Security Analysis of a Biometric Authentication System Using UMLsec and JML*

John Lloyd[1] and Jan Jürjens[2]

[1] Atos Origin UK
[2] Open University (UK) and Microsoft Research, Cambridge
john.lloyd@atosorigin.com
http://jurjens.de/jan

Abstract. Quality assurance for security-critical systems is particularly challenging: many systems are developed, deployed, and used that do not satisfy their security requirements. A number of software engineering approaches have been developed over the last few years to address this challenge, both in the context of model-level and code-level security assurance. However, there is little experience so far in using these approaches in an industrial context, the challenges and benefits involved and the relative advantages and disadvantages of different approaches. This paper reports on experiences from a practical application of two of these security assurance approaches. As a representative of model-based security analysis, we considered the UMLsec approach and we investigated the JML annotation language as a representative of a code-level assurance approach. We applied both approaches to the development and security analysis of a biometric authentication system and performed a comparative evaluation based on our experiences.

Keywords: Security analysis, JML, UMLsec, biometric authentication.

1 Introduction

Designing and verifying security-critical software is very difficult because of the complexity of these mechanisms and of industrial systems, the interaction with adversaries and the ways in which systems use security mechanisms. The traditional industrial approach is to describe security requirements textually, design security features as an add-on to the system design, then patch flaws found or exposed after deployment - called 'penetrate and patch'. However this approach is lengthy, imprecise and difficult to check, which risks leaving security vulnerabilities that might result in major loss or damage before discovery. Removing vulnerabilities found in operational systems can be complex, time consuming and error-prone.

Researchers have developed formal methods for specifying a system in such a way that properties of the specification can be mathematically verified for correctness. However formal methods have not been widely adopted because the complexity of

* Empirical results category paper.

A. Schürr and B. Selic (Eds.): MODELS 2009, LNCS 5795, pp. 77–91, 2009.

both the languages and the systems modeled mean that staff need considerable skill, training and time to produce specifications and proofs, which incurs high costs.

To increase industry acceptance of formal methods, an approach is needed that integrates security requirements specification and verification with a method and language that can be used by general software designers. The UMLsec approach [1] addresses this by expressing security requirements within a system specification using the Unified Modeling Language (UML) and then using analysis tools to verify these requirements for correctness.

UMLsec is an approach based on a formal foundation but it aims to be easier for general software designers to use than traditional formal methods because the UMLsec language is simpler and integrated with UML. This aims to reduce the very high training and usage costs that have marginalised the use of formal methods. This approach also aims to be less expensive than the traditional 'penetrate and patch' approach as it could be applied at an early stage in system development when the cost of change would be low. The practical application reported in this paper therefore evaluated how easy it was to specify security requirements using UMLsec and how difficult it then was to implement a system from this UMLsec specification.

Even though the security requirements specification might have been verified, security flaws may be introduced during the design and implementation of the system or in subsequent changes. We therefore investigated using the Java Modeling Language (JML) to relate the implemented system back to its UMLsec security specification and verify that it is correct in relation to this specification.

We investigated these approaches by implementing a biometric authentication system adapted from a proposal by Viti and Bistarelli [2]). Biometrics are an attractive security mechanism for authentication because they are an inherent feature of a person and so cannot be lost or easily changed. We used this practical application to evaluate the model-level UMLsec security assurance approach and the code level JML assurance approach. We here compare their advantages and disadvantages, describe specific practical experiences with the two approaches and their combination in particular, discuss lessons learned and suggest possible improvements.

Although there is traditionally a lack of practical validation of software engineering research, it is increasingly being realised that this is an important component of such research, leading to increasing activity in this area. The work presented here is different from earlier reports of industrial applications of model-based security in that it focuses on security analysis on the code rather than the model level.

In the next section, we summarise the UMLsec and JML approaches. We then give an overview of the biometric authentication system in Section 3. Our security assurance using a combination of UMLsec and JML is described in Section 4. We cover lessons learned from the UMLsec and JML approaches in Section 5 and we end with a summary and suggestions for further work.

2 Software Security Assurance

In this section, we first discuss the practical challenges involved in providing assurance for security critical software and we then summarise the two approaches in our application: UMLsec for the model-level assurance and JML for the code-level assurance.

