange(ckex1)
S7	S.finished(md5_C, sha_C)	Case1−4: finished(gnu.java.security.hash.MD5@b00d27f8, gnu.java.security.hash.Sha160@b00d2f78)
S8	C.finished(verifyData)	Case1−4: finished("6a:df:3d:90:ec:0b:33:bc:2d:ce:ef:aa")

implementation by extracting 58 lines of code from the doClientHandshake method into a new public method Veri in the SSLSocket class. The extracted Veri method is then called in the advice to reimplement the already existing check. In addition to this check, we introduced an additional cheVal method into the aspect module. After weaving in this aspect, the date validity check is performed before the existing certificate check.

Using the test aspect, we were able to detect that the certificate() pointcut crosscuts three call sites with different argument settings (see accordingly the wildcard signature call(* C.certificate(..)) defined for this pointcut in our aspect definition above). One of them is without any arguments, whilst the other two are instantiated with arguments. From the execution log, we found all are executed after weaving our security aspect. However, if our aspect is not woven in (i.e. in the original JESSIE implementation), the original library only invokes the function of Veri when certificate is called without argument. In other words, the aspect has placed the check on all obtained certificates whilst the existing implementation misses some of them, which clearly results in a significant security vulnerability as explained earlier.

When weaving in the security aspect at the JSSE implementation, we could determine that it did not further harden the security for JSSE beyond the existing implementation since the security check implemented in the aspect is already correctly enforced in JSSE. This is confirmed by the logs of the two test cases that were reused. These test cases also helped us to verify that the messages are sent and received in a way that is consistent with the sequence diagram in Figure 6.

5.4. Continuous Integration

Continuous integration [48] has been adopted by our process where the regression test subprocess is augmented with the regressive refactoring: whenever code or model are changed in the repository – e.g., a developer committed a set of changes – the continuous integration script will check out the change set into a sandbox to conduct various automated builds and tests. Adding our refactoring scripts to the continuous integration script allows us to integrate our security assurance approach with the continuous integration framework. The error report subprocess is also augmented with an explanation of the counter-example of potential attack traces and the mismatch between the UMLsec model and the implementation code. Therefore we can incorporate a continuous integration process to make sure that whenever there is a change to the artefacts in the repository, a sequence of actions will be triggered to fully integrate the otherwise separate security tools.

For example, the usual compilation and function test steps are integrated with the additional actions in our proposed framework. Whenever there is a change in the design or in the implementation of the system, or there is a change to the refactoring scripts, the automated refactoring tool (ART) is called to check whether this causes the traceability links to be broken. If so, then the run-time verification tools will be

```
public aspect testCryptoProtocolSecurity {
  pointcut certificate():
      call(* Certificate.Certificate(..));
  Object around(): certificate() {
    X509Certificate[] X509Cert_s =
       (X509Certificate[]) proceed();
    if (! X509Cert_s.checked) {
     System.out.println("Problematic␣
         traceability␣found!");
    }
  }
}
public aspect CryptoProtocolSecurity {
  pointcut certificate():
      call(* Certificate.Certificate(..));
  Object around(): certificate() {
       X509Certificate[] X509Cert_s =
        (X509Certificate[]) proceed();
       SSLSocket s = (SSLSocket)
           thisJoinPoint.getThis();
       for (int m=0; m<pCs.length;m++) {
        assert cheVal(pCs[m].D_nb(),
                     pCs[m].D_na()):
             "+++␣The␣date␣is␣invalid␣+++";
       }
       s.Veri(X509Cert_s);
       return X509Cert_s;
  }
}
```

FIGURE 17. Aspect to check vulnerable certificates

invoked to check whether the new system still has correct traceability between design and implementation. If not, the developers will be informed to obtain a new refactoring script through further analysis.

The CruiseControl system is one of the most widely used continuous integration systems. A CI process in CruiseControl is driven by an XML-based build script for the Java-based build tool Apache Ant[2]. By default, the script would periodically monitor the designated repository for any changes. Then, based on the Ant build dependencies, these changes may trigger a sequence of actions, normally including building (compilation, packaging, deploying) and testing.

