A Security Analysis of a Biometric Authentication System using UMLsec and the Java Modeling Language

J G Lloyd

29 September, 2007

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk
A Security Analysis of a Biometric Authentication System using UMLsec and the Java Modeling Language

A dissertation submitted in partial fulfilment of the requirements for the Open University’s Master of Science Degree in Computing for Commerce and Industry

John Lloyd
M9164362

10 February 2008

Word Count: 15,806
Preface

I would like to thank my Supervisor, Jonathan Stephenson, for his continuous help and support through this dissertation and my Specialist Adviser, Dr Jan Jürjens, for his helpful comments. I acknowledge the considerable help of David Hook’s excellent book ‘Beginning Cryptography with Java’ and the use of his Java cryptography extension provider in producing a working prototype. I also acknowledge the financial sponsorship of Atos Origin UK.
# Table of Contents

Preface ..............................................................................................................................i

List of Figures .................................................................................................................v

List of Tables ..................................................................................................................vi

Chapter 1  Introduction ..................................................................................................1

1.1 Background to the research ..................................................................................1

1.2 Aims and objectives of the dissertation ..................................................................4

1.3 Overview of the dissertation ..................................................................................5

Chapter 2  Literature Review .......................................................................................6

2.1 Introduction ..............................................................................................................6

2.2 The UMLsec notation ............................................................................................6

2.2.1 fair exchange ......................................................................................................7

2.2.2 provable ............................................................................................................7

2.2.3 rbac ..................................................................................................................7

2.2.4 Internet, encrypted, LAN, wire, smart card, POS device, issuer node ..........8

2.2.5 secure links, secrecy, integrity, high ...............................................................8

2.2.6 Critical, data security ......................................................................................8

2.2.7 secure dependencies .......................................................................................9

2.2.8 no down-flow, no up-flow ............................................................................10

2.2.9 guarded, guarded access .............................................................................10

2.3 UMLsec analysis tools .........................................................................................10

2.4 Techniques and tools for verifying code ...............................................................12

2.5 UMLsec comparison ............................................................................................13

2.6 Biometric authentication .......................................................................................13

2.7 Code validation techniques ..................................................................................15

2.7.1 Java Assert ..................................................................................................15

2.7.2 JML ..............................................................................................................16

2.7.3 Bogor ............................................................................................................18

2.7.4 Prolog ...........................................................................................................19

2.8 Summary ..............................................................................................................20
# Chapter 3 Research Method

3.1 Introduction ......................................................................................................... 21

3.2 System specification ............................................................................................ 21

3.2.1 Host – smart card protocol ........................................................................... 23

3.2.2 Host – scanner protocol ................................................................................ 26

3.2.3 Host – server protocol .................................................................................. 27

3.2.4 System operations specification ................................................................... 29

3.3 Prototype design .................................................................................................. 29

3.4 Prototype implementation ................................................................................... 30

3.5 Code validation technique selection and use....................................................... 31

3.6 Summary ............................................................................................................. 32

# Chapter 4 Evaluation of the UMLsec approach

4.1 Introduction ......................................................................................................... 33

4.2 UMLsec diagram development ........................................................................... 33

4.3 Prototype design .................................................................................................. 36

4.4 Prototype implementation ................................................................................... 37

4.5 Summary ............................................................................................................. 38

# Chapter 5 Code verification using JML

5.1 Introduction ......................................................................................................... 39

5.2 JML code validation ............................................................................................ 39

5.3 JML tools ............................................................................................................. 42

5.4 Confidentiality and integrity JML patterns ......................................................... 43

5.5 JML specifications .............................................................................................. 43

5.6 Manual code validation check ............................................................................ 44

5.7 Limitations of the approach .............................................................................. 47

5.8 Summary ............................................................................................................. 48

# Chapter 6 Conclusions

6.1 Dissertation review .............................................................................................. 49
6.2 UMLsec evaluation ........................................................................................................49
6.3 Implementation from the UMLsec specification .......................................................51
6.4 Implementation verification .......................................................................................51
6.5 Future research on enhancing tool support ..............................................................53
6.6 Other future research ...............................................................................................56
References ......................................................................................................................58
Bibliography ..................................................................................................................61
Index ...............................................................................................................................63
Appendix A Cryptographic notation and variables .........................................................64
Appendix B Glossary of terms .........................................................................................66
Appendix C Biometric Authentication System Operation Specifications ....................69
Appendix D Biometric Authentication Prototype Source Code .................................75
Appendix E JML Signature specification comparison with javadoc documentation ...................105
List of Figures

Figure 2.1 The UML verification framework ................................................................. 11
Figure 2.2 Viti and Bistarelli's biometric authentication system ............................... 14
Figure 3.1 Biometric authentication prototype deployment diagram ......................... 21
Figure 3.2 Biometric authentication prototype class diagram ..................................... 22
Figure 3.3 Sequence diagram for user interaction ...................................................... 23
Figure 3.4 Sequence diagrams for negotiation of session key and PIN check .......... 24
Figure 3.5 Sequence diagram for decrement of misuse counters .............................. 25
Figure 3.6 Sequence diagram for biometric scan ....................................................... 26
Figure 3.7 Sequence diagram for biometric matching ................................................. 27
Figure 3.8 Sequence diagram for HTTPS handshake ............................................... 28
Figure 3.9 Final prototype software architecture ...................................................... 29
Figure 6.1 Tool verification support ........................................................................... 54
Figure 6.2 Sequence diagram components ............................................................... 54
Figure 6.3 Draft JML specification generated from sequence diagram components .... 55
List of Tables
Table 3.1 Prototype Classes ............................................................................................31
Table A.1 Cryptographic notation used in UML sequence diagrams .........................64
Table A.2 Variables used in UML diagrams ..................................................................65
Abstract

The UMLsec approach is intended to be a simpler way for software designers to specify system security requirements and to verify their correctness. UMLsec is a set of Unified Modeling Language (UML) stereotypes with associated tags and constraints that can be added to UML diagrams to specify security requirements such as secrecy, integrity and authenticity. The approach includes the description of protocol security requirements using a cryptographic protocol notation. The UML diagrams can then be analysed by a set of tools to automatically verify these requirements for correctness.

However, even if the specification is provably correct, security flaws might be introduced during the design, implementation and subsequent maintenance of the system through errors and omissions. The UMLsec approach includes a set of techniques and tools that seek to automatically verify that implemented code does not contain security flaws, but these techniques and tools do not yet relate back to the specified security requirements in UMLsec so ways are needed to verify that the implemented system is correct in relation to this specification.

This research dissertation designed a prototypical biometric authentication system using UMLsec to evaluate how easy UMLsec is to use in this context and to investigate how easy it is to implement a system in Java from this design. It then examined the use of the Java Modeling Language (JML) to relate the code back to its specification to verify that the implementation was secure.

The UMLsec approach was effective in specifying security requirements succinctly and sufficiently precisely to avoid significant change during coding, although the capabilities of the implementation language need to be taken into account to avoid redundancy in the specification. The threat model was particularly useful in clarifying the extent of an adversary’s access to the system. However UMLsec is not a design or implementation approach and so does not assist with issues such as selecting the type or strength of security algorithms.

The approach should contribute to reducing the high training and usage costs of formal methods but it will need training, simpler documentation and CASE tool support to appeal to industry. The UMLsec specification would also need to be maintained throughout the life of a system so that all changes made could be verified, which would increase maintenance effort and cost.
JML was applied to parts of the prototype code and helped to verify it by focusing attention on the consistency of the code with its UMLsec specification, which revealed a number of security flaws and weaknesses. However a subsequent manual check revealed more flaws, design weaknesses and inconsistencies in the UMLsec specification, some of which would not have been revealed by JML even if it had been fully applied.

The jmlc and ESC/Java 2 tools were used to compile and statically check JML against the Java code. However there is no tool support for verifying the JML against a UML specification so the Java code could not be verified against its UMLsec specification via JML. The JML specifications might therefore not completely reflect the UMLsec specification.

The value of JML was limited because it was applied after code development to a software design that was not entirely suitable; it would have been more useful to have used it to specify methods during design and made less use of the Object Constraint Language (OCL) to specify class operations. JML was also sometimes cumbersome to use in verifying security requirements. Any organisation adopting JML would therefore need to develop a design style guide and security patterns, and provide adequate training for designers and developers. Using JML does not eliminate security flaws since they could be contained in products, design features not implemented in code, infrastructure, and associated business and operational processes.

Future research should develop a tool to generate draft JML specifications from UMLsec sequence diagrams to improve JML coding efficiency and reduce the risk of omissions. 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.
Chapter 1  Introduction

1.1 Background to the research
Modern society has now become highly dependent on computer systems to administer and operate business, government and defence. Consequently, successful attacks on computer systems can do substantial financial damage, threaten lives and erode confidence in organisations.

Systems have become more open to attack as they have increasingly adopted distributed networked technical architectures involving publicly-accessible Internet and wireless links. As a result, system designers have increasingly sought to protect the confidentiality, integrity and authenticity of data sent across these networked links against threats.

Most systems require users to authenticate themselves before they are able to use the application or to gain access to premises protected by a system. The widespread use of electronic payments and online banking provides new opportunities for theft and fraud, placing much more emphasis on the need for reliable user authentication. However, designing and checking security mechanisms is very difficult to do correctly because of the complexity of these mechanisms and of industrial systems, the interaction with adversaries and the ways in which systems use these mechanisms. There is currently still no overall methodology for ensuring security in the development of general-purpose IT systems.

The traditional industrial approach has been to describe security requirements textually, design security features as an add-on to the system design, then patch flaws found or exposed after deployment - ‘penetrate and patch’. This approach is deficient in several ways:

- specifying security requirements textually is lengthy, imprecise and difficult to check, which risks leaving security vulnerabilities;
- adding security mechanisms to an existing design may constrain these mechanisms’ use and effectiveness; and
- penetrate and patch is risky and often expensive because major loss or damage might be suffered before discovery, and making changes to operational applications is often complex, time consuming and error-prone.
The increasingly serious consequences of inadequate system security point to the need for a rigorous security design and implementation methodology to minimise the risk of security flaws. This methodology should:

- specify security requirements sufficiently succinctly and precisely to enable their verification before design starts;
- integrate security and functionality requirements so security mechanisms are in turn integrated into the system design;
- design and implement software to support verified requirements without introducing new security flaws; and
- verify that the implemented software is free from vulnerabilities.

Researchers have developed formal methods - languages based on mathematical techniques - for specifying a system in such a way that properties of the specification can then be formally verified for correctness. Proving system properties can be done manually as a hand-written mathematical proof, or automatically using automated theorem proving or model checking. The Z language based on first-order set theory and the VDM specification language, based on propositional calculus and predicate logic, are examples of formal methods.

However formal methods have not been widely adopted because the complexity of both the languages and the systems modelled mean that staff need considerable skill, training and time to produce specifications and proofs, which incurs high costs. In practice, formal methods have been limited to use in high-integrity systems where safety or security is sufficiently important to justify their costs, such as in aerospace and nuclear power engineering.

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 proposed by Jürjens (2005) addresses this by expressing security requirements within a Unified Modeling Language (UML) system specification and then using analysis tools to verify these requirements for correctness.
UML arose from the unification of three object-oriented methods and is now a widely-used Modeling language. It contains a stereotype mechanism to extend the language by introducing new concepts: Jürjens uses this mechanism to introduce a set of security requirements concepts which he collectively calls UMLsec.

The UMLsec approach is a formal methods approach but Jürjens claims that it is easier for general software designers to use than a formal methods specification language because UMLsec is simpler and integrated with UML. Jürjens claims that this difference should reduce the very high training and usage costs that have marginalised the use of formal methods. If so, this approach would probably also 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. This dissertation will evaluate this claim by using the UMLsec approach to specify security requirements, using a case study of a biometric authentication system proposed by Viti and Bistarelli (2003).

Biometric authentication uses a unique body feature such as a fingerprint or eye iris pattern to help confirm a person's identity, often in combination with other factors such as a smart card and PIN. Biometrics are attractive for authentication because they are an inherent feature of a person and so cannot be lost or easily changed. The technology for reliably reading fingerprints and irises has now matured to the point where they are in widespread use: the U.S. Department of Homeland Security's US-Visit programme, which requires machine-readable passports containing biometric data from foreign visitors, is an example of the large-scale use of biometrics.

Even though the security requirements specification might be provably correct, security flaws may be introduced during the design and implementation of the system or in subsequent changes. The UMLsec approach includes a set of techniques and tools that seek to automatically verify that implemented code does not contain security flaws. However these techniques and tools do not all relate back to specified security requirements and no work appears to have been published on directly verifying implemented system code against security requirements expressed in UMLsec. This dissertation will therefore investigate ways to relate the implemented system back to its sequence diagram security specification and verify that it is correct in relation to this specification.
1.2 Aims and objectives of the dissertation

The research question addressed in this dissertation is 'How effective is UMLsec when used in the development of secure systems?'.

The dissertation’s objectives are:

- to evaluate UMLsec when applied to the design of a biometric authentication protocol;
- to investigate how easy it is to implement a prototypical system from this UMLsec specification; and
- to examine how this implementation could be related back to its specification to verify that the implementation is secure.

The dissertation excludes the following areas:

- UML activity, statechart and use case diagrams;
- assessment of tools to automatically verify the UMLsec specification for completeness or correctness;
- comparison of UMLsec against other formal method specification languages;
- examination of the formal foundation of the UMLsec approach;
- full development of smart card, scanner and server software; and
- assessment of a CASE tool for UML diagram development.

Jürjens (2005) used the UMLsec approach to successfully expose flaws in a variant of the Transmission Layer Security (TLS) protocol and in the Common Electronic Purse Specification. He is also using it in a number of industrial applications including a banking application, biometric authentication system and electronic health card. However the UMLsec approach has not yet been used independently or by software designers with no background in the approach so this dissertation provides an independent analysis of the use of UMLsec by an industrial software designer.

The audience for this dissertation is those in the academic application design, IT security and biometric authentication communities interested in the specification and verification of security requirements. There may also be interest within industry since the use of biometrics to improve authentication is currently a very active area, driven in the UK by the Government's ID card and border controls programmes, and by private sector initiatives to improve retail payments security.
The level of industrial interest will depend on the degree of UML knowledge required to specify security requirements, and the skill and time needed to verify code against this specification. The rigour with which systems are analysed and designed in industrial practice often falls short of academic theory due to time, cost and skill constraints. In my experience, systems analysts frequently specify both functional and system security requirements in text, schematic diagrams or simple UML such as use case diagrams. Software system designers or programmers then translate this specification into code using system design documentation of sometimes limited rigour. Verification of code is almost always done by various forms of testing, and is frequently constrained by budget, deadline and testing resource constraints.

A technique for verifying code against a UML specification would therefore need to demonstrate reduced time and improved accuracy compared to normal testing, but without needing a rare combination of good coding and security analysis skills. These benefits would need to be sufficiently large to justify the effort and skill required to construct the full UML specification, although the benefits would be difficult to measure objectively.