2.1 Challenges for Security Assurance

In practice, security is compromised most often not by breaking dedicated mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being used. Thus it is not enough to ensure the correct functioning of security mechanisms used. They cannot be 'blindly' inserted into a security-critical system, but the overall system development must take security aspects into account in a coherent way. While functional requirements are generally analyzed carefully in systems development, security considerations often arise after the fact. Adding security as an afterthought, however, often leads to problems and security engineers get little feedback about the secure functioning of their products in practice, since security violations are often kept secret for fear of harming a company's reputation.

In practice, the traditional strategy for security assurance has been 'penetrate and patch'. It has been accepted that deployed systems contain vulnerabilities: whenever a penetration of the system is noticed and the exploited weakness can be identified, the vulnerability is removed. For many systems, this approach is not ideal: each penetration may already have caused significant damage before the vulnerability can be removed. For systems that offer strong incentives for attack, such as financial applications, the prospect of being able to exploit a weakness even only once may be enough motivation to search for such a weakness. System administrators are often hesitant to apply patches because of disruption to the service. Having to create and distribute patches costs money and leads to loss of customer confidence. It would thus be preferable to consider security aspects more seriously in earlier phases of the system lifecycle, before a system is deployed, or even implemented, because late correction of requirements errors can be significantly more expensive than early correction.

2.2 Model-Based Security Using UMLsec

UMLsec is an extension of UML for secure systems development. Recurring security requirements, such as *secrecy*, *integrity*, and *authenticity* are offered as specification elements by the UMLsec extension. These properties and its associated semantics are used to evaluate UML diagrams of various kinds and indicate possible security vulnerabilities. One can thus verify that the desired security requirements, if fulfilled, enforce a given security objective. One can also ensure that the requirements are actually met by the given UML specification of the system (design solution). UMLsec encapsulates knowledge on prudent security engineering and thereby makes it available to developers who may not be experts in security. The extension is given in the form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate security requirements and assumptions on the system environment. *Constraints* give criteria that determine whether the requirements are met by the system design (design solution), by referring to a precise semantics mentioned below.

The tags defined in UMLsec represent a set of desired properties. For instance, "freshness" of a value means that an attacker cannot guess what its value was. Moreover, to represent a profile of rules that formalize the security requirements, the following are some of the stereotypes that are used: *«critical»*, *«high»*, *«integrity»*, *«internet»*, *«encrypted»*, *«LAN»*, *«secrecy»*, and *«secure links»*. The definition of the

stereotypes allows for model checking and tool support. As an example consider *«secure links»*. This stereotype is used to ensure that security requirements on the communication are met by the physical layer: when attached to a UML subsystem, the constraint enforces that for each dependency *d* with a stereotype s representing a certain security requirement, the physical architecture of the system should support that security requirement at the given communication link.

A detailed explanation of the tags and stereotypes defined in UMLsec can be found in [1]. The extension has been developed based on experiences on the model-based development of security-critical systems in industrial projects involving German government agencies and major banks, insurance companies, smart card and car manufacturers, and other companies. There have been several applications of UMLsec in industrial development projects [3,4,5]. UMLsec supports automatic verification of security properties using the UMLsec tool-support [1] as well as IT security risk assessment [6].

2.3 The Java Modeling Language

The JML is a behavioural interface specification language which describes method pre- and post-conditions. It is based on the design-by-contract approach [7] that describes a requirement for a client to guarantee that agreed pre-conditions hold before calling a method defined by a class; in return, the class guarantees that agreed post-conditions will hold after the call. For example, a client calling a square root function would guarantee that the argument was a positive number and the class would guarantee that the result was approximately equal to the square root of the argument.

[8], [9] describe the basic features of the language and these papers are complemented by the reference manual [10]. JML is written as annotation comments beginning with `//@` within Java code. A `requires` clause specifies the method's pre-condition and ensures clause specifies normal and exception post-conditions. For example, the annotation `//@ requires x>= 0.0` specifies the pre-condition of the square root function. The client code calling this function must then ensure that this function is only called with a positive number argument. As the contracts expressed in JML are compiled into executable code, any run-time violation of them can be immediately detected, such as a negative argument in this function call.