We extend the CruiseControl system by adding a few more tasks to the Ant build and test scripts. A daemon process on the build/test machine periodically monitors whether there is any change to the repository. Whenever changed artifacts (including the code, the model, the test cases, the refactoring scripts and the security aspects and assurance test cases) are committed, the event triggered a run of the extended Ant build.xml script, cf. the following example:

[2]http://ant.apache.org

```
<project name="jessie"
   default="test"
   basedir="jessie">
  <target name="build" depends="refactoring"/>
  <target name="test" depends="build"/>
   // the following tasks are augmented
  <target name="umlsec"/>
  <target name="refactoring"/>
  <target name="saspect" depends="test"/>
</project>
```

The first parameter specifies an environment variable for the Eclipse *headless* build process. Since our refactoring and aspect tools have dependencies on the basic Eclipse platform and JDT, in order to run the scripts for refactoring and security aspects it is necessary to start Eclipse without GUI.

The dependencies between the targets of the build.xml are straightforward. Before one can build the new system, the modified code must be refactored such that the changes committed by the programmers are synchronised with the model. The UMLsec security check for model vulnerabilities is performed after the system is built and the refactoring is done. Based on the UMLsec model and the LTL formulae to be monitored, an updated security monitor can now be generated automatically, if required by the system changes.

Integrating with the rest of the system through continuous integration, these aspects are thus reusable whenever a change to the design or the code does not affect the traceability.

6. RELATED WORK

6.1. Formal Security Verification and Model-based Security

Model-based Security [49] uses UML for the risk assessment of an e-commerce system within the CORAS framework for model-based security risk assessment. This framework is characterised by an integration of aspects from partly complementary risk assessment methods. [50] proposes an extension of the i*/Tropos requirements engineering framework to deal with security requirements. [51] shows how UML can be used to specify access control in an application and how one can then generate access control mechanisms from the specifications. The approach is based on role-based access control and gives additional support for specifying authorisation constraints. [52] presents the SECTET framework for Model Driven Security which is then specialised towards a domain-specific approach for healthcare scenarios, including the modelling of access control policies, a target architecture for their enforcement, and model-to-code transformations. [53] presents an approach for the transformation of security requirements to software architectures.

In an approach for model-based development of cryptographic protocols, [54] explains how to generate "provably correct" implementations from formal models.

Formally Verifying Cryptographic Protocol Implementations: There have recently been some approaches towards formally verifying implementations of cryptographic protocols against high-level security requirements such as secrecy, for example [18, 19, 20].

6.2. Security Traceability and Maintenance

Traceability and Model Synchronisation Software maintenance makes use of related models at different stages of development. Example models are goal trees for requirements, UML diagrams for design and source code for implementation. When some model elements change, it is necessary to *synchronise* the change on related elements in order to maintain model consistency [55]. Existing traceability approaches aim to recover traceability links that connect elements of certain software engineering artifacts in requirements, design and implementation [56, 57, 58, 59]. Search-based techniques recover traceability links between documents and code with a precision below 100% [56, 59]; a probability-model based approaches relies on a softgoal-interdependency graph to recover traceability links between functional and non-functional requirements [58]; a scenario-driven approach generates traceability links from observations of system executions [57]. Other work on requirements tracing includes [60]. In general, none of them can recover accurate requirements traceability links. Though efficient techniques have been proposed to account for incremental update of traceability links recovered from search-based approaches, these incrementally maintained traceability links are still inaccurate [59]. Graph transformation-based techniques [55] may accurately trace structural semantics, yet another mechanism is required to trace behavioural semantics.

Reverse Engineering Existing reverse engineering frameworks were proposed to improve accuracy of traceability for reference architecture [61] and for known design patterns [62]. In our previous work [63], refactoring was proposed to enable accurate abstraction of behavioural implementations such that they can be compared to the goal-oriented requirements. In this work, refactoring is not only used for comparing the source and target, but also for transforming the source into the target.