1.3 Overview of the dissertation
Chapter 2 describes the UMLsec approach and reviews literature on it, on the biometric authentication system used as a case study and on code verification techniques.

The research method is then described in Chapter 3. This method comprises the development of UML system requirement diagrams, system design, system implementation and selection and evaluation of a code validation technique.

Chapter 4 describes the evaluation of the UMLsec approach when applied to the design of the prototype biometric authentication protocol, and of the implementation of a prototypical system from this UMLsec specification. Chapter 5 then describes the use of the Java Modeling Language (JML) to try to verify the prototype’s code against this specification. Finally, Chapter 6 draws conclusions from this work and makes recommendations for further research.

Appendices describe the cryptographic notation used in UML sequence diagrams, provide a glossary of terms, describe class operations, list the prototype source code and JML, and compare JML specifications with javadoc documentation.
Chapter 2 Literature Review

2.1 Introduction

This section describes the UMLsec approach and reviews literature on it, on the biometric authentication system used as a case study and on code verification techniques.

The UMLsec approach consists of:

- the UMLsec security requirements notation for UML diagrams, which is summarised in section 2.2;
- a set of UMLsec analysis tools for verifying diagrams - section 2.3; and
- a set of techniques and tools for verifying code - section 2.4.

The approach is described in detail in Jürjens (2005) and compared with other approaches in section 2.5.

Section 2.6 then summarises the biometric authentication system used for this research and section 2.7 reviews literature on four candidate code validation techniques considered in this research.

2.2 The UMLsec notation

UMLsec is a set of UML stereotypes with associated tags and Object Constraint Language (OCL) constraints that are used to describe security requirements such as secrecy, integrity and authenticity in UML diagrams (Jürjens 2005a). UMLsec contains the following stereotypes:

- fair exchange;
- provable;
- rbac;
- Internet, encrypted, LAN, wire, smart card, POS device, issuer node;
- secure links, secrecy, integrity, high;
- critical, data security;
- secure dependency;
- no down-flow, no up-flow; and
- guarded, guarded access.
In a UML diagram, each stereotype is enclosed in chevrons, e.g. <<fair exchange>>, each tag is enclosed in braces, e.g. {adversary=insider}, and each OCL constraint is enclosed in square brackets, e.g. [i < 3].

UMLsec is also used to define the abilities and initial knowledge of an adversary in a threat model, such as whether he can read, insert or delete messages sent between parties.

2.2.1 fair exchange
This stereotype represents a requirement that parties to a transaction should be prevented from cheating. The stereotype uses three tags: {start}, {stop} and {adversary}. {start} and {stop} both define a product to be sold and the state of a system in an activity diagram; whenever a start state is reached then eventually a stop state must also be reached. For example, a buyer must pay for goods purchased from a seller – a ‘purchase’ start state - before taking delivery of them – a ‘deliver’ stop state.

The {adversary} tag defines a level of attacker against which the security requirements should hold and the attacker’s previous knowledge of the system. For example, a default adversary represents an outsider who can delete, read and insert messages on an Internet link but has no access to a wire link. An insider adversary can also delete, read and insert messages on a wire link. The initial knowledge of the adversary is assumed to exclude information protected by security requirements.

2.2.2 provable
<<provable>> represents a requirement that an action at a system state defined in an {action} tag was actually performed. An expression in a {cert} tag is then output as evidence that the action was performed.

2.2.3 rbac
This stereotype enforces role-based access control, where system users are assigned roles that carry access and activity rights. A {role} tag defines a list of pairs of actors from the activity diagram and their roles. A {right} tag defines a list of pairs of roles and protected states of a subsystem. A {protected} tag defines these protected states for which access to activities should be controlled. For example, a supervisor might be given a credit approver role to authorise credit - the state in a process of granting credit.
2.2.4 Internet, encrypted, LAN, wire, smart card, POS device, issuer node
These stereotypes are used in UML deployment diagrams to describe types of communications links and system nodes. Adversaries may, depending on their level, be able to delete, read or insert messages on these links, and access these system nodes.

2.2.5 secure links, secrecy, integrity, high
The <<secure links>> stereotype labels a UML subsystem containing dependencies whose physically implemented communication links are trusted to be secure against an adversary described in the {adversary} tag. A dependency may be labelled with <<secrecy>>, <<integrity>> or <<high>> stereotypes to require that the link preserves data confidentiality, integrity and no down-flow between two subsystems. Confidential information must not be read in a form that its meaning can be understood. Integrity is preserved if the information is not changed during transmission. Highly-sensitive information must not leak to, or affect low-sensitivity areas of the subsystem where it might be exposed.

A secure link implies that an application would not need to provide additional measures to protect the security of data; for example, if a dependency is labelled <<secrecy>> then the trusted link might encrypt data traffic so there would be no need for an application designer to design a security protocol to protect confidentiality. However, if a link is labelled <<Internet>> then this would, by itself, provide no security protection against an adversary and would therefore need additional security measures.

2.2.6 Critical, data security
The <<critical>> stereotype labels objects or subsystems containing data that has critical security requirements, where the subsystem is labelled with the <<data security>> stereotype. These requirements are described using the {secrecy}, {integrity}, {authenticity}, {fresh} and {high} tags, where the:

- {secrecy} tag lists attributes, expressions or message arguments that should remain secret;
- {integrity} tag lists attributes of an object that should be protected against being changed, or optionally being changed to any expression not listed in this tag;
- {authenticity} tag lists the attributes whose origins should be authenticated;
- {fresh} tag lists data and keys that should be freshly generated, for example, to prevent a message replay attack; and the
• {high} tag lists highly sensitive messages that should be protected from direct or indirect leakage into a low-sensitivity part of the subsystem. The subsystem is labelled with the <<no down flow>> or <<no up flow>> stereotype if this tag is used.

These requirements are with respect to a threat defined in an {adversary} tag also associated with the <<data security>> stereotype.

2.2.7 secure dependencies

The <<secure dependencies>> stereotype labels a subsystem to require that dependencies respect the security requirements of messages communicated across them. These security requirements are defined by the {secrecy}, {integrity} and {high} tags of the stereotype <<critical>> on messages at each end of the dependencies. These security requirements must be consistent; for example if a message name is labelled {secret} in one object then the method in the called object must also be labelled {secret}.

<<secure links>>, <<data security>> and <<secure dependencies>> all label subsystems. The distinction between them is:

• <<secure links>> applies to the physical communication links implementing dependencies in the subsystem;
• <<data security>> applies to tagged static data in the subsystem; and
• <<secure dependencies>> applies to the preservation of message security requirements passing across dependencies in the subsystem.

The meaning of, and distinction between these stereotypes should be made much clearer in a non-mathematical way, and should be illustrated with further examples to avoid misunderstanding by designers, who will not have the time or commitment to work through mathematical descriptions.
2.2.8 no down-flow, no up-flow
The <<no down-flow>> stereotype labels a subsystem containing some sensitive data and operations labelled with the <<critical>> stereotype and its {high} tag, and requires the prevention of any indirect leakage of sensitive information via low sensitivity data or operations. For example, hospital administrators might not be able to see sensitive patient data using a patient administration system’s screen transactions but if they can run a report on hospital activities which includes clinical procedures carried out on patients then this leaks sensitive patient data.

The meaning of the <<no up-flow>> stereotype is unclear: it appears to require the prevention of any ‘leakage’ of low-sensitivity information via highly sensitive data or operations. This might be inefficient or confusing but it would not carry a security risk.

2.2.9 guarded, guarded access
The <<guarded access>> stereotype labels subsystems to require that each object stereotyped <<guarded>> and tagged with a {guard} tag cannot be accessed directly but only through the object identified in the {guard} tag. The guarding objects would typically have constraints defining the security checks that they should carry out.

2.3 UMLsec analysis tools
UMLsec diagrams can be verified with a set of analysis tools developed by Jürjens. The verification process is shown in Figure 2.1. A user firstly adds UMLsec notation to UML diagrams using an XML Metadata Interchange (XMI)-compliant UML Editor. XMI is a standard language developed by the Object Management Group (OMG) for exchanging UML models between tools. The user then exports these UML diagrams in UML XMI file format and imports them into a Metadata Repository (MDR) to position these requirements for verification. A MDR is a store of Modeling information that conforming to the OMG's Meta-Object Facility (MOF) standard. This standard describes a language and framework for specifying, constructing and managing Modeling languages such as UML.
Jürjens has developed tools for verifying the constraints associated with the UMLsec stereotypes. Each tool is called through a textual, GUI or web interface and accesses repository data using the Java Metadata Interface (JMI) standard, which defines a mapping of MOF objects to Java. The tools construct a formal model – a precise definition of the structure and behaviour of the system – which they then analyse to check whether the security requirements contain flaws.

The tools check static security features and simpler dynamic properties to verify basic behavioural security requirements. They can also convert more complex dynamic properties of the model into input language for an external checker such as an Automated Theorem Prover (ATP) (Jürjens and Shabalin 2004). An error analyser then provides a report on the results (Jürjens and Shabalin 2005).

These tools are not yet available either as open source or commercially, and are not used in this dissertation. The degree to which they verify requirements and their usability has not been independently analysed.
2.4 Techniques and tools for verifying code

Security properties may not always be preserved by the refinement of a specification into a system, for example because of design or coding errors, or subsequent functional changes where security has not been adequately considered. The UMLsec approach contains a set of techniques and tools for verifying that the implemented system preserves security properties by run-time security checks, tests generated from the UMLsec specification, automated code security analysis and automated analysis of security configuration data (Jürjens 2006a).

Jürjens (2005b) describes one form of automated code security analysis using control flow graphs generated from a C program and an Automated Theorem Prover (ATP) which may reveal security flaws. These flaws can then be shown most efficiently by transforming the ATP input code into Prolog and generating an attack trace. The ATP may raise ‘false positives’ - attacks that are not realistic in reality - but these can be detected using the Prolog attack generator.

Jürjens (2006b) modifies this analysis for crypto-based Java implementations by introducing comments annotating the code to more abstractly define security behaviour. These comments are used to insert explicit transition information and first order logic formulae in the control flow graph. The comments are also compiled into run-time checks using the Java assert command to confirm that the logical conditions are not violated during software execution. However this annotation is a manual task requiring an understanding of both the code and the annotation semantics, which constrains its scalability.

Jürjens successfully applies this analysis to three industrial-sized applications. When applied to a variant of the Transmission Level Security (TLS) protocol, this analysis immediately detected a flaw that could be exploited by a man-in-the-middle attack. The analysis also revealed a flaw in the Java implementation of part of an electronic purse specification where messages could be redirected to allow a merchant to claim payment for goods actually supplied by another merchant. However the analysis verified the security of the open source implementation of the Java Secure Sockets Extension (JSSE) of the SSL protocol.
Heldal and Hultin (2003) use an extension to UML and the Java + Information Flow (Jif) language to automatically verify Java code. Their extension - UML for Security (UMLS) - uses labels to describe security constraints on data instead of stereotypes and tags because they claim that labels are easier to read. Each label describes a constraint in terms of the principal - the user or role - owning confidential data and a list of principals who are allowed to read it. These labels are used in class and collaboration diagrams. Jif is a security-typed programming language that extends Java with support for information flow control and access control at compile time and at run time. Heldal and Hultin insert labels in manually written Jif code implementing class, data and method definitions corresponding to UMLS labels, and then use a Jif compiler to validate the code. However this approach is less comprehensive that UMLsec as it does not support integrity, authenticity, freshness, or deployment or state chart diagrams. It also does not provide security verification of the UMLS diagrams themselves.

2.5 UMLsec comparison
Mellado et al. (2006) compare eight proposed approaches for establishing system security requirements during the information systems development process, including an extended use-case process using UMLsec. They compare these approaches on five criteria: their degree of agility; the support available; the degree of integration with other software environments; user friendliness and new security contributions. They rate the extended use-case approach quite highly except in its degree of agility - its ability to model requirements and incorporate changes quickly. This stems from UML’s role in modeling deliverables in a sequential analysis-design-build systems development method rather than in more agile ways.

2.6 Biometric authentication
This dissertation evaluates the UMLsec approach using a biometric authentication system case study described by Viti and Bistarelli (2003) and shown in Figure 2.2. They review the advantages and disadvantages of the main authentication factors in use, and promote biometric authentication factors as a major step forwards. They then propose a basic biometric authentication system and develop a prototype implementation in C for both Internet Explorer and Netscape/Mozilla browser-to-smart card cryptographic communication. The protocol uses the authentication handshake in the Secure Sockets Layer (SSL) protocol.
The authentication process consists of five main steps:

1. insert a smart card containing a fingerprint biometric template into the USB port of a combined smart card reader/scanner device;

2. enter a PIN at the controlling host PC to activate the card;

3. place a finger on the scanner and have the scan compared to the fingerprint template;

4. if they match, encrypt a nonce\(^1\) sent from the server with the card’s private key and return it to the server; and

5. the server application authenticates the user by decrypting the nonce with the user’s public key and confirming it matches the nonce that it sent to the host.

---

\(^1\) A nonce is a method of ensuring that a message is fresh by inserting an unpredictable value so that any replay of the message can be detected. The term is derived from ‘Number used ONCE’
Schmidt (2004) documents a smart card-based biometric authentication protocol for an entry protection system developed as part of the Verisoft project, which is researching the pervasive formal verification of computer systems. The protocol mutually authenticates a host and smart-card, generates a symmetric session key pair on the smart-card and host, uses it to securely transfer the reference template to the host and then matches a live biometric scan with this template. If there is sufficient similarity, the user is authenticated. Schmidt describes the protocol in UMLsec, uses an ATP to verify the protocol and Prolog to analyse a flaw.

Jürjens (2005a and 2005b) applies the UMLsec approach to a biometric authentication protocol inspired by Viti and Bistarelli. The analysis detected a flaw where the biometric match could continue to be made without decrementing the misuse counter, permitting unlimited authentication attempts by an adversary until a match was obtained. Protocol fragments from these papers and from Schmidt are incorporated in this dissertation’s biometric authentication protocol.

2.7 Code validation techniques
This section briefly reviews literature on four code verification techniques:

- Java assert constructs;
- the Java Modeling Language (JML);
- Bogor; and
- Prolog.

2.7.1 Java Assert
The Java assert construct can help support an informal design-by-contract programming approach. It has the form ‘assert expression’; when the system runs the assertion, it evaluates the expression and throws an AssertionError if it is false. For example:

```java
assert i == 2;
```

would throw an exception if \( i \) did not have the value 2. By default, assertions are disabled to avoid the performance penalty of evaluating them but may be enabled when required.
Assertions are inappropriate to validate pre-conditions in public methods because, by convention, they guarantee enforcement through specific checks in code and because assertions cannot throw the specified exception. However assertions are appropriate for validating pre-conditions in non-public methods and for validating post-conditions in public and non-public methods.

2.7.2 JML

The Java Modeling Language (JML) is based on the design by contract approach and can be used to specify the behaviour of Java modules. Leavons and Cheon (2006) describe design by contract as 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.