3 The Biometric Authentication System

3.1 Context

The implemented biometric authentication system consists of a controlling PC with a combined scanner/smart card reader. The PC is connected to a server application that authenticates the user. The user firstly inserts a smart card containing a fingerprint biometric template into the USB port of the smart card reader. He then enters a PIN at the controlling host PC to activate the card, and places a finger on the scanner. The host then compares the scan to the fingerprint template; if they match, the host encrypts a nonce sent from the server with the user's private key stored on the smart card and returns it to the server. The server authenticates the user by decrypting the received nonce with the user's public key and confirming that it matches the nonce

originally sent. This then completes user authentication by three factors: possession of a smart card; a PIN; and a biometric.

3.2 Specification Modeling Using UMLsec

The host-smart card and host–scanner protocols incorporate protocol fragments described in [11], [12]. The host establishes a shared symmetric key with the smart card and scanner which is then used to encrypt messages over the un-trusted connection between them. The system uses misuse counters to limit the numbers of PIN entries, biometric scans and server connection attempts.

The server connection uses the HTTPS protocol (HTTP over a Secure Sockets Layer (SSL)) to authenticate the server using its digital certificate and request client authentication using the user's digital certificate stored on the smart card.

We specified requirements using UML deployment, class and sequence diagrams. The deployment diagram describes the physical hardware and connections in the system. The class diagram describes the four main classes modeling the smart card, host PC, scanner and server in terms of their attributes and operations, plus secrecy, integrity, freshness and authenticity security requirements described using UMLsec. The sequence diagrams describe the interaction with the system user and the protocol of messages between the host and the smart card, scanner and the server. These protocols use cryptographic data and functions, and again we use UMLsec to express security requirements.

The system's software architecture is a software-only implementation of the host, scanner and smart card classes where the host is invoked by a user through a browser. Each message in the protocols between these classes is constructed, passed as an argument in a method call to the receiving object and then processed. The object's reply is handled as a method return value to retain the host's overall control of the dialogue.

The system uses the Java Cryptography Architecture (JCA) to encrypt and decrypt message data, the Java security package to implement nonces, message authentication codes and digital signatures, and the Java Secure Sockets Extension (JSSE) to provide the HTTPS server connection.

4 Security Assurance

4.1 JML Contracts

We implemented the software components of the system using about 1,300 lines of Java 6 code. We then used JML to try to verify parts of the system's code against its UMLsec specification. We had described system operations using the Object Constraint language (OCL) that is part of the UML notation, but we did not use it to formally describe all system constraints because we found OCL difficult to use for this task due to its complexity. This did not cause any development problems in our particular application because the designer and developer were the same person, but in a larger-scale project, where these roles are usually separate, more formal constraint documentation would be needed to communicate information. Also, an earlier decision to use JML at the design modeling stage would have helped structure the code

during the implementation phase to increase the effectiveness of using JML. In particular, an early decision on the implementation language would allow a specific specification language like JML to be used rather than the generic OCL.

As an example of a method-level JML contract that we added, we now consider a method that compares the Message Authentication Code (MAC) calculated from a received message with the MAC in the message. The method handles different message types with different lengths. It was difficult to write a JML contract since neither the message type nor the return code were expressed as arguments as the method was an addition during implementation to consolidate a number of blocks of similar code. We eventually verified the check in JML by comparing each byte of the two MACs using a JML `\forall` expression:

```
//@ ensures (\forall int x; 0 <= x &&
//@  x <  mac.getMacLength();
//@  calcMac[x].compareTo(receivedMac[x])==0);
```

This example demonstrates that JML can be difficult to apply unless the method has been designed with JML verification in mind. A similar comment would apply when using another notation instead of JML (such as OCL). This is also in line with the intuitive idea that security contracts can be easier to specify if we can take advantage of the algebraic properties of the data involved.

We also used the finite state machine model proposed in [13] to verify that messages are sent in the correct sequence by the host. For this, static integers are declared defining the messages sent before the start and end of each method, thus:

```
//@ public static final ghost int
//@ INITIAL = 1,
//@ RESET = 2,
//@ ASKZ = 3,
//@ public ghost int state = INITIAL;
```

The JML ghost field state is assigned one of these values at the end of each method, for example: `//@ set state = ASKZ`.

The JML contract for the method can then test the relevant pre- and post-condition using this variable. For example, for a method which must begin after an `ASKZ` has been set and must end by sending an `ACK`, we defined the following:

```
/*@ public normal_behavior
 @       requires state == ASKZ;
 @       assignable \everything;
 @       ensures state == ACK;
 @*/
```

However, since each method handles several messages, we could not use JML to fully verify the correct message sequence, since JML can only check the state before the start and after termination of each method. Had these methods been written at a lower level of granularity to each handle only one message exchange then this would have been possible. We refactored several methods to confirm this. This example demonstrates that the software to be verified using JML should be designed in a style consistent with the use of JML to gain most value from its use.