Refactoring Scripts Dig et al. [64] first studied the evolution of component APIs that can be replayed as refactoring steps. They argued that the refactoring of library components may indeed change the behaviour of the overall system especially when the client of the components are not refactored accordingly. For example, a function 'foo' may be renamed to 'bar' in the library, yet the call site of the function may still try to invoke 'foo', only to find broken contracts. Therefore, it is useful to keep track of (or detect in Dig's case) the refactoring steps as a script such that they can be replayed at the client side. Our tool supports tracking refactoring steps by translating the refactoring steps recorded by the IDE into change resilient refactoring specifications. Comparing with [64]'s work, our use of refactoring is not for replaying the changes, rather for maintaining the traceability between design elements and implementation regardless of changes. Though the RefactorCrawler tool [64] cannot be used directly, we can make use of the refactoring preview dialog code in the MolhadoRef tool [65].

Refactoring for Aspects In [66, 67], specialised refactoring actions are defined mainly for aspect-orientation. In this work, we expand the scope to any general-purpose refactoring steps supported by existing tools. We have exploited the opportunity to perform aspect-oriented instrumentation in order to harden the security that require general-purposed refactoring actions. In [68], Binkley et al. proposed a number of aspect-aware refactoring transformations to convert object-oriented programs into aspect-oriented ones. If the design element is implemented by crosscutting code, then Binkley et al.'s technique may be applied to our work to maintain the traceability between such elements. Since refactoring alone does not change the behaviour of the system, aspects derived from such refactoring transformations must not change the behaviour. Consequently, they cannot improve the security of existing implementation. In our work, we employ AOP to instrument the code with additional functionality to enforce security hardening. Therefore our aspect is introduced for a different purpose.

6.3. Run-time Verification for Traceability

In this work we employ run-time verification as a tool to trace security requirements not only to the source code level, but beyond to the level of the execution of code. There are various reasons as to why this is advantageous. For example, assumptions that are inherent in design models may not adequately address real-world challenges, such as assumptions about attacker behaviour or the correctness of an implementation. Run-time verification as used in Section 3 has become a popular tool to verify that a system's execution adheres to a set of predefined properties.

As far as we know, this is the first work in which run-time verification is used for the traceability of high-level security properties in evolving systems.

Work in the area of run-time verification such as [69, 70, 10] consider it foremost from a theoretical point of view; that is, the complexity of the underlying problems, the theoretical expressiveness of the formalism used to express monitoring properties, or the efficiency of the generated monitors. In contrast, we focus on the methodological aspects of this technique for achieving traceable security beyond the source-code level.

As such, there are two aspects to be considered:

(1) the use of run-time verification for traceability of security properties in evolving systems, and
(2) the evolution of the run-time verification "layer" itself in terms of changing properties, monitor code, etc.

Regarding 1), although there seems to be no prior work on run-time security verification for evolving systems, there is some previous work on run-time security verification. The techniques used in run-time verification bear a resemblance with the well-known *security automata* as introduced by Schneider [35]. Formally, Schneider's work is based on temporal logic as well, however, it imposes restrictions on the types of formulae which can be monitored or "enforced", to use the terminology of [35]. Security automata are restricted to the so-called *safety fragment* (of LTL) and do not impose an explicit acceptance condition; that is, a trace (finite or infinite) is accepted as long as there exists a corresponding run on it in the automaton. When a security automaton is used to monitor a safety property such as $\mathbf{G}a$ ("always a"), then this semantics, arguably, is the desired one; that is, the automaton would yield "accept" as long as a is observed and "reject", otherwise. In our application, however, we encountered properties that are expressed, for example, using (sub-) formulae of the kind $a\mathbf{U}b$ ("a until b"), for which, arguably, a more fine grained distinction between observations is desirable. For these types of formulae, our monitors would yield ? if the first observation contained an a (and not b), and "accept" if the first observation contained a b. Disregarding the fact that this formula is not a safety formula, and as such by definition not enforceable using security automata, a security automaton—would it be, nonetheless, applied—would yield "accept" in both cases, i.e., if the first observation contained an a (and not b) and if the first observation contained only b. In other words, it would not distinguish whether or not all future observations will indeed be satisfying the formula, or whether or not it is still possible to violate the formulae in the future. Our monitors do give the user this type of fine grained feedback for the properties we identified relevant to our application of monitoring cryptographic protocols.