Leavons, Baker and Ruby (1999 and 2006) describe the basic features of the language, and these papers are complemented by Leavons et al’s JML reference manual (2007). JML is written as annotation comments beginning with //@ within Java code. A requires clause specifies the method's pre-condition and an ensures clause specifies normal and exception post-conditions. For example:

//@ 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, for example, a negative argument in this function call.

Leavons and Cheon (2006) claim that understanding program code is easier with JML because it supports modularity of reasoning. To understand a code method also requires an understanding of the code in its called methods, which is termed non-modular reasoning - reasoning about many code modules. JML specifications are at a higher level of abstraction so understanding a method requires only an understanding of the JML specifications of its called methods, not their code. This allows a method to be understood more quickly.
JML is claimed to be supported by a range of open-source tools for statically checking assertions, checking assertions at runtime, unit testing support, generating specifications and documentation (Burdy et al, 2005). The jmlc compiler and runtime assertion checker tests for violations of the JML assertions when the Java code is run and the jmlunit tool uses this to automate unit test outcome reporting. There are several static checkers, such as ESC/Java, that parse and type check JML, and to differing degrees of rigour statically check the consistency of the Java code against the JML specification. Three tools help a developer write JML specifications and the jmldoc tool produces browsable web pages describing the API and specifications in the style of Java documentation generated by javadoc².

Agarwal et al (2006) claim to have documented a significant portion of the java.security package in JML, including the MessageDigest and Signature classes used in this dissertation's biometric authentication protocol prototype. They started from the javadoc API comments in the Java source code but found these incomplete and ambiguous and so 'reverse engineered' JML specifications from the code itself. They describe an 'abstract with model fields' pattern for producing an abstract specification, comprising model fields, pre- and post-conditions and private abstraction functions, for runtime assertion checking. However they found using JML classes for Modeling cumbersome and verbose.

Agarwal et al use a finite state machine model for checking that methods are run in the correct sequence, although they found this cumbersome, error prone and difficult to understand. Cheon and Perumandha (2006) propose a call sequence clause that would specify the method protocol - the sequence in which methods should be called - in a much more succinct and easy to understand way. This is very relevant to security protocols, where messages need to be exchanged in a strict sequence and some classes such as Signature have methods that must be called in sequence. However there may not yet be any tool support for this clause. They also note that code runs significantly slower with JML checking, which might cause timeout problems when testing security protocols so testing would normally need to be done without checking.

² javadoc is a tool for writing Java API documentation in HTML based on tagged source code comments.
Warnier (2006) introduces JML specification patterns for confidentiality and integrity, where he defines confidentiality as non-interference between variables of different security levels. If, for any values of high security variables, the values of low security input and output variables do not change then there is no leakage of confidential information. This pattern supports the <<secrecy>> and <<no down-flow>> UMLsec stereotypes. Warnier also defines a similar JML specification pattern for integrity, where the values of high security variables are independent of low security variable values; in other words, changing low security values does not affect high security values. However Jürjens (2006b) reports that JML has not yet been to specify cryptographic protocols.

JML has similarities to the OCL used to specify constraints in UML design models. Hamie (2002) examines the relationship between operation and invariant constraints expressed in OCL and both their Java implementation and JML specification. He shows how such constraints can be translated into JML but does not consider UML sequence diagrams or cryptographic protocols. The AutoJML tool is claimed to generate JML specifications from UML state diagrams in XMI format.

### 2.7.3 Bogor

Bogor is a model checking framework designed to be modular and extensible (Robby et al., 2006). The Bogor framework provides direct support for object-oriented languages, a modeling language, an open modular architecture and teaching materials. Bogor is implemented either as a stand-alone tool or as a plug-in for the Eclipse development tool.

Bogor checks systems specified in the Bandera Intermediate Representation (BIR). BIR has a high-level language flavour suitable for hand-coded models and a lower-level language often used as the target for automatic model compilers, although the two levels can be freely mixed. The high-level flavour provides support for Modeling Java features. Modellers can add new types, expressions and commands to the BIR language using an extension declaration and a Java package to implement the semantics of the extension. Robby et al. describe using the BIR’s extension facility to model Java byte code directly. BIR is comprehensive enough to resemble a programming language in its own right.
Bogor has focused on non-compositional verification – the checking of non-modular software - of concurrent programs. However the Bogor/Kaisan extension has been developed to enable the compositional verification of sequential Java programs by interpreting byte codes symbolically. Robby claims that this extension can automatically check code against user-supplied pre- and post-conditions, invariants and assertions defining contracts between modules. It also automatically generates JUnit test cases. Kaisan has a specification language with similar features to JML and the JML compiler can be used to translate the specification into executable code. Similarly, Bogor supports the checking of JML specifications. Kaisan is claimed to be able to reason using the implementation of an invoked method as well as its specified contract, which reduces the need to develop a comprehensive specification.

Robby et al make the point that model checking should be applied after computationally cheaper static analysis and testing techniques have been used and he recognises the current impracticability of checking large system code bases. One way of reducing the size of systems to be checked is to omit the checking of some library modules using slicing techniques. Another is to develop an abstract model of a complex implementation for checking, although this risks introducing errors in the abstraction.

Bogor appears to be able to check, within user-specified bounds, both design models expressed in BIR and Java code with JML contract specifications. Bogor design model checking would be less relevant where UMLsec is used since UMLsec analysis tools could be used. Bogor is a complex framework with a strong concurrency focus. Bogor appears more powerful than JML for implementation code checking since it is claimed to be able to check code against specifications.

2.7.4 Prolog

Prolog - PROgramming in LOGic - is a general-purpose logical programming language that uses the declarative programming approach. Prolog applies a resolution theorem prover to Horn clauses and supports negation by failure, where negative conditions such as not($B_i$) are proved by failing to prove the corresponding positive conditions $B_i$.

---

3 JUnit is a simple framework for writing repeatable tests in Java code that can be run simultaneously and do not require human judgement to interpret.
Prolog programs consist of clauses that describe relations. There are two types of clauses: facts and rules. The example \texttt{not(B)} is a rule clause. The body of this clause - \(B_1, \ldots, B_n\) - consists of calls to predicates \(B_i\) that are called goals, which can be combined in conjunction and disjunction. Prolog has a single data type called a term which may be either atoms (or invariants), real or integer numbers, variables or compound terms. A user poses a query as a single goal and the Prolog engine then tries to disprove the negated query; if it cannot then the query is shown to be a logical consequence of the program.

Programming library bridges between Java and Prolog, such as \textit{InterProlog}, allow Java objects to be mapped into Prolog and vice-versa. They also allow the development of GUI front-ends and other functionality in Java while leaving logic processing in Prolog. In \textit{InterProlog}, Java can call any Prolog goal, Prolog can invoke any Java method, and Java objects and Prolog terms can be passed between them.

As another example, \textit{SICStus} provides a Prolog development system with a bi-directional interface between programs written in Java and in Prolog. This interface consists of a Java package containing classes representing the \textit{SICStus} emulator and an extension to Prolog's foreign language interface where declarations map Java methods to Prolog predicates and automatically convert arguments.

\textbf{2.8 Summary}

The literature review has described the UMLsec notation and recognised that UMLsec appears effective when used by its originator in industrial projects to verify the UMLsec requirements and implemented code. However, there is no evidence of independent evaluation of its use by security system developers, in particular in the context of specifying security requirements for a biometric authentication system. There are techniques to verify code against its specification - such as JML, Bogor and Prolog - but no work has yet been published using any of these techniques to relate code back to a UMLsec specification.

The next chapter describes the research method for evaluating the use of UMLsec and for relating code back to the UMLsec specification. This method specifies the biometric authentication system introduced in Chapter 2, designs and implements a prototype of the system in Java, and then selects and uses a technique for code verification.
Chapter 3  Research Method

3.1 Introduction
The research method comprises four steps:
1. specify system requirements using UML diagrams, UMLsec primitives, a cryptography notation and operation pre- and post-conditions – section 3.2;
2. design the prototype’s software architecture – section 3.3;
3. implement the prototype design using Java – section 3.4; and
4. select a technique for code validation and apply it to the prototype to verify the implemented security features against the UMLsec requirements – section 3.5.

3.2 System specification
I firstly developed UML deployment, class and sequence diagrams for the protocol proposed by Viti and Bistarelli (2003). The system requirements are based on the fragments of biometric authentication protocols in books and papers in the literature survey (section 2.6). Table A.2 in Appendix A describes the terms used in these diagrams.

Figure 3.1 Biometric authentication prototype deployment diagram
The deployment diagram (Figure 3.1) describes the physical hardware and connections in the system: a controlling host PC, a server connected via an Internet link, and a scanner and smart card reader modelled as separate logical devices within a single hardware device.

The UML class diagram (Figure 3.2) describes the main SmartCardOS, Host, BioSensor and Server classes that model the smart card, host PC and scanner in terms of their attributes and operations, and it adds secrecy, integrity, freshness and authenticity security requirements using UMLsec. It also models other classes relevant to displaying screen messages to the user.

The sequence diagrams describe the interaction with the system user (Figure 3.3) and the protocol of messages between:

- the host and the smart card - dialogue A described in section 3.2.1;
- the host and the scanner – dialogue B described in section 3.2.2; and
- the host and the server – dialogue C described in section 3.2.3.

![Figure 3.2 Biometric authentication prototype class diagram](image-url)
These protocols use cryptographic data and functions that are modelled using a convention used by Jürjens based on an established approach initiated in Dolev and Yao (1983), which includes the terms listed in Table A.1 of Appendix A.

3.2.1 Host – smart card protocol

The host-smart card protocol consists of three parts:

- negotiation of an agreed session symmetric key (dialogue A in Figure 3.4);
- checking that the user-entered PIN is correct, using this agreed key (dialogue B in Figure 3.4); and
- decrementing a misuse counter to protect against unlimited PIN guesses (Figure 3.5).

The host initiates the dialogue by sending a ‘reset’ message which might be triggered by a ‘device plugged into USB port’ event. The smart card replies with an identifier containing a temporary symmetric key and the storage address of the smart card’s session key specification.
The host requests a nonce from the smart card and then returns this nonce with one of its own and the address, all encrypted with the temporary key. The smart card firstly checks a misuse counter that limits the number of server connection attempts. The smart card then checks its nonce, generates a new nonce and then generates a session key using the host nonce and new nonce. It then sends the two nonces back to the host. The host checks its nonce and then generates the same session key.

At the end of this protocol segment, the host and smart card have a session key with which to encrypt and decrypt all further messages between them to ensure confidentiality, and they have avoided the security risk of passing the key between them.
The user is asked to enter a PIN and a misuse counter is then decremented to record the number of PIN attempts, in order to protect against a brute-force PIN guessing attack. The host then sends the user PIN to the smart card with a Message Authentication Code (MAC)\(^4\). The smart card checks the MAC and PIN, and returns a code indicating whether it matches the PIN held on the card. If it matches, the misuse counter is reset to its default value.

**Figure 3.5 Sequence diagram for decrement of misuse counters**

Three misuse counters all stored on the smart card protect against unlimited server connection attempts, biometric scans and PIN guesses. They are all set to default values and decremented in the same way (Figure 3.5). The host sends a misuse counter number, a nonce and a MAC to the smart card. The smart card checks the MAC and returns the misuse counter value, the nonce and a MAC. The host checks the nonce and MAC, decrements the value and, if it is non-zero, sends it to the smart card which stores the decremented value. The host then requests the decremented value to confirm that it has been stored correctly in the smart card.

---

\(^4\) A MAC is a hash value calculated from a message which is used to check message integrity. The receiver re-calculates the MAC from the message and compares this with the MAC received from the sender; if they are the same then the message has not been changed.
The remaining protocols below also use the session key, Nonces and MACs to confirm message confidentiality, freshness and integrity in the same way but are omitted for clarity in this description.

3.2.2 Host – scanner protocol

The host - scanner protocol (Figure 3.6) begins with the host asking the user to place a finger on the scanner, decrementing the misuse counter protecting against unlimited scans and then requesting the user’s biometric template from the smart card. This template is a digital representation of a biometric, such as a fingerprint, that is recorded on registration and stored on the smart card for comparison with a live scan of the user’s biological feature. The smart card returns the template and a digital signature of it, allowing the host to confirm the template’s authenticity.

![Figure 3.6 Sequence diagram for biometric scan](image)

The host then opens a session with the scanner (Figure 3.7), sending it the template and a request for a scan of the user’s biometric. The scanner compares the scan it takes with the template and, if it is within a set tolerance, returns a ‘successful scan’ code. The template is then erased from the scanner and the host resets the misuse counter. If the scan fails then the request, scan and response can be repeated up to two more times before the session is closed.
Although the scanner and smart card reader are logically separate, they are implemented physically as a combined scanner/reader device in Viti’s prototype. This device would probably have controller software with which the host would interact so in the prototype the host does not negotiate a different session key with the scanner; it uses the same smart card – host session key.

At this point, the user has been authenticated by three factors: possession of a smart card, a PIN and a biometric.

### 3.2.3 Host – server protocol

The host – server protocol (Figure 3.8) begins with the host asking the user to enter the URL of the server and to confirm that the digital certificate stored on the smart card should be used for user authentication with the server (Figure 3.3). The host requests the certificate from the smart card and decrements a misuse counter protecting against unlimited server connection attempts.

![Biometric match sequence diagram](image)
The host then establishes an HTTPS connection with the server. This connection authenticates the server by checking the server digital certificate’s common name, certificate date, issuing Certification Agency and whether it has been revoked using a test certificate revocation list. The host requests the server to perform client authentication - authentication of the user’s certificate - in a similar way to that of the server’s certificate. If both client and server authentication are successful, the host then resets the misuse counter on the smart card and closes the smart card session.

![Figure 3.8 Sequence diagram for HTTPS handshake](image)

This completes the biometric authentication protocol: the user is authenticated through three factors – possession of the smartcard, the PIN check and the biometric scan check - and the host and server may now securely exchange information over the HTTPS connection.

If the smart card is withdrawn at any point in this authentication then the protocol should terminate, although this is not implemented in the prototype.

---

5 HyperText Transport Protocol (Secure) is HTTP run over a Secure Sockets Layer (SSL) connection.
3.2.4 System operations specification

Working from these diagrams, I specified system operations informally in terms of the pre- and post-condition constraints listed in Appendix C in the style of unit 4 of the Open University's M878 Object-oriented Software Development course. I chose not use OCL constructs to precisely describe system constraints because I originally found it difficult to understand and use.

3.3 Prototype design

I designed a prototypical implementation of Viti’s protocol by considering candidate software architectures and relevant classes to minimise the amount of bespoke design and coding needed. I then selected the most suitable software architecture.

The prototype’s software architecture (summarised in Figure 3.9) is a software-only implementation of the host, scanner and smart card classes where the host would be invoked by a user through a browser to control a protocol with the other classes to authenticate the user. 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.