The JML `\fresh` expression asserts that objects are freshly allocated and were not allocated in the pre-state. This contributes to the freshness security requirements implemented by nonces in the system in a useful way. For example:

```
//@ ensures \fresh (zSc);
```

verifies that the object `zSc` that stores a random number is freshly generated. Note that this check does not aim to guarantee that the pseudo-random algorithm used to generate the random number is itself secure. It does however make sure that an existing random number object is not reused accidentally or maliciously, which does prevent a certain class of fresh value security flaws.

We investigated using JML to verify that a message has been encrypted. There is no JML expression to support this directly and the cipher Java class does not have a method to return the encrypted or decrypted state to which it has been set. We could have rewritten the encryption code as a separate method and added an attribute that is set to the current cipher state; a JML or Java `assert` statement could then test this attribute value when each message is sent. However this would not directly test the encryption of the message. We therefore chose to test that the cipher text was different to the plain text by using a JML assert statement of the form:

```
//@ assert (\forall int x; 0 <= x &&
//@       x < plainText.length();
//@       plainText[x] != cipherText[x]);
```

after each message send. Again, this check does not aim to enforce, for example, that the used encryption algorithm is secure, but does make sure that the application of the encryption algorithm is not simply left out accidentally or maliciously.

Note that this example shows that it is important to distinguish the kind of property we would like to specify from what we can specify in a verifiable manner: The intended property refers to how hard it is for someone who does not have access to the encryption key to retrieve the original message. This is a property that cannot be expressed directly as a pre/post/invariant condition.

We also intended to specify in JML that methods called from outside the smart card methods could not read or assign to the field containing the PIN. This turned out to be relatively cumbersome because there is no JML keyword with the direct meaning ‘not accessible’. Instead, one needs to add a JML to every method to specify all the fields that are accessible by each method.

4.2 JML Specification Patterns for Security

Warnier [14] proposes JML specification patterns for confidentiality and integrity. As part of our application, we investigated their usefulness for identifying security flaws. Warnier defines confidentiality as non-interference between variables of different security levels: the values of all non-confidential (low security) fields in the post state should be independent of the values of all confidential (high-security) fields in the pre-state. He defines a similar JML specification pattern for integrity where the values of high security variables in the post-state are required to be independent of low security variable values.

Although using specification patterns is a good idea in principle, in our experience, they were difficult to apply because many of the system's methods were rather large and therefore too complex to investigate using these patterns; they would have been more useful when applied to smaller methods. Also, there was a difficulty in applying the confidentiality pattern in cryptography which is common to information-flow type definitions of confidentiality: given a high-security plain-text, its encryption would be considered low-security (because it can only be decrypted by trusted parties, so the cipher text can be communicated publically). Thus, the encryption function itself would be considered to violate confidentiality according to this definition, although that is clearly not the case in reality.

4.3 Security Specifications in JML

Agarwal *et al.* [13] have written JML specifications for some security-relevant Java classes, although only a few are cryptography classes. These specifications aim to provide a more precise understanding of the behaviour of the classes than *javadoc* comments. This might reduce security flaws caused by using these classes incorrectly, for example with invalid pre-conditions or handling post-conditions incorrectly, and it might identify such errors during run-time assertion checking.

To investigate this potential, we compared the JML specifications of four methods of the `Signature` Java class used to support digital signature processing in the system to the *javadoc* comments for these methods. Overall, the JML specifications were somewhat more precise than *javadoc* but they required significantly more time to understand. Also, only three cryptography classes currently have JML specifications. The ability to automatically check JML contracts at run-time checking is of limited use for these specifications since nearly all pre-conditions are also handled by Java exception handling. The post-conditions could be relied on as the Java methods are rather unlikely to contain errors, being part of a standard library implementations that has been extensively tested and used already. We therefore found JML specifications for Java classes to be mainly a useful supplement to *javadoc* comments, which are not always completely precise and unambiguous.