However, it is worth mentioning that some properties which are not safety are, in fact, monitorable using security automata in a way not anticipated in [35]; that is, if the properties belong to a complementary class of properties, the so-called *co-safety* properties. Then, a security automaton could be built for the negated formula (say, $\neg\varphi$), and the monitor's result be inverted by the user; that is, a reported violation would then, in fact, signal that the system has satisfied the original co-safety property, φ. However, since not all properties are divisible into safety and co-safety properties, this method is not generally applicable. For example, consider Property 2 of Table 5, which is neither a safety nor a co-safety property, yet we were able to build a 3-valued monitor for it using our methods.

Another application of monitoring to security was presented in [71]. The paper proposes a caller-side rewriting algorithm for the byte-code of the .NET virtual machine where security checks are inserted around calls to security-relevant methods. The work is different from ours in that it has not been applied to the security verification of cryptographic protocols, which pose specific challenges (such as the correct use of cryptographic functions and checks). In another approach, [72] proposes to use formal patterns of LTL formulae that formalise frequently reoccurring system requirements as *security monitoring patterns*. Again, this does not seem to have been applied to cryptographic protocols so far.

Regarding 2), an important step regarding evolvable systems was recently also made by Barringer et al. in [73]. However, their view on evolution differs from the one presented in this paper. Notably, their approach to run-time verification is not just passive, but active, in that a failing system is modified by a monitor noticing the failure. As such, the failing system evolves, and the monitors continuously adapt. In contrast, the evolution of our systems is sparked by comparably major changes in the software's implementation, e.g., triggered by new requirements that warrant a new release of a system, or specific rewrites for efficiency gains. As a consequence, our use of run-time verification is not as tightly integrated as that presented in [73], formally and practically.

7. CONCLUSIONS

We have used 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 authentication. An implementation of the specification can then be verified against the model by making use of run-time verification. Using the approach to run-time security verification, one can raise an alarm at run-time in case of a security violation, and terminate the given protocol execution, before the secret is leaked out to the network. We also explained how to remove the security vulnerability in a implementation that has been detected in this way to make sure the same problem will not appear again, making use of techniques from aspect-oriented programming (AOP).

Despite the similarities between testing and run-time verification, run-time verification can provide a level of assurance that goes beyond what testing can usually achieve when applied to highly complex security-critical software: While testing complex systems can usually not be exhaustive, run-time verification ensures, by construction, that every system trace that will ever be executed will be verified – while it is executed. In the case of the cryptographic protocols that we consider, it is indeed sufficient to notice attempted security violations at run-time to still be able to maintain the security of the system: The monitor is constructed in such a way that, if it detects a violation, the current execution of the security protocol will be terminated before any secret information is leaked out on the network.

In practice, systems do not remain unchanged after they are being used but may evolve over their life-time. We have therefore enabled our security assurance approach to cope automatically with the fact that systems will evolve at run-time, and still provide valid run-time security assurance.

We demonstrated the approach at the hand of an application to the Java-based implementation JESSIE of the Internet security protocol SSL, in which a security weakness was detected and fixed using our approach. We also explained how the traceability link can be transformed to the official implementation of the Java Secure Sockets Extension (JSSE) that was recently made open source by Sun.

There are a number of possible directions for future work.

- 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 how to combine run-time security verification with static compositional software verification such as [74].
- In another direction, it would be interesting to see whether it would be possible to expand the kinds of attacks that could be detected by this approach, for example by including weaknesses in the implementations of cryptographic algorithms (such as encryption and digital signature). It remains, however, to be seen which impact this would have on the performance of the monitors.
- The current monitoring approach relies on the assumption of having access to the source code of the monitored software. It would be interesting to see whether one could develop a monitor approach that does not rely on this assumption but is still sufficiently precise and performant.

Acknowledgements Discussions with Martin Leucker about a draft of this paper are gratefully acknowledged, as well as constructive comments by the reviewers which helped improving the presentation significantly.

REFERENCES

[1] Jürjens, J. (2004) *Secure Systems Development with UML*. Springer-Verlag, Berlin.

[2] Jürjens, J. (2005) Sound methods and effective tools for model-based security engineering with UML. *Intl. Conference on Software Engineering (ICSE)*, pp. 322–331. ACM Press, New York, NY.

[3] Jürjens, J. and Yu, Y. (2007) Tools for model-based security engineering: Models vs. code. *22nd Intl. Conference on Automated Software Engineering (ASE)*, pp. 545–546. ACM Press, New York, NY.