Figure 3.9 Final prototype software architecture

The design uses the Java Cryptography Architecture (JCA) to encrypt and decrypt data in these messages, and the Java security package for nonces, message authentication codes and digital signatures.
The architecture adds a server as a separate program communicating with the host over an HTTPS connection using the Java Secure Sockets Extension (JSSE). The server-host HTTPS connection interaction includes verification of server and smart card test digital certificates stored in key stores and the checking of a test certificate revocation list.

The architecture also includes user screen dialogues and some exception handling, but excludes data communications and device input/output for the smart card reader and scanner, as no physical devices are used. It also excludes comprehensive error handling and performance features such as support for threads.

3.4 Prototype implementation

I implemented and tested the prototype in Java 6 using the Eclipse 3.2 open-source software development tool. Table 3.1 describes the prototype’s classes and Appendix D lists the approximately 1,300 lines of source code.

<table>
<thead>
<tr>
<th>Prototype Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioSensor</td>
<td>A software representation of the scanner, containing methods to exchange messages with the Host.</td>
</tr>
<tr>
<td>Host</td>
<td>The software controlling the authentication process, containing methods to send messages to the user, smart card, scanner and server, and to receive and process replies. It creates a test certificate revocation list and checks the server certificate against it.</td>
</tr>
<tr>
<td>KeyStoreSetup</td>
<td>A class that checks the correct installation of the Bouncy Castle provider, creates key stores containing test X509 v1 digital certificates for the smart card and server, and creates a trust store and test digital certificates.</td>
</tr>
<tr>
<td>Server</td>
<td>The server application, containing a single method that handles an HTTPS connection with the Host and performs client authentication.</td>
</tr>
<tr>
<td>SmartCardOS</td>
<td>A software representation of the smart card containing methods to exchange messages with the Host.</td>
</tr>
<tr>
<td><strong>Prototype Class</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UserBioscanInstructions</td>
<td>Displays a screen frame asking the user to put a finger on the scanner.</td>
</tr>
<tr>
<td>UserPIN</td>
<td>Displays a screen frame asking the user to enter a PIN.</td>
</tr>
<tr>
<td>UserURL</td>
<td>Displays a screen frame asking the user to enter the URL of the server.</td>
</tr>
<tr>
<td>Utilities</td>
<td>Contains methods to check MACs and Nonces.</td>
</tr>
<tr>
<td>Validator</td>
<td>Implementation of HostnameVerifier to verify the server name.</td>
</tr>
</tbody>
</table>

**Table 3.1 Prototype Classes**

The prototype uses simplified cryptographic techniques to reduce development effort, such as allowing the cipher to generate an initialisation vector instead of supplying one, and using a stream cipher so the plain text and cipher text are the same length. These simplifications should not compromise UMLsec requirements although they do reduce cryptographic strength. In order to create my own test digital certificates, I used the Bouncy Castle JCA provider from Hook (2005).

**3.5 Code validation technique selection and use**

Section 2.7 describes the Java assert statements, JML, Bogor and Prolog code validation techniques considered for code validation of the prototype. JML was selected because:

- Java asserts offer only limited code validation support;
- JML offer rich code validation support, has an Eclipse workbench static checker plug-in and is consistent with the prototype system operation specification approach using pre- and post-conditions;
- Bogor is comprehensive, supports JML and also has Eclipse plug-in tool support, but also appears quite complex and possibly more focused on concurrency checking; and
- there is insufficient time to learn Prolog.
I added JML statements to the Java code using the *Eclipse* workbench and then checked it for static errors using the *ESC/Java 2* plug-in. Appendix D includes these JML statements. JML exception conditions were omitted because the prototype does not support comprehensive error handling.

### 3.6 Summary

This chapter has described a research method for evaluating the use of UMLsec and for trying to relate code back to its UMLsec specification. This method specifies the biometric authentication system introduced in Chapter 2 using UMLsec, then designs and implements a prototype of the system in Java, and finally selects and uses a technique for code verification. The next chapter describes the evaluation of the UMLsec approach using this research method and chapter 5 describes the use of JML to verify the prototype’s code against its specification.
Chapter 4  Evaluation of the UMLsec approach

4.1 Introduction
This chapter describes the evaluation of the UMLsec approach when used in the design of the prototype biometric authentication protocol, and how easy it was to implement the prototypical system from the UMLsec specification. The results are structured using the first three steps of the research method:

- developing UMLsec diagrams to describe system security requirements – section 4.2;
- designing the prototype – section 4.3; and
- implementing the prototype design using Java – section 4.4.

4.2 UMLsec diagram development
This dissertation assumes that the cryptographic notation used in the sequence diagrams is part of the UMLsec approach, although distinct from the UMLsec notation. The cryptographic notation is succinct and unambiguous but very difficult to understand without substantial study and would probably be indecipherable to a non-specialist. It therefore requires a complementary summary textual description for the non-specialist reader, although I recognise the risk of the text becoming inconsistent with the diagrams if any changes are not applied consistently to both.

I found UMLsec and the cryptographic notation adequate for describing security requirements and only three requirements could not easily be described. The specified system is vulnerable to an established connection being misused by insiders if the user moves away from the host device without logging off. This could be addressed by specifying a connection timeout value but I could not express this in UMLsec. I also could not specify a requirement in UMLsec for protocol termination if the smart card was removed at any point. UMLsec includes LAN, wire, smart card, POS device, issuer node stereotypes but does not include other useful types of communications links; for example, the scanner USB port into which a USB smart card plugs could be described with a <<connection>> stereotype, which was used by Schmidt (2004).
The meaning of the UMLsec <<secure dependencies>> stereotype is unclear. Data sent over the untrusted dependencies between the host and smart card, scanner and server requires the preservation of security requirements described using the <<data security>> and <<critical>> stereotypes and their associated tags in the respective classes. For example, the secrecy of the biometric template expressed in the \{secrecy=\texttt{t}_{\texttt{user}}\} tag on the Host and Smart Card classes (Figure 3.2) must be preserved when it is sent from a Smart Card to a Host object. It is unclear whether the <<secure dependencies>> stereotype (sec. 2.2.7) should be used in this situation; it appears to relate to messages rather than to data attributes and, following the example of Jürjens (2005), I have not specified the security requirements of messages. This stereotype is not used by Jürjens in many similar situations so I have not used it in the prototype specification. This requirement should be expressed explicitly by extending <<secure dependencies>> to data items.

I used general-purpose graphics software for the UML diagrams but this became inadequate to support the growing complexity of these diagrams and to keep information consistent between them. The use of UMLsec on a complex system would probably be impractical without a CASE tool with support for UMLsec and the ability to highlight or filter out layers of information.

The value of UMLsec diagrams is diminished if they are not maintained beyond the specification stage. The class diagram (Figure 3.2) did not include the server as I initially perceived it to be outside the scope of system. However I later found a server security requirement which could not be expressed so I introduced a Server class, but neglected to add it to the class diagram until the end of the dissertation period because of my focus on implementation. In an industrial situation, this would have prevented me from re-verifying the consistency of the changed security requirements using the UMLsec toolset.

This is an example of the general issue that systems analysis, design and implementation are often iterative; my developing understanding had caused me to revisit earlier assumptions, which often happens in real systems development. Functional requirement changes also frequently force design iterations.
More mistakes and omissions may be introduced into systems during changes than in initial design, especially if they are rushed to meet a deadline, as focus is often on the detail of each change rather than on the effect on other aspects of the system, such as security. Unfortunately changes are often not applied to system specification documentation because of time, manpower or budgetary pressure, which would prevent UMLsec automated checks being used to confirm that changes do not compromise security, and would also prevent further code-to-specification verification. A project must therefore commit to updating the system specification and UMLsec notation for every subsequent change if it is to fully benefit from its initial investment in applying the UMLsec approach in initial design.

The level at which security requirements are specified should take account of the design to avoid redundant effort. For example, the host – server dependency is implemented using HTTPS: I initially described the full SSL protocol in UMLsec but derived no value from this, apart from a better understanding of the protocol, since there was no need to explicitly code these messages as they are handled by HTTPS JSSE classes.

This research uses version 1.5 of UML because most of the source material uses this version. However UML is continually being enhanced and version 2.1.1 is now current. Jurjens (2005) assesses the effect of UML 2.0 on UMLsec as minor because most of the new model elements and deletions do not affect security engineering. However future versions of UML might have more of an effect, so UMLsec would need to be maintained consistently with UML if organisations are to have confidence that an investment in adopting UMLsec will provide long-term benefits.

I described system operations using an OCL structure (Appendix C) but I did not formally describe all system constraints because of OCL’s complexity. This did not cause any development problems because I acted as both the designer and developer but in an industrial-scale project, where these roles are often separate, more formal constraint documentation would be needed to communicate information. I should have made an earlier decision to evaluate JML so that I could have then used JML for design. Similarly, in an industrial project, an early decision on the development language would then allow a specific specification language like JML to be used rather than the generic OCL.
4.3 Prototype design

UMLsec is a security requirements specification approach rather than a security design method so it does not define how to satisfy security requirements and a security designer is still needed. UMLsec also does not describe the level of security to be specified, so the designer is not given explicit guidance on, for example, the cryptographic algorithms or key strengths necessary, or the quality of randomness required for freshness.

Java was selected as the development framework because it is platform independent and can be ported to mobile devices. However I eventually chose not to adopt Viti's software design since this only implemented part of the required system and was not easily converted to Java. I initially tried to convert Viti’s software architecture, which he had implemented in C, to Java. In Viti’s design, Dynamic Link Library (DLL) functions manage cryptographic interaction with the smart card using the PKCS#11 cryptographic token interface standard. Viti interposes an extra PKCS#11 DLL library between a controlling browser and the real DLL, and modifies the extra logon DLL to additionally call a biometric match of the template against a live scan. However this architecture is not easily directly replicable in Java, since PKCS#11 DLL functions are written in C. Smart card reader vendors provide PKCS#11 DLL functions that are invoked via Java cryptographic operations and smart card APIs, such as the OpenCard Framework (OCF) but, since the prototype does not use smart card reader hardware, PKCS#11 functions did not need to be implemented in the prototype.

Viti’s biometric match is also only part of the overall biometric authentication system: Viti does not describe how this is controlled from the browser, the host-smart card or host-scanner security protocol, or how the HTTPS connection to the server is controlled. I found that a controlling client program invoked by a browser was necessary to manage the protocol. This program could then call the biometric match function directly after the smart card logon.

The JSSE supports an HTTPS connection, allowing the connection to be programmatically configured to satisfy security requirements such as client authentication, but without having to code each SSL message exchange. Although I considered other products, the final software design uses JSSE because its use was well documented, such as in Hook (2005), which lowered the risk of encountering development problems.
The system requirements in Viti’s paper do not make it clear whether the server is in the system’s scope. This point is important since it affects the security of the HTTPS connection; a server outside the system might weaken security by negotiating a reduction of the cryptographic strength of the SSL connection during the protocol negotiation stage, for example by only supporting an earlier version of SSL or by not supporting client authentication. A server selected from an approved list of servers that supported all the features required, such as client authentication, would avoid this risk, so I included the server in the system which required changing the specification. The system specification needs to be as clear and complete as possible to reduce the cost and delay of changes such as this, so it should be reviewed by a security designer before being approved.

The UMLsec approach defines a threat model (section 2.2.1) describing an adversary. I had initially assumed a default adversary with no access to the trusted wire link between the smart card, scanner and host but I found that scanner and smart card reader vendors implement a cryptographic protocol to the host, implying the link is untrusted. I therefore changed my assumption of a default adversary 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 cryptographic protocols to preserve security requirements. This emphasises the value of UMLsec's threat model in forcing attention on the nature of the security threat.

4.4 Prototype implementation

The specification and design proved to be a sound basis for implementation, particularly the sequence diagrams, and the main implementation issues were due to my lack of Java cryptography knowledge.

As the development progressed, I made improvements to the implementation, for example by changing a text string instruction to the smart card to an integer value that is easier to process. These improvements required changes to the specification. To minimise these changes, the protocol specification should be influenced by the capabilities of the development language; it is inefficient to specify a protocol independently of a language that then proves difficult to implement if security requirements can be met more efficiently by designing with language features in mind. Using language features would also reduce the risk of introducing security flaws by reducing the amount of bespoke coding needed.
The development language should therefore be selected early in the development process before specification is complete. Either the designer would need to know language security features well or the developer should be allowed to change the protocol, with the agreement of the designer, to fit language facilities. The design and requirements documentation would need to be kept consistent with any changes.

4.5 Summary
This chapter has described my evaluation of the UMLsec approach in the design of the prototype biometric authentication protocol and the implementation of a prototypical system from the UMLsec specification. The next chapter describes the use of JML to try to verify the prototype’s code against this specification.
Chapter 5  Code verification using JML

5.1 Introduction
A developer might introduce security flaws during coding because of omissions, errors or deliberate changes so a way of verifying the consistency of the code against its specification is desirable. This chapter evaluates the use of JML to try to verify the prototype’s code against its UMLsec specification.

Section 5.2 examines the results of using JML in parts of the prototype. Section 5.3 then comments on the JML tools used. The usefulness of JML patterns and JML specifications is reviewed in sections 5.4 and 5.5 respectively. Section 5.6 then discusses the results of a manual consistency check of the code against the specification. Finally, section 5.7 summarises some limitations of the JML approach.

5.2 JML code validation
I firstly added a JML contract for the Utilities.checkMac method in the Java code (Appendix D). This method checks whether the MAC calculated from a received message is the same as the MAC in the message - if it is then the message has not been modified.

The method handles different message types, each with a different length, but I could not easily write JML to validate the message format because I had not coded the method type as an argument. I could not also easily verify the result of the check by testing a return code since the method does not return a value; it errors and terminates the protocol if the MACs do not match. However I verified the check by comparing each byte of the two MACs using a JML \forall expression:

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

The checkMac method was an addition during implementation to consolidate a number of blocks of similar code. It demonstrates that JML can be difficult to apply unless the method has been designed with JML verification in mind.
I then tried to apply JML to the `Host.v` method, which calculates a session key using smart card and host nonce values, and the `Host.find` method, which derives a temporary symmetric key and the storage address of a session key. I found both difficult because the methods do not return a value and were so small that any JML would duplicate the Java code.

Since Cheon and Perumandha’s (2006) JML call sequence clause is not yet available, I used Agarwal et al’s (2006) finite state machine model for checking that messages are sent in the correct sequence in `Host`. Static integers are declared defining the message send before each method start and ends:

```java
/*@ public static final ghost int
@ INITIAL = 1,
@ RESET = 2,
@ ASKZ = 3,
@ ACK = 4,
......
//@ public ghost int state = INITIAL;
```

The JML ghost field `state` is assigned one of these values at the end of each method; for example, at the end of the `checkPIN` method when the SETFBZ message has been sent by the Host:

```java
//@ set state = SETFBZ;
```

The JML specification for the method can then test the pre- and post-condition using this variable, for example `checkPIN` must begin after an ACK has been set and end by sending a SETFBZ:

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

However, each method handles several messages and I 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 I written these methods at a lower level of granularity to each handle only one message exchange then this would have been possible.
To confirm this, I extracted code from the `Host.negSessionKey` method into separate `Host.setIdsc` and `Host.initCipher` methods. I was then able to verify more of the message sequence. This again suggests that a system needs to be designed in a style consistent with JML to extract most value from it.