4.4 Manual UMLsec / Code Validation Check

We manually checked the consistency of each UMLsec security requirement with its implementation in the prototype to examine whether and how it could be verified using JML. This was done in two stages: UMLsec to protocol; and protocol to code where each protocol component should map to code in the prototype. This protocol-to-code mapping check would have been easier had the protocol been implemented using one method to process each message in each class. Such a consistent structure would be essential for cost-effective manual checking by someone not already familiar with the code. If the contract for each method is then written in JML, it can be automatically checked statically and at runtime to complement a manual check.

This mapping check revealed several issues, which were examined as to how easily they could be detected using JML. Firstly, a MAC check had been omitted. This could be detected by coding a suitable JML contract on the Java code level. Secondly, the

system deliberately did not implement secure storage of the smart card PIN or biometric template to protect their confidentiality or integrity because of insufficient development time. This could not be easily detected by a JML contract since the methods are either missing or return correct values.

The MAC and encryption cipher both use the same key, which is weak since if the key is broken then the whole system is insecure: different keys should be used for different types of cryptographic operations. This is a design weakness in the sequence diagram specification, which specifies usage of the same key. The code implementation actually correctly uses different keys here, so this would not have been revealed by a code-level check. This weakness is an example of a model-level security design check which could be very usefully added to the UMLsec verification tool framework.

There was initially an omission on the UMLsec specification level compared to the textual specification [2] regarding the generation of a new session key: Viti's protocol generates a session key from a combination of two nonces and a key stored in the smart card, but the UMLsec specification initially omitted the smart card key. This model-level inconsistency was revealed when coding, with the help of both the textual and the UMLsec specification. This example demonstrates the usefulness of performing assurance on both the model and the code level since the added redundancy further increases the trustworthiness of the resulting system.

An initial version of the implementation did not enforce the implied integrity requirements for the smart card ID in the smart card and host classes as there was no MAC check. This was a deliberate omission based on the erroneous assumption of the implementer that it would not lead to a security weakness. This is an example of a change by a developer with the intention of improving the protocol that actually introduces a security flaw. That this variation is insecure would have been revealed by the UMLsec model-level security analysis tools for verifying crypto-protocols. The inconsistency of the implementation with the UMLsec model can be revealed by JML contracts based on the UMLsec model if the change alters the interface or behaviour of methods (as it does in this example).

These issues reflect common issues occurring in industrial development: security specialists leave inconsistencies and flaws in specifications, and developers sometimes do not fully implement requirements because they forget, they make mistakes or they deliberately omit them to meet development deadlines. The success of our manual check in detecting these flaws is in line with earlier findings on the effectiveness of code reviews [15]. It argues for a rigorous manual code review against the specification as well as JML contracts, although this is time consuming. Further automated support to facilitate this check would therefore be very beneficial indeed.

4.5 JML Tools

JML aims to be supported by a range of open-source tools for statically checking assertions, checking assertions at runtime, unit testing support, and generating specifications and documentation [16]. The *jmlc* compiler and runtime assertion checker tests for violations of the JML assertions when the Java code is run. There are several static checkers, such as *ESC/Java*, that parse and type check JML, and statically check the consistency of the Java code against the JML specification.

We used the *ESC/Java 2* plug-in for the *Eclipse* workbench to statically check our JML and highlight syntax errors in the source code. The tool was easy to install and use, but had limited documentation. It identified some JML errors but not all warnings could be eliminated because of the missing explanatory documentation. Because of problems with installing the *JMLEclipse* runtime assertion checker plug-in, we could not investigate its usefulness towards verifying the system. We did however install the latest version of the common JML tools project [17], which are not yet integrated with *Eclipse*, and used the *jmlc* compiler to parse and type check the source code. This identified ten additional errors in the JML annotation previously missed by *ESC/Java 2*, although none revealed new security flaws. We then repeated the check using *ESC/Java 2* after installing the latest JML tools (including the latest version of the *jmlc* compiler) and it additionally reported errors in core Java classes within each prototype class. The likely causes of these errors are new language features of Java 6 in which the biometric authentication system is coded, since *ESC/Java 2* only supports earlier Java versions.

5 Lessons Learned

5.1 Evaluation of UMLsec

We found UMLsec and the associated cryptographic notation adequate for describing the system's security requirements since only three requirements could not easily be described: connection timeout value, protocol termination and types of communications links. Although the UMLsec notation is aimed to be extensible by the user in a given application, it would be very useful to have a process which would feed these ad-hoc extensions back into the standard UMLsec notation. Also, the meanings of some UMLsec stereotypes were not immediately easy to understand.