[4] Jürjens, J. and Yu, Y. (2001-09). Security analysis tools. http://computing-research.open.ac.uk/jj/sectracetool.

[5] Best, B., Jürjens, J., and Nuseibeh., B. (2007) Model-based security engineering of distributed information systems using UMLsec. *Intl. Conference on Software Engineering (ICSE)*, pp. 581–590. ACM Press, New York, NY.

[6] Jürjens, J., Schreck, J., and Bartmann, P. (2008) Model-based security analysis for mobile communications. *Intl. Conference on Software Engineering (ICSE)*, pp. 683–692. ACM Press, New York, NY.

[7] Colin, S. and Mariani, L. (2004) Run-time verification. *Model-Based Testing of Reactive Systems*, Lecture Notes in Computer Science, **3472**, pp. 525–555. Springer-Verlag, Berlin.

[8] Bauer, A. and Jürjens, J. (2008) Security protocols, properties, and their monitoring. *4th Int. Workshop on Software Engineering for Secure Systems (SESS)*, pp. 33–40. ACM Press, New York, NY.

[9] Leucker, M. and Schallhart, C. (2009) A brief account of runtime verification. *J. Log. Algebr. Program.*, **78**, 293–303.

[10] Bauer, A., Leucker, M., and Schallhart, C. (2006) Monitoring of real-time properties. *26th Intl. Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Lecture Notes in Computer Science, **4337**, pp. 261–273. Springer-Verlag, Berlin.

[11] Bauer, A., Leucker, M., and Schallhart, C. (2009). LTL$_3$ Tools. http://ltl3tools.SourceForge.Net/.

[12] Jürjens, J., Yu, Y., and Bauer, A. (2008) Tools for traceable security verification. *Proceedings of the BCS International Academic Conference 2008—Visions of Computer Science*, Swindon, UK, pp. 367–378. The British Computer Society.

[13] Hoare, C. (1996) How did software get so reliable without proof? *Formal Methods Europe (FME'96)*, Lecture Notes in Computer Science, **1051**, pp. 1–17. Springer-Verlag, Berlin.

[14] Calder, M. (1998) What use are formal design and analysis methods to telecommunications services? *5th Intl. Conference on Feature Interactions in Telecommunications and Software Systems*, pp. 23–31. IOS Press.

[15] Thomas, M. (2004) Engineering judgement. *9th Australian workshop on Safety critical systems and software (SCS)*, pp. 43–47. Australian Computer Society, Inc.

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

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

[18] Jürjens, J. and Yampolskiy, M. (2005) Code security analysis with assertions. *20th Intl. Conference on Automated Software Engineering (ASE)*, pp. 392–395. ACM Press, New York, NY.

[19] Goubault-Larrecq, J. and Parrennes, F. (2005) Cryptographic protocol analysis on real C code. *6th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, Lecture Notes in Computer Science, **3385**, pp. 363–379. Springer-Verlag, Berlin.

[20] Bhargavan, K., Fournet, C., Gordon, A., and Tse, S. (2006) Verified interoperable implementations of security protocols. *Computer Security Foundations Workshop*, pp. 139–152. IEEE Computer Society.

[21] Breu, R., Burger, K., Hafner, M., Jürjens, J., Popp, G., Wimmel, G., and Lotz, V. (2003) Key issues of a formally based process model for security engineering. *16th Intl. Conference "Software & Systems Engineering & their Applications" (ICSSEA)*. IEEE Computer Society.

[22] Jürjens, J. and Shabalin, P. (2004) Automated verification of UMLsec models for security requirements. *The Unified Modeling Language (UML)*, Lecture Notes in Computer Science, **2460**, pp. 412–425. Springer-Verlag, Berlin.

[23] Borland, Inc (2009). Borland Together. http://www.borland.com/us/products/together/.

[24] Jürjens, J. (2000) Secure information flow for concurrent processes. *11th Intl. Conference on Concurrency Theory (CONCUR)*, Lecture Notes in Computer Science, **1877**, pp. 395–409. Springer-Verlag, Berlin.

[25] Jürjens, J. (2002) Formal semantics for interacting UML subsystems. *5th Intl. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pp. 29–44. International Federation for Information Processing (IFIP) Kluwer Academic Publishers.