The JML \texttt{\fresh} expression asserts that objects are freshly allocated and were not allocated in the pre-state. This supports the freshness security requirements implemented by nonces in the prototype well. For example, the JML contract for `SmartCardOS.askZ()` uses the expression:

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

to verify that the \texttt{zSc} random number is freshly generated.

I investigated coding 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. I 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. I 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.

JML supports storage integrity and confidentiality within a method using several keywords and expressions. JML supports integrity using the `assignable` keyword, which constrains the fields to which a method can assign, and the \texttt{\not_modified} expression, which asserts that values of the named field are the same in the post-state as in the pre-state. JML supports confidentiality using the `accessible` keyword, which constrains the fields read by a method. If nothing is specified using these JML keywords then all fields are accessible and assignable.
I could have used these JML contracts to specify that SmartCardOS methods could not assign to, or read the field containing the PIN, except for a read by valPin. However this would have been cumbersome because there is no JML keyword meaning ‘not accessible’, so JML would have needed to have been added to every SmartCardOS method to specify all the fields that were accessible.

Gross omissions in code should be detected when trying to write a relevant JML contract as long as the code is not too monolithic. For example, the omission of setFBZ (1) and Close message implementations was detected whilst examining how JML could check the message sequence, which demonstrates that the process of applying JML also serves as a code walk though.

5.3 JML tools
I used the ESC/Java 2 plug-in for the Eclipse workbench to statically check the JML and highlight syntax errors in the source code editor. The tool was easy to install and use but had limited documentation. It identified some JML errors but I had doubts about the completeness of its checking as it consistently generated the same error message for all prototype classes, which may have masked other errors. I could not eliminate this message because there was no explanatory documentation.

ESC/Java 2 does not support runtime checking so I tried to install the JMLEclipse runtime assertion checker plug-in to help verify the prototype but it did not install properly after several attempts. The tool does not appear to have been maintained for new versions of Eclipse and its moribund development project appears to have been superseded by a common JML tools project. I installed the latest version of these common tools, which are not yet integrated with Eclipse, and used the jmlc compiler to parse and type check the prototype source code. This identified ten additional JML errors missed by ESC/Java 2, although none revealed new security flaws.

After obtaining a clean jmlc compilation, I re-checked the code with ESC/Java 2 and it additionally reported errors in core Java classes within each prototype class. The likely cause of these errors is the new language features of Java 6 in which the prototype is coded, since ESC/Java 2 only supports Java 1.3 and Java 1.4. Its user manual describes a similar error and records the developers’ intention to support Java 5.
5.4 Confidentiality and integrity JML patterns
Warnier (2006) introduces JML specification patterns for confidentiality and integrity (sec. 2.7.2). These were applied to the prototype to see how useful they might be in 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 are independent of the values of all confidential (high-security) fields in the pre-state. `SmartCardOS.valpin()` has a high security input parameter `cipherText` and leaves the value of `plainText` in the post-state with its decrypted value, so losing confidentiality. An adversary would not easily be able to access it because it is not stored or transmitted, although the value of `plaintext` should be erased at the end of the method.

Warnier defines a similar JML specification pattern for integrity where the values of high security variables in the post-state are independent of low security variable values. Using this pattern, I could not find a dependency that caused a loss of integrity.

Overall, I found these patterns to be difficult to apply because many of my methods were too large and therefore too complex; they would have been more useful with smaller methods.

5.5 JML specifications
JML specifications have been written for some Java classes, although only a few are cryptography classes (Agarwal et al 2006) (sec. 2.7.2). Agarwal et al claims that these specifications provide a more precise understanding of the behaviour of the classes than the javadoc comments which, they assert, are ambiguous and imprecise. If so, this might reduce security flaws caused by using these classes incorrectly, such as with invalid pre-conditions or in handling post-conditions incorrectly, and it might identify such errors during run-time assertion checking.

To investigate this claim, I compared the normal behaviour JML specifications of four methods of the `Signature` Java class used in the prototype to the javadoc comments for these methods: the results are summarised in appendix E.
Overall, the JML is a little more precise than javadoc over these four methods but it requires much more time to understand and only three cryptography classes currently have JML specifications. The ability to automatically check JML contracts at run-time checking would be of little use for these specifications since nearly all pre-conditions are also handled by Java exception handling and the post-conditions could be relied on to be satisfied as the Java methods are extremely unlikely to contain errors. I therefore found JML a supplement to javadoc, which should be made more precise and unambiguous.

5.6 Manual code validation check

After applying JML to selected methods, I then 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.

The security requirements defined using UMLsec notation in the class and deployment diagrams should map to protocol components in the sequence diagram. For example, the host – biosensor link integrity requirement implied by the Host and Biosensor attributes {integrity} tags in the class diagram (Figure 3.2) are supported by MACs and nonces to check that messages have not been altered or substituted; for example, in the scanResult message, where $Z$ is a nonce (Figure 3.7). JML does not assist this UMLsec-to-protocol component check but I would expect it to be automated in the UMLsec static checking tools discussed in sec 2.3; if not, then this would be an area for tool development.

Each protocol component should then map to code in the prototype. For example, the scanResult message is generated in the Biosensor.scanMatch method and verified in Host.bioMatch, which calls Utilities.checkMac and Utilities.checkNonce to check the MAC and nonce respectively.

This protocol-to-code mapping check would have been easier had I consistently designed and written one method to process each message in each class. In an industrial setting, 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 was then written in JML, it could be automatically checked statically and at runtime to complement a manual check.
This mapping check revealed several security flaws and weaknesses. Firstly, the integrity of the retFBZ message was not confirmed in the code because a call of Utilities.checkMac in Host.getCounterValue had been omitted. The omission of this MAC check would probably have been detected in coding a suitable JML contract for previously written Java code.

Secondly, the prototype did not implement some requirements. The getCert and retCert messages (Figure 3.8) that request and retrieve a digital certificate from the smart card were omitted. I had coded a temporary fix to read the smart card certificate directly from a key store to get the prototype working and then omitted to re-code it later.

The prototype also did not implement secure storage of the smart card PIN or biometric template to protect their confidentiality or integrity, as required by the \{secrecy = pin_{sc}, t_{user}\} tag in the SmartCardOS class (Figure 3.2). Instead, the SmartCardOS.valPin method merely returns a code indicating a successful check of the PIN. This was deliberate because of insufficient development time. However neither of these flaws would have been detected by a JML contract since the methods are either missing or return correct values.

The MAC and cipher both use the same key, which is weak since if the key is broken then the whole communication is totally insecure: different keys should be used for different types of cryptographic operations. This is a design weakness in the sequence diagram, which specifies the same key: the code implementation is correct so it would not have been revealed by JML. This weakness shows that security design expertise is still needed even with the use of UMLsec and JML.

The prototype does not implement Viti's protocol correctly when a new session key is generated. Viti’s protocol generates a session key from a combination of two nonces and a key stored in the smart card. The prototype generates a new SecretKeySpec in the $v$ method using the two nonces ($k_{sc} := v_{sc}(Z'_{sc}, Z'_{lh})$ in Figure 3.4) but omits the smart card key. The extraction and return of the session key address by the Host is therefore redundant. The specification is incorrect but the prototype is implemented from it correctly so JML would not have revealed this fault.
The implied integrity requirements for the smart card Id id\textsubscript{sc} in the Smart Card and Host classes (Figure 3.2) are not implemented in the ATR message (Figure 3.4) as there is no MAC check. This is a deliberate omission because there is no security flaw if the id\textsubscript{sc} is modified; the key and storage address extraction would either fail or the address would not be recognised by the smart card, but in either condition the protocol would terminate. This is an example of a change introduced by the developer from an improving knowledge of the protocol that might be revealed by static checks on the consistency of the UML\textit{sec} and protocol, but would not be revealed by JML since it is largely based on the sequence diagram.

A variation of this issue would be an undocumented change introduced by a developer with the intention of improving the protocol but that actually introduces a security flaw. These might be revealed by JML contracts based on the original specification if the change alters the interface or behaviour of methods.

The specification omits the address\textsubscript{k} attribute in the Host and Smart Card class definitions and its security requirements, although they are correctly specified in the ACK message. This is an inconsistency of the specification.

The MACs in messages are not confidential since they cannot be used to obtain the original message. Their encryption in the specification is therefore redundant.

These flaws and inconsistencies reflect real life: security specialists may leave inconsistencies and flaws in specifications and developers might sometimes not fully implement requirements because they forget, they make a mistake or they deliberately omit them to meet development deadlines. The success of my manual check in detecting these flaws argues for a rigorous manual code review against the specification as well as JML contracts.
5.7 Limitations of the approach
Successful specification and code verification does not mean that a system is free from security flaws as they may have been introduced in areas not revealed by code verification, such as in:

- products;
- design;
- shared hardware or software;
- manual operational processes; or
- JML code derivation.

The system may be implemented in part by products for which the source code is not available, rather than in bespoke code. A designer would then need to rely on the certified conformance of those products to security standards, such as the Common Criteria ISO standard for IT security. For example, if a system uses proprietary software to communicate with a scanner then this part of the system could not be checked using code verification. A designer might therefore favour open source products, since their security could be to some extent verified through source code analysis.

An implementation may introduce flaws in the user interface design that are not detectable by code analysis. For example, if a PIN entry screen displays the actual PIN entered instead of masking it then the secrecy of the PIN would be compromised but code analysis would not detect this.

The design of the system may weaken security in other areas. For example, the use of the HTTP CONNECT method would prevent intermediate firewalls along this connection examining this traffic and screening it because it instructs them to pass it on without examination.

JML does not support system design in terms of describing the algorithms with which contracts are implemented such as, for example, whether storage confidentiality and integrity are implemented using a KeyStore or a read-protected file. Weaknesses could be introduced into these algorithms, such as an easily broken file key. However this is consistent with JML’s purpose as a behavioural interface specification language, specifying the behaviour of a method in terms of a set of state transformations that it can perform, rather than in terms of the algorithm or process.
The design and implementation of associated administrative software would need to preserve security requirements otherwise ‘back door’ security flaws would be created. For example, the software that registers users and configures smart cards must preserve the security and integrity requirements of users’ biometric templates.

The JML and UMLsec approaches would not detect security flaws in manual operational processes such as user or smartcard registration needed for the operation of a system, unless it could be extended to them if they were described in UML.

A managed services organisation may run the system on a hardware architecture designed to optimise efficiency, such as virtual hosts where several organisations’ server software and data resides on the same physical hardware. A special case of this is where the application has been specifically designed to share the use of code and database by multiple organisations, such as configurable sales applications provided by Internet Application Service Providers. Here, there is a risk of security flaws that could be exploited by legitimate application users from other sharing organisations as well as by an external adversary.

JML does not directly verify code against a UML specification. JML verifies invariants and pre- and post-conditions, which are derived from the UML specification as, for example, in appendix C and then expressed in JML. This derivation itself may introduce errors and omissions that create security flaws, and JML coding may omit checks that would have otherwise revealed a security flaw.

**5.8 Summary**

This chapter has described the results obtained from using JML to try to verify the prototype’s code against its UMLsec specification. It has reviewed the manual check of the code against the specification, and has looked at the usefulness of JML tools, patterns and specifications. It has finally examined some limitations of JML. The next chapter draws conclusions from this research and makes recommendations for future research.
Chapter 6  Conclusions

6.1 Dissertation review
This research has investigated how effective UMLsec is when used in the development of secure systems (sec. 1.2). The objectives have been:

- to evaluate UMLsec when applied to the design of a biometric authentication protocol;
- to investigate how easy it is to implement a prototypical system from this UMLsec specification; and
- to examine how this implementation could be related back to its specification to verify that the implementation is secure.

This chapter draws conclusions from the results described in the previous two chapters, and makes suggestions for further research.

6.2 UMLsec evaluation
The research used two components of the UMLsec approach: the UMLsec notation to specify security requirements in class and deployment diagrams; and the cryptographic protocol notation to specify a message protocol between the system’s components in a set of sequence diagrams.

The UMLsec approach enabled me to specify security requirements clearly and succinctly in seven diagrams. The threat model in the approach was useful in helping to resolve an issue with the scope of the prototype. The approach supported the biometric aspects of the prototype well, such as the scanner dialogue and biometric template confidentiality, integrity and authentication requirements, and I did not encounter any specific issues in specifying these biometric security requirements. This approach should incur lower training and usage costs than previous formal methods but should be enhanced in some areas if it is to appeal widely in industry.
I had difficulty in understanding the meaning of some stereotypes and tags, and of the cryptographic notation. I needed to make a very considerable commitment of time and effort to understand the approach, which would not be cost-effective in industry. The approach therefore needs an easier, non-mathematical description, more examples and patterns, training courses and perhaps an accreditation system to maintain standards. A textual summary of security requirements and protocols would still be required for those who cannot read the notations.

I had difficulty in maintaining UML diagram security notation consistently. CASE tool support for the UMLsec approach is needed to facilitate this maintenance, as the approach would probably be impractical to use in a large system without it because of the complexity of diagrams.

There were a few minor requirements such as timeouts, protocol termination and device connections that I could not describe, and which the approach should be extended to cover.

The most serious design problem I encountered was the design of the software architecture, which was caused by trying to adapt Viti’s design, and then to lack of knowledge of both software architectural options and Java cryptography. These were not related to the UMLsec approach, which does not address security design. However the approach would be more valuable if it were positioned as an integrated part of a systems development security methodology or a general systems development methodology covering specification and design.

UMLsec may have rather limited applicability to specialist protocols. The prototype implements a proprietary protocol between the host, smart card and scanner. However the development of such protocols requires specialist skills, funding and time, and runs the risk of leaving security flaws; it would be much more cost-effective to use SSL/TLS where possible, as it is free, industry-standard, widely supported and well-examined protocols supported by Java. JML would still be valuable to verify the correct implementation of the specification but the verification of security protocol code would be much reduced.
6.3 Implementation from the UMLsec specification
The implementation of the prototype from the diagrams, system operations specification
and architecture was relatively straightforward because the protocol described in the
sequence diagrams was on the same level of abstraction as the code, and was
sufficiently unambiguous and precise to code directly from it.

The main implementation problem was lack of knowledge of the Java cryptography
classes, which slowed development and may have introduced security weaknesses such
as the cryptographic key strength used. I also introduced minor changes into the
specification to make the implementation in Java easier, but these were not related to
the UMLsec approach, which does not address implementation.

6.4 Implementation verification
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 little 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 during
design.

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 but this may not provide sufficient industry assurance without a
commercial support package.

Although JML does not directly support message or storage confidentiality or integrity
verification, I was able to code contracts to indirectly check these security requirements.
However JML patterns for confidentiality and integrity suggested by Warnier were
difficult to apply because many 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 cumbersome and error prone, so I would endorse Cheon and Perumandha’s (2006) proposal 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 javadoc documentation but are not a substitute for it being made more precise.