The UMLsec approach defines a threat model describing different kinds of adversaries. We had initially assumed a default adversary with no access to the trusted wire link between the smart card reader/scanner and host. However, vendors commonly implement a cryptographic protocol over this link, implying it is un-trusted. We therefore changed our assumption to an insider adversary that was able to delete, read or insert messages on this link, which required a substantial change to design and implement a cryptographic protocol within the biometric authentication system to preserve the security requirements. This emphasises the value of UMLsec's threat model in forcing attention on the nature of the security threat.

The associated cryptographic notation is succinct and unambiguous but not easy to understand without substantial study. It therefore requires a complementary summary textual description for the non-specialist reader, although we recognise the risk of the text becoming inconsistent with the diagrams if any changes are not applied consistently to both.

One could use general-purpose graphics software for the UML diagrams but this insufficiently supports the growing complexity of these diagrams and their consistency, and would not allow one to use the security analysis tool framework available for UMLsec. It is therefore important to create the UMLsec models in a general-purpose UML editor that allows the UML diagrams be imported as XMI files into the

UMLsec tool framework. It would also be very useful if these UML editors could be extended with the ability to highlight or filter out layers of information, although this is beyond the scope of the UMLsec tool framework since it assumes the use of a general-purpose UML editor.

The value of UMLsec diagrams is diminished if they are not maintained beyond the specification stage. Errors and omissions may be introduced into systems during changes after initial design, since the focus is often then on the detail of each change rather than on the effect on other aspects of the system, such as security. In industrial applications, changes are often not applied back to system specification documentation because of time, manpower or budgetary pressure. A project must commit to updating the system specification and UMLsec notation for every subsequent change if it is to fully benefit from its investment in applying the UMLsec approach in initial design. It would also be very useful to have automated tool-support that automatically reflects back changes on the implementation level to the model level. Some steps in this direction are documented in [6].

The analysis used version 1.5 of UML because most of the source material used this version. However UML is continually being enhanced. [1] assesses the effect of UML 2.0 on UMLsec as minor because the new version is sufficiently conservative to the previous diagram types and the new model elements are not needed in security engineering. For UML to have a long-lasting and deep impact in practice, the Object Management Group will need to ensure that future versions of UML continue to be conservative extensions of the previous ones, if organisations are to have confidence that an investment in adopting UML will provide long-term benefits.

UMLsec allows one to specify and automatically analyse security requirements and security design models but it does not prescribe how to create the design so that it will then be shown to satisfy the security requirements using the UMLsec analysis tools.. Thus, this input from a security designer is still needed. UMLsec also does not describe the level of security to be specified at the implementation level. For example the designer is not given explicit guidance on the cryptographic algorithms or key strengths necessary. This would again be a very useful addition to the current UMLsec notation and tools.

The UMLsec specification was a sound basis for design and code implementation, particularly the sequence diagrams as they were at a level of abstraction from which they could be directly coded.

5.2 Evaluation of the JML Approach

The value of using JML to verify the prototype code was limited because it was applied after, rather than during code development. Most of the methods were too large, which made it difficult to check many conditions using JML. JML was of limited value for small helper methods because they were not designed with clear pre- and post-conditions that JML could easily check. JML would therefore be more valuable if applied earlier starting in the design phase.

There was insufficient time to fully evaluate JML tool support. The *jmlc* compiler identified many syntax and type JML errors but *ESC/Java 2* was less useful, probably because it has not been maintained for newer versions of Java. Industry will be reluctant to adopt JML without some assurance of tool maintenance since it would constrain

their ability to use new language versions. The new common JML tools are open source which would allow significant users in industry to update the tools to newer version of Java themselves, but most industrial users will not have the resources to do this themselves and will expect a commercial support package.

Although JML does not directly support message or storage confidentiality or integrity verification, we were able to code contracts to indirectly check these security requirements. However JML patterns for confidentiality and integrity were difficult to apply because many system methods were too large and complex; these patterns would have been more useful with smaller methods. Some guidance on the use of JML patterns during design is therefore needed.

JML's current support for verifying that messages are sent in the correct sequence using a finite state machine is somewhat cumbersome and error prone, so we would endorse proposals for a JML call sequence clause to specify the method protocol more succinctly.

JML specifications for a few Java cryptography classes provide to some extent a more precise description of class behaviour than the *javadoc* documentation but they take time to understand. They complement the *javadoc* documentation but are not a substitute for it being made more precise.