[26] Stenz, G. and Wolf, A. (2000) E-setheo: An automated theorem prover. *Intl. Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, Lecture Notes in Computer Science, **1847**, pp. 436–440. Springer-Verlag, Berlin.

[27] Abadi, M. and Needham, R. (1996) Prudent engineering practice for cryptographic protocols. *IEEE Trans. on Software Engineering*, **22**, 6–15.

[28] Pnueli, A. (1977) The temporal logic of programs. *18th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pp. 46–57. IEEE Computer Society.

[29] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999) *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[30] Holzmann, G. J. (1991) *Design and validation of computer protocols*. Prentice-Hall, Inc.

[31] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. (2001) Automatic predicate abstraction of C programs. *SIGPLAN Not.*, **36**, 203–213.

[32] Godefroid, P. (2005) Software model checking: The verisoft approach. *Form. Methods Syst. Des.*, **26**, 77–101.

[33] Aho, A. V., Sethi, R., and Ullman, J. D. (1988) *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

[34] Eisner, C. and Fisman, D. (2006) *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer New York, Inc.

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

[36] Stärk, R., Schmid, J., and Börger, E. (2001) *Java and the Java virtual machine – definition, verification, validation*. Springer-Verlag.

[37] Hopcroft, J. E. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages and Computation*, first edition. Addison-Wesley.

[38] Mens, T. and Tourwe, T. (2004) A survey of software refactoring. *IEEE Trans. on Software Engineering*, **30**, 126–139.

[39] Widmer, T. (2007). Unleashing the power of refactoring. http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring.

[40] Krueger, C. W. (1992) Software reuse. *ACM Comput. Surv.*, **24**, 131–183.

[41] Chess, B. and West, J. (2007) *Secure Programming with Static Analysis*. Addison-Wesley Professional.

[42] Bauer, A., Leucker, M., and Streit, J. (2006) SALT—Structured Assertion Language for Temporal logic. *Eighth Intl. Conference on Formal Engineering Methods (ICFEM)*, Lecture Notes in Computer Science, **4260**, pp. 757–776. Springer-Verlag, Berlin.

[43] Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M. Y., and Zbar, Y. (2002) The ForSpec temporal logic: A new temporal property-specification language. *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, **2280**, pp. 296–211. Springer-Verlag, Berlin.

[44] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997) Aspect-oriented programming. *17th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, **1241**, pp. 220–242. Springer-Verlag, Berlin.

[45] Tarr, P., Ossher, H., Harrison, W., and Jr., S. S. (1999) Degrees of separation: Multi-dimensional separation of concerns. *Intl. Conference on Software Engineering (ICSE)*, pp. 107–119. ACM Press, New York, NY.

[46] Yu, Y., do Prado Leite, J. C. S., and Mylopoulos, J. (2004) From goals to aspects: Discovering aspects from requirements goal models. *12th IEEE Intl. Conference on Requirements Engineering (RE)*, pp. 38–47. IEEE Computer Society.

[47] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001) An overview of AspectJ. *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. Springer-Verlag, Berlin.

[48] Duvall, P., Matyas, S., and Glover, A. (2007) *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional.

[49] Dimitrakos, T., Ritchie, B., Raptis, D., Aagedal, J., den Braber, F., Stølen, K., and Houmb, S. (2002) Integrating model-based security risk management into ebusiness systems development: The CORAS approach. *Second IFIP Conference on E-Commerce, E-Business, E-Government (I3E)*, pp. 159–175. Kluwer Academic Publishers.

[50] Giorgini, P., Massacci, F., and Mylopoulos, J. (2003) Requirement engineering meets security: A case study on modelling secure electronic transactions by VISA and Mastercard. *22nd Intl. Conference on Conceptual Modeling (ER)*, Lecture Notes in Computer Science, **2813**, pp. 263–276. Springer-Verlag, Berlin.

[51] Basin, D., Doser, J., and Lodderstedt, T. (2006) Model driven security: From UML models to access control infrastructures. *ACM Trans. on Software Engineering and Methodology*, **15**, 39–91.

[52] Alam, M., Hafner, M., and Breu, R. (2007) Model-driven security engineering for trust management in SECTET. *Journal of Software*, **2**, 47–59.