A subsequent manual check of the prototype’s code against the UML specification successfully identified a number of security flaws and weaknesses. Writing JML contracts effectively enforces parts of this check by focusing attention on the code and specification, but there were still flaws present and specification inconsistencies that would not have been revealed by JML. It would require considerable time and a good knowledge of the specification, application code and security to do a manual check thoroughly so it could not be relied upon in an industrial situation with time and cost pressures. However a check could be made easier 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.

The approach does not eliminate security flaws since they could be 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: they 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 de-skilling and diminishing their role, and so resist its introduction. To avoid this, they should therefore 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 walk though.
The use of 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 using none of these techniques 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, design and implementation style guide for JML, and 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 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; smart card biometric authentication protocols would be one such area because of the impracticality of resolving errors in applications on issued smart cards.

6.5 Future research on enhancing tool support

JML tools verify Java code against a JML specification derived from the UMLsec specification, rather than from the UMLsec specification itself. Deriving this JML specification manually requires skill, work effort and time, and might still omit checks that could reveal security flaws. Future research could therefore investigate how JML specifications might be generated directly from an XMI form of a UMLsec sequence diagram describing a cryptographic protocol. Such a tool would improve the efficiency, accuracy and completeness of JML coding. This research could then examine ways to verify the JML specification against the UMLsec.

Figure 6.1 describes the sequence of the prototype’s development deliverables. Most of this sequence is supported by tools: the UMLsec analysis tools described in sec. 2.3 are claimed to verify the consistency of UMLsec diagrams; and the JML tools described in sec. 2.7.2 verify the Java code against its JML. However the generation and verification of JML specifications from the UML sequence diagrams is not supported.
Figure 6.1 Tool verification support

A generation tool could be based on the correlation between the components of a sequence diagram and those of a JML specification. Figure 6.2 shows a protocol dialogue described by a sequence diagram where an object A sends a message \( m_1 \) containing concatenated arguments \( a_{m1}^1, a_{m1}^2, \ldots \) to an object B. B then replies with a message \( m_2 \) containing similar elements. Condition \( c_{A-\text{pre}}^1 \) applies before \( m_1 \) is sent and condition \( c_{A-\text{post}}^1 \) applies afterwards; similar conditions might apply before, and after the message is received by B, and for its reply.

Figure 6.2 Sequence diagram components
The draft JML specification in Figure 6.3 below could be constructed from these components.

```java
Class A {
//@ model import org.jmlspecs.models.*;
byte [] m1, m2;
//@ public static final ghost int
//@ INITIAL = 1,
//@ m1SENT = 2,
//@ m2SENT = 3;
//@*/
//@ public ghost int state = INITIAL;
//@ public normal_behavior
//@ requires state == INITIAL;
//@ requires cal-pre;
//@ assignable \everything;
//@ ensures state == m1SENT;
//@ ensures cal-post;
//@*/
public byte[] sendm1 () {
    ...
    //@ set state = m1SENT;
    return m1;
}
```

**Figure 6.3 Draft JML specification generated from sequence diagram components**

This approach constructs sender and receiver methods: the sender methods would return message as byte arrays that a controlling program would then send, and the receiver methods would be invoked on receiving a message.

The proposed tool would have limitations. Variable types would either need to be read from the class diagram or re-declared in the sequence diagram. JML describing exceptional behaviour would need to be manually added. The tool would need to cope with omitted sequence diagram conditions, messages with no reply and more complex sequence diagram structures, such as branching control flows. The tool would also not be able to specify JML frame conditions.
UML conditions for integrity, confidentiality and freshness do not correlate with the JML expressions for these conditions so they would be difficult to generate. For example, the UML condition from Figure 3.5:

\[ \text{arg}_{sc,4,3} = \text{Mac}_{ksc}(\text{arg}_{sc,4,1} :: \text{arg}_{sc,4,2}) \]

would, from sec. 5.2, be described in JML in an expression of the form:

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

Such UML conditions could be handled by generating JML informal descriptions but these would not be executable. Alternatively, a CASE tool supporting UMLsec might be extended to also support the generation of JML specifications; this CASE tool could then generate JML expressions from user interface commands rather than from reading the UML sequence diagram, which might be an easier way of generating executable integrity, confidentiality and freshness JML expressions.

### 6.6 Other future research

UMLsec is an approach to specifying and verifying security requirements in systems development, but it is not a systems development methodology itself. However it might be incorporated into a development methodology to facilitate its wider adoption, which suggests research to see how well it would integrate with a systems development method that uses UML, such as the Rational Unified Process or Feature Driven Development.

Jürjens (2005) makes the point that one needs to make sure that features in a modeling language that express security properties on the design level need to map to system constructs on the implementation level that provide these properties. For example, he did not use UML package visibility features in UMLsec because they do not appear to be generally implemented in a security-aware way. Security flaws are less likely to be introduced during coding if developers can follow guidance on how to implement UMLsec security primitives in the major development languages. Future research could therefore develop this guidance by mapping UMLsec to features of the Java and .NET languages frameworks.
Jürjens (2005) suggests the use of patterns to re-use standardised solutions to recurring security design problems. He gives an example of a pattern for transmission over a secure channel. Further work could be to collect security design problems expressed in UMLsec and their solutions together in an easy to reference way. This might incorporate Miller’s (2001) mirror image and session façade patterns for representing external entities in sequence diagrams.

UMLsec might be extended to support legislative security requirements, for example from the UK Data Protection Act 1998 which imposes personal data requirements such as retention of personal data no longer than is necessary for the purpose for which it has been collected. This would make UMLsec more comprehensive and avoid such requirements being overlooked. However it would not be possible to verify such requirements in code and some might need to be implemented in manual processes, for example to ensure that physical data storage media such as CDs are stored and transported securely. This would suggest the use of UMLsec in business processes specifications as well, which would be consistent with the extension of UML into business process Modeling proposed by Eriksson and Penker (2000).

JML statements could also be developed to make specifying security features such as confidentiality and integrity easier, for example, with a `\not_accessable()` expression to match the current `\not_modifiable`. 
References


Bibliography


Open University (2000) M878 Object-Oriented Software Development, Milton Keynes, Open University

Programming with Assertions, [online]


Tielens J., Using Custom Attributes for Validation, [online]
http://www.microsoft.com/belux/msdn/nl/community/columns/jtielens/attributes.mspx
[accessed 20 April 2007]
Index

ATP, 11, 12, 15, 66
authenticity, viii, 1, 6, 8, 13, 22, 26
AutoJML, 18, 61
Biometric authentication, 3, 13, 21, 22, 66
Bogor, 15, 18, 19, 20, 31, 59
CASE, viii, 4, 34, 50, 56
class diagram, 22, 34, 44, 55
data security, 6, 8, 9
deployment diagram, 21, 22
Eclipse, 18, 30, 31, 32, 42, 61
ESC/Java 2, ix, 32, 42, 51, 61
fair exchange, 6, 7
formal methods, viii, 2, 3, 49
freshness, 13, 22, 26, 36, 41, 56, 86
guarded access, 6, 10
HTTPS, 28, 30, 35, 36, 37, 65
integrity, viii, 1, 2, 6, 8, 9, 13, 18, 22, 25, 26, 41, 43, 44, 45, 46, 47, 48, 49, 51, 56, 57, 67
Java assert, 12, 15, 31, 41
javadoc, 5, 17, 43, 44, 52, 105, 106
JCA, 29, 31, 67
JML, viii, ix, 5, 15, 16, 17, 19, 20, 31, 32, 39, 40, 42, 43, 51, 58, 59, 60, 61
JML specification, 17, 18, 40, 43, 53, 54, 55
jmlc, ix, 17, 42, 51
jmldoc, 17
jmlunit, 17
JSSE, 12, 30, 35, 36, 67
JUnit, 19
no down-flow, 6, 10, 18
OCL, ix, 6, 7, 18, 29, 35, 58, 67
penetrate and patch, 1, 3
PKCS#11, 36, 61
Prolog, 12, 15, 19, 20, 31, 62
provable, 3, 6, 7
rbac, 6, 7
secrecy, viii, 6, 8, 9, 18, 22, 34, 45, 47
secure dependency, 6
secure links, 6, 8, 9
SSL, 12, 13, 28, 35, 36, 37, 50, 62, 68, 69, 71, 82, 91
TLS, 4, 12, 50, 62, 68, 82, 91
UML, viii, 2, 3, 5, 6, 7, 8, 10, 11, 13, 21, 22, 48, 56, 57, 58, 59, 61, 64, 65, 68
### Appendix A Cryptographic notation and variables

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>::</em></td>
<td>concatenation of variables</td>
</tr>
<tr>
<td>head(_)</td>
<td>head (i.e. the first term) of a concatenation</td>
</tr>
<tr>
<td>{_}k</td>
<td>encryption, using the symmetric key k</td>
</tr>
<tr>
<td>Dec&lt;sub&gt;k&lt;/sub&gt;(_)</td>
<td>decryption, using the symmetric key k</td>
</tr>
<tr>
<td>Sign&lt;sub&gt;k&lt;/sub&gt;(_)</td>
<td>signing</td>
</tr>
<tr>
<td>Ext&lt;sub&gt;k&lt;/sub&gt;(_)</td>
<td>extraction from a digital signature</td>
</tr>
<tr>
<td>Mac&lt;sub&gt;k&lt;/sub&gt;(_)</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>Hash(_)</td>
<td>hashing</td>
</tr>
<tr>
<td>arg&lt;sub&gt;h,5,1&lt;/sub&gt;</td>
<td>the first argument of the fifth message to be received by object h</td>
</tr>
<tr>
<td>fst(<em>), snd(</em>), thd(_)</td>
<td>the first, second and third arguments of a concatenation</td>
</tr>
<tr>
<td>_ $\in$ _</td>
<td>is a member of a set</td>
</tr>
<tr>
<td>h'</td>
<td>the returned value of a variable h sent to another party</td>
</tr>
<tr>
<td>k, k&lt;sup&gt;-1&lt;/sup&gt;</td>
<td>public and private asymmetric keys</td>
</tr>
</tbody>
</table>

| Table A.1 Cryptographic notation used in UML sequence diagrams |
address Address within smart card storage of smart card key.
cert_{sc}, Smart card certificate.
certValStart The certificates start date of validity.
certValEnd The certificates end date of validity.
Date The current date.
FBZ_1 Misuse counter 1, which protects against unlimited HTTPS connection attempts (acronym of Fehlbedienungszahler from Schmidt (2004)).
FBZ_2 Misuse counter 2, which protects against unlimited biometric scan attempts.
FBZ_3 Misuse counter 3, which protects against unlimited PIN guesses.
issuingCA The Certification Agency issuing the certificate.
k_{sc}, k_h Session keys, initially temporary and then recalculated.
K_{sc}, K^{-1}_{a} Public and private keys of smart card administrator used for signing biometric template.
K_{sc}, K^{-1}_{sc} Public and private keys of smart card certificate.
K_s Server’s public key.
matchTol The maximum allowable difference between the template and live scan.
m The result of the scan
pin_{sc}, pin_h PINs stored on smart card and entered by user.
return code return code: 0 indicates success of PIN validation.
S_h URL of server (entered by user).
S_s Common name of server (from server certificate)
trustedCAs The Certification Agencies trusted by the host.
t_{user} Biometric template stored on smart card.
Z_{hs}, Z_s Random numbers, aka nonces (acronym of Zufallszahl from Schmidt (2004)).

Table A.2 Variables used in UML diagrams
### Appendix B Glossary of terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>ATP</td>
<td>Automated Theorem Prover</td>
</tr>
<tr>
<td>Authentication</td>
<td>A check of whether a user is who he claims to be.</td>
</tr>
<tr>
<td>Authentication factor</td>
<td>A credential with which a user can authenticate himself: something known (e.g. a password), something possessed (e.g. a token) and/or a biometric (e.g. a thumbprint).</td>
</tr>
<tr>
<td>Authorisation</td>
<td>The granting of privileges to a user to carry out functions and access data.</td>
</tr>
<tr>
<td>Biometric authentication</td>
<td>The use of the encoding of a user’s biological characteristic, such as a fingerprint, to check whether he is who he claims to be.</td>
</tr>
<tr>
<td>Biometric template</td>
<td>A biometric recorded on registration and stored for comparison with a live scan of a user’s biological feature.</td>
</tr>
<tr>
<td>Brute Force Attack</td>
<td>An attempt to gain access to a system using every possible combination of characters in a password field.</td>
</tr>
<tr>
<td>CA</td>
<td>Certification Authority</td>
</tr>
<tr>
<td>Contract</td>
<td>A JML description of the invariant, pre- and post-conditions that a method must satisfy.</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>Decryption</td>
<td>The conversion of an encrypted message into a form in which its meaning can be understood.</td>
</tr>
<tr>
<td>Design</td>
<td>[In this dissertation] the software architecture described in sec. 3.3</td>
</tr>
<tr>
<td>Digital certificate</td>
<td>An electronic public key certificate issued by a CA associating a user and his public key.</td>
</tr>
<tr>
<td>Digital signature</td>
<td>A message digest (hash) encrypted with the private key of the sender. Decryption using the sender’s public key and comparison with a fresh hash of the message by the receiver authenticates the message as having come from the sender.</td>
</tr>
<tr>
<td>Encryption</td>
<td>The conversion of a message into a form in which its meaning cannot be understood to preserve its confidentiality.</td>
</tr>
<tr>
<td>Hash</td>
<td>A small block of data calculated from a message using a mathematical hash function with the property that if the message changes in any respect then...</td>
</tr>
</tbody>
</table>
the hash value also changes.

**Hash function**  
An algorithm that produces a hash value, or digest, using one or more inputs such as a message, a password or a timestamp. With a *one-way hash function*, it is computationally infeasible to derive the message from the digest. With a *strongly collision free hash function*, it is computationally infeasible to find two different messages for which the same hash value is produced by the function.

**Integrity**  
A property of data such that it cannot be modified without detection.

**JCA**  
Java Cryptography Architecture

**JMI**  
Java Metadata Interface

**JML**  
Java Modeling Language

**JSS**  
Network Security Services for Java

**JSSE**  
Java Secure Sockets Extension

**MAC**  
Message Authentication Code: a hash value calculated from a message which is used to check message integrity. The receiver re-calculates the MAC from the message and compares this with the MAC received from the sender; if they are the same then the message has not been changed.

**Nonce**  
A method of ensuring that a message is fresh by inserting an unpredictable value so that any replay of the message can be detected. The term is derived from ‘Number used ONCE’

**OCL**  
Object Constraint Language

**PIN**  
Personal Identification Number

**PKCS**  
Public Key Cryptography Standard

**PKI**  
Public Key Infrastructure, used to manage keys and certificates, and their distribution to users.

**Post-condition**  
A JML description of the conditions satisfied after a method terminates.

**Pre-condition**  
A JML description of the conditions that must be satisfied before calling a method.

**Registration**  
The recording of information about a user in a directory that can later be used for authentication and identification.