Using JML, we identified six instances of requirement implementation and integrity check omissions, and Java code errors. However a subsequent manual check of the prototype's code against the UML specification successfully identified a further 13 security flaws, inconsistencies and weaknesses, some of which were discussed in Section 4.4. Writing JML contracts thus effectively enforces parts of the security check by focusing attention on the code and specification, but it does not reveal all the flaws and specification inconsistencies present It would require considerable time and a good knowledge of the specification, application code and security to thoroughly manually check code, which might not be feasible in an industrial situation with time and cost pressures. However a check could be made by following design and coding style guidelines, and by using a checklist, which would help ensure that important areas were covered and act as documentation of the quality review.

JML will not detect security flaws contained in products, design features not implemented in code, associated business and operational processes, and infrastructure.

JML verifies code against a derivation of the UML specification rather than the specification itself: security flaws might also be missed if this derivation was not complete.

JML contracts impose a code structure if they are written during design. If these contracts are written by software or protocol designers then the developers' role is reduced to implementing and testing each defined method. Developers might regard this as diminishing their role and so resist its introduction. To avoid this, they could be asked to write the JML contracts from specifications produced by the protocol designer, although an independent reviewer should confirm the consistency of the JML with the specification during a code walkthough (and it would also be very useful to have automated tools that would check this).

JML is not dependent on UMLsec or UML; JML contracts could be written from any specification that described requirements in a clear and unambiguous way. An organisation could therefore introduce JML and UMLsec separately to avoid overloading the organisation with change.

The use of JML would increase development timescales and costs: JML requires time to learn and apply since the literature is still academically-focused, and it is not sufficiently comprehensive to reduce normal system testing. An organisation adopting JML would therefore need to develop a software specification style guide for JML, and to train designers and developers. However it would contribute to reducing security flaws in the system, which would reduce costs because of fewer subsequent fixes, and it would reduce the risk of loss and reputational damage from the exploitation of security flaws. JML would be easiest to justify in areas where this cost reduction and these risks were highest; biometric authentication protocols would be one such area because of the impracticality of resolving errors in applications on issued smart cards.

5.3 Reflections on This Experience

In this subsection, we summarise our views on this application experience.

How much effort was involved? About 56 person days (pd), comprising: 11 pd creating the design model of the protocol; 15 pd on the technical design of the software architecture; 11 pd for coding and testing the prototype; and 19 pd for verification of the prototype using JML and the manual check.

Were there ways in which the application of UMLsec and JML did not go as expected? The UMLsec approach was effective in specifying security requirements succinctly and precisely, and the threat model was particularly useful in clarifying the extent of an adversary's access to the system. Only three requirements could not be easily described and the meaning of one UMLsec stereotype was unclear to the given user so it was not used. It was difficult to check many conditions using JML because most of the application's methods were too large.

Did the approaches have to be changed or adapted to work properly and, if so, in what way? The UMLsec analysis tools had been used to verify the correctness of the specification in earlier work [3], which therefore did not have to be repeated in the current application. After using JML to verify the code, we carried out an additional manual check for security flaws to examine the effectiveness of the approach. Apart from that, the approaches were not changed.

Did the method reveal interesting or unexpected results? The process of manually applying UMLsec identified two security weaknesses, even though we did not use the automated UMLsec analysis tools (which had been applied in earlier work [11]). JML helped to verify system code by focusing attention on the consistency of the code with its UMLsec specification, which revealed a number of unexpected security flaws and weaknesses.

Did it not pick up issues that you expected it would? JML did not identify some security flaws, design weaknesses and inconsistencies in the UMLsec specification because the implementation was not suitably structured. However some of these would probably not have been easily revealed by JML even if it had been fully applied.

How did its use differ from previous uses? To our knowledge, this was the first combined application of UMLsec and JML to a biometric authentication system.

Can you say anything specific about the security of the application now that you have done the modeling? We investigated in some detail the correctness of the application against its security specification but we cannot completely prove this correctness since JML verifies code against a derivation of the UML specification rather than the specification itself. Future research might develop a tool to generate draft JML specifications from UMLsec sequence diagrams to both improve JML coding efficiency and reduce the risk of omissions.

How can you be sure that you have applied the method correctly or even optimally? The value of JML was limited in so far as it was applied after code development to an implementation that was not entirely structured in a suitable way. It would have been more useful to have used it to specify methods already during the design phase.