[53] Yskout, K., Scandariato, R., Win, B. D., and Joosen, W. (2008) Transforming security requirements into architecture. *3rd Intl. Conference on Availability, Reliability and Security (ARES)*, pp. 1421–1428. IEEE Computer Society.

[54] Pironti, A. and Sisto, R. (2007) An experiment in interoperable cryptographic protocol implementation using automatic code generation. *IEEE Symposium on Computers and Communications*, pp. 839–844. IEEE Computer Society.

[55] Ivkovic, I. and Kontogiannis, K. (2004) Tracing evolution changes of software artifacts through model synchronization. *20th IEEE Intl. Conference on Software Maintenance (ICSM)*, pp. 252–261. IEEE Computer Society.

[56] Antoniol, G., Canfora, G., Casazza, G., de Lucia, A., and Merlo, E. (2002) Recovering traceability links between code and documentation. *IEEE Trans. on Software Engineering*, **28**, 970–983.

[57] Egyed, A. (2003) A scenario-driven approach to trace dependency analysis. *IEEE Trans. on Software Engineering*, **9**, 116–132.

[58] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., and Christina, S. (2005) Goal-centric traceability for managing non-functional requirements. *Intl. Conference on Software Engineering (ICSE)*, pp. 362–371. ACM Press, New York, NY.

[59] Jiang, H., Nguyen, T. N., and Chen, I. (2008) Incremental latent semantic indexing for effective, automatic traceability link evolution management. *Intl. Conference on Software Engineering (ICSE)*, pp. 59–68. ACM Press, New York, NY.

[60] Spanoudakis, G., Zisman, A., Pérez-Miñana, E., and Krause, P. (2004) Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, **72**, 105–127.

[61] Murphy, G. C., Notkin, D., and Sullivan, K. J. (2001) Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. on Software Engineering*, **27**, 364–380.

[62] Beyer, D., Noack, A., and Lewerentz, C. (2005) Efficient Relational Calculation for Software Analysis. *IEEE Trans. on Software Engineering*, **31**, 137–149.

[63] Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., and do Prado Leite, J. C. S. (2005) Reverse engineering goal models from legacy code. *13th IEEE Intl. Conference on Requirements Engineering (RE)*, pp. 363–372. IEEE Computer Society.

[64] Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006) Automated detection of refactorings in evolving components. *20th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, **4067**, pp. 404–428. Springer-Verlag, Berlin.

[65] Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. N. (2007) Refactoring-aware configuration management for object-oriented programs. *Intl. Conference on*

Software Engineering (ICSE), pp. 427–436. ACM Press, New York, NY.

[66] Hannemann, J. (2006) Role-based refactoring of crosscutting concerns. PhD thesis Vancouver, BC, Canada.

[67] Laddad, R. (2006) *Aspect Oriented Refactoring*. Addison-Wesley Professional.

[68] Binkley, D., Ceccato, M., Harman, M., Ricca, F., and Tonella, P. (2006) Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Trans. on Software Engineering*, **32**, 698–717.

[69] Havelund, K. and Rosu, G. (2004) Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, **6**, 158– 173.

[70] Havelund, K. and Rosu, G. (2002) Synthesizing Monitors for Safety Properties. *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, **2280**, pp. 342–356. Springer-Verlag, Berlin.

[71] Vanoverberghe, D. and Piessens, F. (2008) A caller-side inline reference monitor for an object-oriented intermediate language. *10 Intl. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science, **5051**, pp. 240–258. Springer-Verlag, Berlin.

[72] Spanoudakis, G., Kloukinas, C., and Androutsopoulos, K. (2007) Towards security monitoring patterns. *ACM Symposium on Applied Computing (SAC)*, pp. 1518–1525. ACM Press, New York, NY.

[73] Barringer, H., Gabbay, D. M., and Rydeheard, D. E. (2007) From runtime verification to evolvable systems. In Sokolsky, O. and Tasiran, S. (eds.), *Intl. Workshop on Runtime Verification*, Lecture Notes in Computer Science, **4839**, pp. 97–110. Springer-Verlag, Berlin.

[74] Abramsky, S., Ghica, D., Murawski, A., and Ong, C.-H. (2004) Applying game semantics to compositional software modeling and verification. *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 421–435. Springer-Verlag, Berlin.