**Replay attack**  
An attempt to gain access to a system using a message that has been sent
previously, and recorded by an adversary.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secrecy, or</td>
<td>The prevention of unauthorised users reading data.</td>
</tr>
<tr>
<td>confidentiality</td>
<td>The prevention of unauthorised users reading data.</td>
</tr>
<tr>
<td>Security</td>
<td>The control of access to information or premises, and of the execution of</td>
</tr>
<tr>
<td></td>
<td>functions to those users who have permission.</td>
</tr>
<tr>
<td>Specification</td>
<td>[In this dissertation] the UML diagrams expressing system requirements</td>
</tr>
<tr>
<td></td>
<td>and the operation pre- and post-conditions in appendix C</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UMLsec</td>
<td>UML Security: an extension of UML for expressing security requirements</td>
</tr>
<tr>
<td>UMLsec approach</td>
<td>The application of security requirements to UML diagrams using UMLsec</td>
</tr>
<tr>
<td></td>
<td>and cryptographic protocol notations</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
</tbody>
</table>
Appendix C  Biometric Authentication System Operation Specifications

Host Operations

Context Host :: registerListeners()

post:
-- listeners registered for smart card insertion and removal
-- authentication initiated on load
-- terminate protocol initiated on unload

call self :: find(idsc:Integer)

call self :: v (Zsc:Integer, Zh:Integer) : Integer

context Host :: authenticateUser ()

pre:
-- Smart card plugged in

post:
-- Session key is negotiated
-- User's PIN is verified
-- User's live scan is verified
-- SSL session negotiated with Server

body:
call self :: negSessionKey ()
call self :: checkPin() 
call self :: scanBiometric ()
call self :: SSLHandshake ()

context Host :: negSessionKey ()

pre:

post:
-- a SmartCardOS object is created
-- a session key is established

body:
call SmartCardOS :: smartCardOS () : SmartCardOS

call SmartCardOS :: reset () : Integer

call SmartCardOS :: askZ () : Integer

call SmartCardOS :: ack (Z_w:Integer, Z_h:Integer, address_k:Integer) : Xchk

call self :: find (idsc:Integer)

call self :: v (Zsc:Integer, Zh:Integer) : Integer

context Host :: find(idsc:Integer)

pre:

post:
-- a temporary session key and the address of the smart card's key has been extracted from the smart card's Id

context Host :: v (Zsc:Integer, Zh:Integer)

pre:

post:
-- a session key has been calculated
context Host :: checkPin ()

pre:
-- the third misuse counter is non-zero.

post:
-- the user's PIN entry is valid
-- misuse counter 3 is set to its default value

body:
call self :: getUserPin () : Integer
call self :: decMisuseCounter (:Integer)
call SmartCardOS :: valPin (pin, Integer, :Integer) : retValPin
call SmartCardOS :: setFBZ (:String, :Integer)

context Host :: getUserPin () : Integer

post:
-- User has entered PIN

context Host :: decMisuseCounter (:Integer)

pre:
-- the value of the parameter is 1,2 or 3

post:
-- The respective misuse counter is decremented and verified
-- the calculated MACs are the same as the MACs supplied.
-- The sent nonce is the same as the one returned.

body:
call SmartCardOS :: getFBZ (i: integer, Zh : Integer, :Integer) : retFBZ
call SmartCardOS :: decFBZ (i: integer, :Integer)
call SmartCardOS :: getFBZ (i: integer, Zh : Integer, :Integer) : retFBZ

context Host :: scanBiometric ()

pre:
-- PIN has been verified

post:
-- The calculated hash is the same as the one returned
-- The smart card id and template extracted from the signature are the same as those sent.
-- The nonce sent is the same as the one returned.
-- The smart card template has been retrieved
-- The live scan matches biometric template after up to three scans
-- Misuse counter 2 is reset to its default value

body:
call self :: giveUserInstruction ()
call self :: decMisuseCounter (:Integer)
call SmartCardOS :: getData () : Xchd
call self :: bioMatch ()
Call SmartCardOS :: setFBZ (:String, :Integer)

context Host :: giveUserInstruction ()

post:
-- User has been told to place finger on scanner
context Host :: bioMatch ()

pre:
-- User has been told to place finger on scanner

post:
-- Live scan matches template after up to a maximum of three scans
-- Nonce returned is same as one sent.
-- Calculated MAC is same as one returned
-- Template deleted from Host.

body:
call BioSensor :: open ()
call BioSensor :: scanMatch (:Integer, :Integer) : scanResult
call BioSensor :: close ()

context Host :: SSLHandshake ()

pre:
-- PIN verified
-- biometric scan verified

post:
-- nonces match on get Certificate
-- misuse counter 1 reset to its default value
-- Smart Card session closed.

body:
call self :: getUserURL ()
call SmartCardOS :: getCert (:Integer, :Integer) : retCert
call self :: decMisuseCounter (:Integer)
call self :: httpsHandshake ()
Call SmartCardOS :: setFBZ (:String, :Integer)
Call SmartCardOS :: close ()

context Host :: getUserURL ()

pre:

post:
-- the user has input the server URL
-- the user has input the location of the certificate for client authentication

context Host :: httpsHandshake ()

post:
-- SSL connection established
-- SSL handshake completed
-- Server has authenticated client
-- Host has authenticated server
-- Server's certificate is valid:
-- Today's date is within server's certificate validity period
-- Issuing CA is a trusted CA
-- certificate is not revoked
-- Issuing CA's public key validates issuer's digital signature (i.e. subject name and subject
  public key extracted from signature using issuing CA public key match subject name and
  server public key on certificate). Will need to find and read CA cert in certificate store for
  this.
-- Server's certificate is listed in LDAP entry for user (to check a restricted list of servers that
  implement client authentication)
SmartCardOS operations

context SmartCardOS :: smartCardOS (): SmartCardOS
pre:
--   the user has plugged in the smart card
post:
--   a new SmartCardOS object has been created
   -- Nb. No smart card or reader is available so, for testing, create a smart card instance with a
certificate, template and password.

context SmartCardOS :: reset (): Integer
pre:
--   a SmartCardOS object has been created and linked to Host
post:
--   the id of the smart card is returned, containing a temporary session key and the address of the
smart card's key.

context SmartCardOS :: askZ (): Integer
pre:
post:
--   a nonce is returned to the Host.

context SmartCardOS :: ack (Zsc:Integer, Zh:Integer, addressk:Integer): Xchk
pre:
--   the first misuse counter is not zero.
--   the nonces match.
post:
--   the session key is calculated
--   the smart card and host nonces are returned to enable the host to calculate a session key
body:
call self :: find (addressk:Integer)
call self :: v (Zsc:Integer, Zh:Integer)

context SmartCardOS :: v (Zsc:Integer, Zh:Integer): Integer
pre:
post:
--   a session key has been calculated

context SmartCardOS :: valPin (:Integer): retValPin
pre:
--   the calculated MAC is the same as the MAC supplied.
post:
--   The user's PIN entry is the same as the smart card PIN.

context SmartCardOS :: getFBZ(i:integer, Zh:Integer, :Integer): retFBZ
pre:
--   the first parameter is 1, 2 or 3.
--   the calculated MAC is the same as the MAC supplied.
post:
--   the respective misuse counter is returned

context SmartCardOS :: decFBZ(i:integer, :Integer)
pre: the first parameter is 1, 2 or 3
-- the calculated MAC is the same as the MAC supplied.
post: the respective misuse counter is decremented.

context SmartCardOS :: setFBZ (:String, :Integer):
pre: the first parameter is SetFBZ1, SetFBZ2 or SetFBZ3.
-- the calculated MAC is the same as the MAC supplied.
post: the respective misuse counter is set to its default value

context SmartCardOS :: getData () : Xchd
pre: post: The biometric template has been retrieved from the smart card

context SmartCardOS :: getCert (:Integer, :Integer) : retCert
pre: post: The certificate is retrieved from the smart card

context SmartCardOS :: close ()
pre: post: the SmartCardOS object is destroyed

context SmartCardOS :: terminateProtocol ()
pre: post: smart card is unplugged
-- user PIN, server name and certificate location are erased
-- the SmartCardOS object is destroyed
-- server information is erased
-- live scan is erased
**BioSensor operations**

**context** BioSensor :: bioSensor () : BioSensor

**pre:**

**post:**
-- a new BioSensor object has been created

**context** BioSensor :: open () : BioSensor

**pre:**

**post:**
-- the BioSensor object has been initiated

**context** BioSensor :: scanMatch (t:Integer, :Integer) : scanResult

**pre:**
-- a BioSensor object has been created and initiated

**post:**
-- the smart card template has been loaded into the scanner
-- the user's feature has been scanned (live scan)
-- the live scan has been compared against the smart card template
-- the smart card template and live scan on the scanner have been erased

**body:**
call self :: match (tsc:Integer, scan:Integer) m:Integer

**context** BioSensor :: match (tsc:Integer, scan:Integer) m:Integer

**pre:**

**post:**
-- The live scan has been taken and compared with the template

**context** BioSensor :: close ()

**pre:**

**post:**
-- the connection to the Scanner has been closed
-- the object has been destroyed
package m801.framework;

import java.security.InvalidKeyException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.Mac;
import javax.crypto.ShortBufferException;
import javax.crypto.spec.SecretKeySpec;

public class BioSensor {
    private SecretKeySpec key;
    private Cipher cipher;
    private Mac mac;
    private int matchTol = 10;
    private int m, i, match;
    public Utilities utilities;

    BioSensor(SecretKeySpec sessionKey, Cipher hostCipher, Mac hostMac) {
        // Initialise session key.
        // Scanner and Smart Card reader are in same hardware so //
        // no need to do secure transfer.
        // previous key agreement protocol with smart card reader
        // has established key.
        key = sessionKey;
        cipher = hostCipher;
        mac = hostMac;
        utilities = new Utilities();
    }

    public byte[] scanMatch(byte[] cipherText) {
        byte[] bioTemplate, plainText;
        // Decrypt and validate message
        // bioTemplate = new byte [20];
        plainText = new byte [mac.getMacLength() + 28];
        try {
            cipher.init(Cipher.DECRYPT_MODE, key);
            int ctLength = cipher.update(cipherText, 0, 8,
                                          plainText, 0);
            ctLength += cipher.update(cipherText, 8, 20,
                                       plainText, ctLength);
            ctLength += cipher.doFinal(cipherText, 28,
                                         mac.getMacLength(), plainText, ctLength);
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        } catch (ShortBufferException e) {
            e.printStackTrace();
        } catch (IllegalBlockSizeException e) {
            e.printStackTrace();
        } catch (BadPaddingException e) {
            e.printStackTrace();
        }
    }
}
utilities.checkMac(plainText, 28, mac);
for (i=8; i < 28; i++) {
    bioTemplate[i-8] = plainText[i];
}
// Perform scan, delete template and construct and send
// scanResult message.
//
m = match(bioTemplate);
if (m <= matchTol)
    match = 0;
else
    match = 1;
for (i=0; i < 20; i++) {
    bioTemplate[i] = (byte) 0;
}
plainText[8] = (byte) match;
mac.update(plainText, 0, 9);
System.arraycopy(mac.doFinal(), 0, plainText, 9,
    mac.getMacLength());
return plainText;

private int match (byte[] bioTemplate) {
    //
    // Call to scanner hardware here. Scanner not
    // included in prototype so result assumed.
    //
    return 8;
}
Host

package m801.framework;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.net.ssl.HttpURLConnection;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
import org.bouncycastle.asn1.x509.CRLReason;
import org.bouncycastle.x509.X509V2CRLGenerator;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;
import java.net.URL;
import java.nio.*;
import java.security.*;
import java.security.cert.X509CRL;
import java.security.cert.X509Certificate;
import java.util.Date;
//@ model import org.jmlspecs.models.*;

public final class Host {
    public byte [] zSc, zH, zH1, kH, addressK, idSc,_plainText,
cipherText, xChk, xChd, pinH, retValPin, bioTemplate,
sigBytes;
private SmartCardOS smartCard;
public Cipher cipher;
public ByteBuffer byteBuffer;
public SecretKeySpec key;
public int ctLength, counterVal;
static byte returnCode0 = 0;
byte setMisuseCounterCode;
public Mac mac;
private SecureRandom random;
public Utilities utilities;
public UserPIN userInterface;
private PublicKey adminPublicKey;
private Signature signature;
KeyPair keyPair;

/*@ public static final ghost int
@ INITIAL = 1,
@ RESET = 2,
@ ASKZ = 3,
@ ACK = 4,
@ GETFBZ = 5,
@ SETFBZ = 6,
@ GETCERT = 7,
@ CLOSE = 8;
@*/
//@ public ghost int state = INITIAL;
}
public static void main(String[] args) {
    // This method would be invoked by an event listener
    // in JavaScript code when the smart card is inserted
    // into a reader.