6 Summary

This paper describes the application of the UMLsec and JML assurance approaches to a biometric authentication system, with the focus on the use of JML to verify the code against its UMLsec specification. UMLsec was effective in specifying security requirements, in particular in modeling threat levels. The implementation was straightforward as the UMLsec protocol was unambiguous and on the same level of abstraction as the code. JML helped to verify the code by focusing attention on its consistency with the specification, revealing a number of security flaws. However, its value was limited in so far as it was used after code development on an implementation with an unsuitable structure.

A tool to generate draft JML conditions from UMLsec would improve the value of JML in this context. Other research might map UMLsec to features of implementation language frameworks, develop UMLsec and JML security patterns, evaluate other JML tools in a security requirements context and integrate these techniques within a coherent security systems development method.

Acknowledgements. Many thanks to Jonathan Stephenson for advice and guidance during this analysis, to Atos Origin UK for sponsorship, and to the reviewers for providing constructive feedback which helped improving the paper.

References

1. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
2. Viti, C., Bistarelli, S.: Study and development of a remote biometric authentication protocol, Technical Report IIT B4-04/2003, Consiglio Nazionale delle Ricerche,Istituto di Informatica e Telematica (September 2003)
3. Grünbauer, J., Hollmann, H., Jürjens, J., Wimmel, G.: Modelling and Verification of Layered Security Protocols: A Bank Application. In: Anderson, S., Felici, M., Littlewood, B. (eds.) SAFECOMP 2003. LNCS, vol. 2788, pp. 116–129. Springer, Heidelberg (2003)
4. Deubler, M., Grünbauer, J., Jürjens, J., Wimmel, G.: Sound Development of Secure Service-based Systems. In: 2nd International Conference on Service Oriented Computing (ICSOC 2004), pp. 115–124. ACM, New York (2004)

5. Best, B., Jürjens, J., Nuseibeh, B.: Model-based Security Engineering of Distributed Information Systems using UMLsec. In: 29th International Conference on Software Engineering (ICSE 2007), pp. 581–590. ACM, New York (2007)
6. Houmb, S., Georg, G., France, R., Bieman, J., Jürjens, J.: Cost-Benefit Trade-Off Analysis Using BBN for Aspect-Oriented Risk-Driven Development, Engineering of Complex Computer Systems. In: 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005), pp. 195–204 (2005)
7. Leavens, G., Cheon, Y.: Design by Contract with JML (2006), ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf
8. Leavens, G., Baker, A., Ruby, C.: JML: A Notation for Detailed Design. In: Behavioral Specifications of Businesses and Systems, ch. 12, pp. 175–188. Kluwer, Dordrecht (1999)
9. Leavens, G., Baker, A., Ruby, C.: Preliminary Design of JML: A Behavioural Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes 31(3) (May 2006)
10. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Muller, P., Kiniry, J., Chalin, P.: JML Reference Manual, DRAFT, Release 1.210, 2007/7/01 Ames, Iowa State University
11. Schmidt, R.: Modellbasierte Sicherheitsanalyse mit UMLsec: ein Biometrisches Zugangskontrollsystem (Model-based Security Analysis with UMLsec: a Biometric Access Control System) Ludwig-Maxim. Univ. München (2004)
12. Jürjens, J.: Model-based Security Engineering with UML. In: Aldini, A., Gorrieri, R., Martinelli, F. (eds.) FOSAD 2005. LNCS, vol. 3655, pp. 42–77. Springer, Heidelberg (2005)
13. Agarwal, P., Rubio-Medrano, C., Cheon, Y., Teller, P.: A Formal Specification in JML of the Java Security Package. Computer, Information, and Systems Sciences and Engineering, December 4–14 (2006)
14. Warnier, M.: Language Based Security for Java and JML, PhD thesis, Radboud University Nijmegen (2006)
15. Glass, R.: Inspections - Some Surprising Findings. Commun. ACM 42(4), 17–19 (1999)
16. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. STTT 7(3), 212–232 (2005)
17. JML common tools, December 10 (2007), http://sourceforge.net/projects/jmlspecs/
18. Yu, Y., Jürjens, J., Mylopoulos, J.: Application of Traceability to Maintenance of Secure Software. In: Int. Conf. for Software Maintenance (ICSM). IEEE, Los Alamitos (2008)