    Host host = new Host();
    host.negSessionKey();
    host.checkPin();
    host.scanBiometric();
    try {
        host.SSLHandshake();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public Host (){
    //
    // Construct smart card object and initialise with
    // administrator's private key
    //
    try {
        KeyPairGenerator keyGen;
        keyGen = KeyPairGenerator.getInstance("DSA");
        keyGen.initialize(512, new SecureRandom());
        KeyPair keyPair = keyGen.generateKeyPair();
        signature = Signature.getInstance("DSA");
        smartCard = new SmartCardOS(keyPair.getPrivate(),
                                   signature);
        adminPublicKey = keyPair.getPublic();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    random = new SecureRandom();
    utilities = new Utilities();
}

/*@ public normal_behavior
@ requires state == INITIAL;
@ assignable \everything;
@ ensures state == CLOSE;
@*/
public void negSessionKey () {
    //
    // Get smart card's id and derive initial key and key
    // storage address from it.
    //
    setIdsc();
    //
    // Get nonce from smart card and create cipher.
    //
    zSc = smartCard.askZ();
    initCipher();
    //
    // Concatenate ACK message.
    //
    plainText = new byte[20];
    cipherText = new byte[20];
    byteBuffer = ByteBuffer.wrap(plainText);
    byteBuffer.put(zSc, 0, 8);
    zH = new byte[8];
}
random.nextBytes(zH);
byteBuffer.put(zH, 0, 8);
byteBuffer.put(addressK, 0, 4);

//
// Encrypt and send Ack message
//
try {
    ctLength = cipher.update(plainText, 0, 20, cipherText, 0);
    ctLength += cipher.doFinal(cipherText, ctLength);
} catch (ShortBufferException e) {
    e.printStackTrace();
    System.exit(1);
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
    System.exit(1);
} catch (BadPaddingException e) {
    e.printStackTrace();
    System.exit(1);
}

/*@ assert (∀ int x; 0 <= x && x < plainText.length; @
    plainText[x] != cipherText[x]);
@*/
cipherText = smartCard.ack(cipherText);

//
// Decrypt returned Xchk message, check nonces and
// calculate new key
//
xChk = new byte [16];
try {
    cipher.init(Cipher.DECRYPT_MODE, key);
    int ptLength = cipher.update(cipherText, 0, 16, xChk, 0);
    ptLength += cipher.doFinal(xChk, ptLength);
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (ShortBufferException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
}

utilities.checkNonce (xChk, 8, zH);
v();
System.out.println("Smart card handshake completed successfully");

//@ set state = ACK;
}

/*@ invariant pinH != null && pinH.length == 4
@    && (∀all int i; 0 <= i && i < 4;
@        0 <= pinH[i] && pinH[i] <= 9);
@*/
/*@ public normal_behavior
@    requires state == ACK;
@    assignable \everything;
@    ensures state == SETFBZ;
@*/
public void checkPin() {
    //
// Validate PIN entered by user.
//
// Decrement misuse counter 3 and get user-entered PIN
//
decMisuseCounter(3);
userInterface = new UserPIN("PIN");
pinH = new byte [4];
pinH = null;
while (pinH == null) {
    pinH = userInterface.getPin();
}

// Construct and send ValPin message
//
try {
    mac.update(pinH);
    cipher.init(Cipher.ENCRYPT_MODE, key);
    cipherText = new byte [mac.getMacLength() + 4];
    ctLength =
            cipher.update(pinH, 0, 4, cipherText, 0);
    ctLength += cipher.doFinal(mac.doFinal(), 0,
            mac.getMacLength(), cipherText, ctLength);
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (ShortBufferException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
}
retValPin = smartCard.valPin(cipherText);

// Decrypt and validate RetValPin message
//
byte [] plainText = new byte [mac.getMacLength() + 1];
try {
    cipher.init(Cipher.DECRYPT_MODE, key);
    int ptLength = cipher.update(retValPin, 0,
            mac.getMacLength() + 1, plainText, 0);
    ptLength += cipher.doFinal(plainText, ptLength);
} catch (ShortBufferException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
} catch (InvalidKeyException e) {
    e.printStackTrace();
}
utilities.checkMac(plainText, 1, mac);
if (plainText[0] != returnCode0) {
    System.out.println("Return Code " + returnCode0 + " -
            Invalid PIN");
    System.exit(1);
}

// Reset misuse counter 3
//@ set state = SETFBZ;
}
/*@ public normal_behavior
@ requires state == SETFBZ;
@ assignable \everything;
@ ensures state == SETFBZ;
@*/
public void scanBiometric () {
    //
    // Instruct user to put finger on scanner
    //
    UserBioscanInstructions instructions = new UserBioscanInstructions("Bioscan");
    boolean ack = false;
    while (!ack) {
        ack = instructions.getAck();
    }
    //
    // Decrement second misuse counter and send 'getData'
    // message to request template and certificate
    //
    decMisuseCounter(2);
    random.nextBytes(zH);
    plainText = new byte [mac.getMacLength() + 8];
    for (int i=0; i < 8; i++) {
        plainText[i] = zH[i];
    }
    mac.update(zH);
    try {
        mac.doFinal(plainText, 8);
    } catch (IllegalStateException e) {
        e.printStackTrace();
    } catch (ShortBufferException e) {
        e.printStackTrace();
    }
    xChd = smartCard.getData(plainText);
    //
    // Decrypt xchd message
    //
    zH1 = new byte [8];
    bioTemplate = new byte [20];
    sigBytes = new byte [xChd.length - 28];
    try {
        cipher.init(Cipher.DECRYPT_MODE, key);
        int ptLength = cipher.update(xChd, 0, 8, zH1, 0);
        ptLength +=
            cipher.update(xChd, 8, 20, bioTemplate, 0);
        ptLength +=
            cipher.update(xChd, 28, xChd.length - 28, sigBytes, 0);
        cipher.doFinal();
    } catch (ShortBufferException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (BadPaddingException e) {
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    }
    //
    // Validate nonce and signature
    //
    utilities.checkNonce(zH1, 0, zH);
try {
    signature.initVerify(adminPublicKey);
    signature.update(idSc);
    signature.update(bioTemplate);
    if (!signature.verify(sigBytes)) {
        System.out.println("Signature of smart card id and 
            template is invalid");
        System.exit(1);
    }
}
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (SignatureException e) {
    e.printStackTrace();
}

// Scan finger and match to template
//
if (!bioMatch()) {
    System.out.println("Finger scan does not match biometric 
        template");
    System.exit(1);
}

setFBZ(2);
System.out.println("Finger scan successful");
//@ set state = SETFBZ;

/**
@ public normal_behavior
  @ requires state == SETFBZ;
  @ assignable everything;
  @ ensures state == CLOSE;
@*/
public void SSLHandshake () throws Exception {
    /**
     * Ask the user to input the server URL
     */
    UserURL userURLFrame = new UserURL("URL");
    String userURL = null;
    while (userURL == null) {
        userURL = userURLFrame.getUserURL();
    }

    URL serverURL = new URL(userURL);
    /**
     * Create an SSL Context using the smart card key store 
     * and trust store
     */
    KeyManagerFactory mgrFact =
        KeyManagerFactory.getInstance("SunX509");
    KeyStore serverKeyStore = KeyStore.getInstance("JKS");
    serverKeyStore.load(new FileInputStream("smartCard.jks"),
        Utilities.SMARTCARD_PASSWORD);
    mgrFact.init(serverKeyStore,
        Utilities.SMARTCARD_PASSWORD);

    TrustManagerFactory trustFact =
        TrustManagerFactory.getInstance("SunX509");
    KeyStore trustStore = KeyStore.getInstance("JKS");
    trustStore.load(new FileInputStream("trustStore.jks"),
        Utilities.TRUST_STORE_PASSWORD);
    trustFact.init(trustStore);

    SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(mgrFact.getKeyManagers(),
    trustFact.getTrustManagers(), null);
/**
 * Establish a connection with the server.
 */
HttpsURLConnection connection = (HttpsURLConnection)
    serverURL.openConnection();
SSLSocketFactory fact = sslContext.getSocketFactory();
connection.setSSLSocketFactory(fact);
connection.setHostnameVerifier(new Validator());
connection.connect();
/**
 * Generate a test Certificate Revocation List and
 * check that the server's certificate is not revoked.
 * Adapted from Hook D. (2005)
 * The CRL would be normally read from an external
 * distribution point. It is created here to include a
 * CRL check in the prototype.
 */
BigInteger revokedSerialNumber = BigInteger.valueOf(5);
X509Certificate caCert;
caCert = (X509Certificate) trustStore.getCertificate("Trust
certificate");
PrivateKey caKey = (PrivateKey)
    trustStore.getKey("Trust private key",
    Utilities.TRUST_STORE_PASSWORD);
X509CRL crl = generateCRL(revokedSerialNumber, caCert,
    caKey);
crl.verify(caCert.getPublicKey());
X509Certificate [] certs = (X509Certificate [])
    connection.getServerCertificates();
for (int i=0; i < certs.length; i++) {
    if (crl.isRevoked(certs[i])) {
        System.out.print("Server certificate is revoked:
                        connection terminated");
        System.exit(0);
    }
}
/**
 * Read the response from the server.
 */
InputStream in;
System.out.println("Server response:");
try {
    in = connection.getInputStream();
    int ch = 0;
    while ((ch = in.read()) >= 0) {
        System.out.print((char)ch);
    }
} catch (IOException e) {
    e.printStackTrace();
}
connection.disconnect();
setFBZ(1);
smartCard.close();
System.out.println("Host successful termination");
//@ set state = CLOSE;
/*@ public normal_behavior
@ requires state == INITIAL;
@ assignable kH, addressK, idSc, state;
@ ensures state == RESET;
/*@*/
public void setIdsc() {

  ByteBuffer idBuffer;
  //
  // Get smart card id, derive initial key and storage
  // address on smart card of symmetric key.
  //
  idSc = smartCard.reset();
  idBuffer = ByteBuffer.wrap(idSc);
  kH = new byte [16];
  addressK = new byte [4];
  idBuffer.get(kH, 0, 16);
  idBuffer.get(addressK, 0, 4);
  //@ set state = RESET;
  return;
}

/*@ public normal_behavior
@ requires state == RESET;
@ assignable cipher, key, zSc;
@ ensures state == ASKZ;
@ ensures \result != null;
/*@*/
public Cipher initCipher() {

  // Set up cipher using temporary key from smart card
  //
  zSc = smartCard.askZ();
  try {
    cipher = Cipher.getInstance("RC4");
    } catch (NoSuchAlgorithmException a) {
      System.out.println("No RC4 algorithm available");
      System.exit(1);
    } catch (NoSuchPaddingException b) {
      System.out.println("No RC4 padding available");
      System.exit(1); 
    }
  key = new SecretKeySpec(kH, "RC4");
  try {
    cipher.init(Cipher.ENCRYPT_MODE, key);
    } catch (InvalidKeyException a) { 
      System.out.println("Invalid key exception");
      System.exit(1); 
    }
  // @ set state = ASKZ;
  return cipher;
}

//@ requires xChk != null;
private void v() {

  // Derive new key from zSc and zH nonces
  // Initialise MAC
  //
  key = new SecretKeySpec(xChk, "RC4");
  try {
    mac = Mac.getInstance("HmacSHA1");
    mac.init(key);
  } catch (NoSuchAlgorithmException a) {
    System.out.println("No HmacSHA1 algorithm available");
    System.exit(1);
  } catch (NoSuchProviderException b) {
    System.out.println("No HmacSHA1 provider available");
    System.exit(1); 
  } catch (InvalidAlgorithmParameterException c) {
    System.out.println("Invalid algorithm parameter");
    System.exit(1); 
  }
  // @ set state = ASKZ;
  return;
}
private void decMisuseCounter(int counterNo) {
    // Decrement the misuse counter on the smart card
    // Get the counter value
    int prevCounterVal = getCounterValue(counterNo);
    // Send a message to decrement the counter value
    plainText = new byte [mac.getMacLength() + 1];
    plainText[0] = (byte) counterNo;
    mac.update(plainText, 0, 1);
    try {
        mac.doFinal(plainText, 1);
    } catch (IllegalStateException e) {
        e.printStackTrace();
    } catch (ShortBufferException e) {
        e.printStackTrace();
    }
    smartCard.decFBZ(plainText);
    // Confirm counter value has been decremented
    int newCounterVal = getCounterValue(counterNo);
    if (newCounterVal != (prevCounterVal - 1)) {
        System.out.println("Error on decrementing misuse counter "+ counterNo);
        System.exit(1);
    }
}

/*@ public normal_behavior
@ requires state == ACK || state == SETFBZ ||
@ state == GETCERT;
@ requires counterNo > 0 && counterNo <= 3;
@ requires mac != null;
@ assignable counterVal, plainText, zH, mac, byteBuffer,
@ state;
@ ensures state == GETFBZ;
@ ensures \result > 0;
@ ensures \fresh (zH);
@*/
public int getCounterValue(int counterNo) {
    // Construct getFBZ(misuse counter+nonce+MAC(misuse
    // counter + nonce) message
    plainText = new byte [mac.getMacLength() + 9];
    byteBuffer = ByteBuffer.wrap(plainText);
    plainText[0] = (byte) counterNo;
    random.nextBytes(zH);
    byteBuffer.position(1);
    byteBuffer.put(zH, 0, 8);
mac.update(plainText, 0, 9);
try {
    mac.doFinal(plainText, 9);
} catch (ShortBufferException e) {
    e.printStackTrace();
} catch (IllegalStateException e) {
    e.printStackTrace();
}

// Send getFBZ message.
// Receive retFBZ (counter value + nonce + MAC(counter value + nonce) message.
// Check freshness and integrity of retFBZ message.
// Check misuse counter is non-zero.

byte retFBZ[] = new byte[mac.getMacLength() + 9];
retFBZ = smartCard.getFBZ(plainText);
utilities.checkNonce(retFBZ, 1, zH);
utilities.checkMac(retFBZ, 9, mac);
counterVal = retFBZ[0];
if (counterVal <= 0) {
    System.out.println("Misuse counter " + counterNo + " is zero: card invalid");
    System.exit(1);
}

//@ set state = GETFBZ;
return counterVal;

private void setFBZ (int setMisuseCounterCode) {

    // Construct and send a message to reset misuse counter to default value
    mac.update((byte) setMisuseCounterCode);
    byte setFBZ[] = new byte[mac.getMacLength() + 1];
    setFBZ[0] = (byte) setMisuseCounterCode;
    System.arraycopy(mac.doFinal(), 0, setFBZ, 1, mac.getMacLength());
    smartCard.setFBZ(setFBZ);
}

private boolean bioMatch() {

    // Scan and match biometric against template up to three times
    byte[] scanResult;
    BioSensor scanner = new BioSensor(key, cipher, mac);
    int i = 0;
    boolean match = false;
    while (((i < 4) && (!match)) ) {
        // Construct and send scanMatch message to scan finger and compare scan with template.
        i++;
        random.nextBytes(zH);
        mac.update(zH);
        mac.update(bioTemplate);
        try {
cipher.init(Cipher.ENCRYPT_MODE, key);
cipherText =
    new byte [mac.getMacLength() + 28];
ctLength =
cipher.update(zH, 0, 8, cipherText, 0);
ctLength += cipher.update(bioTemplate, 0, 20,
cipherText, ctLength);
ctLength += cipher.doFinal(mac.doFinal(), 0,
    mac.getMacLength(), cipherText, ctLength);
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (ShortBufferException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
}
scanResult = scanner.scanMatch(cipherText);
//
// Validate nonce, mac and scan return code
// (0 = success)
//
utilities.checkNonce(scanResult, 0, zH);
utilities.checkMac(scanResult, 9, mac);
if (scanResult[8] == (byte) 0) {
    match = true;
}
}
return match;

/**
* Generate a Certificate Revokation List containing one
* certificate.
* Adapted from Hook D., (2005)
*/
public static X509CRL generateCRL (BigInteger revokedSerialNumber,
    X509Certificate caCert, PrivateKey caKey) throws Exception {

    X509V2CRLGenerator crlGen = new X509V2CRLGenerator();
    Date now = new Date();
    crlGen.setIssuerDN(caCert.getSubjectX500Principal());
    crlGen.setThisUpdate(now);
    crlGen.setNextUpdate(now.getTime() + 100000);
    crlGen.setSignatureAlgorithm("SHA256WithRSAEncryption");
    crlGen.addCRLEntry(revokedSerialNumber, now,
        CRLReason.privilegeWithdrawn);

    return crlGen.generate(caKey, "BC");
}
KeyStoreSetup

package m801.framework;

import java.io.FileOutputStream;
import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.cert.X509Certificate;
import java.util.Date;
import java.util.Enumeration;
import javax.security.auth.x500.X500Principal;
import org.bouncycastle.x509.X509V1CertificateGenerator;

public class KeyStoreSetup {

    private static final int VALIDITY_PERIOD = 7 * 24 * 60 * 60 * 1000;  // one week

    /**
     * Basic class to confirm the Bouncy castle provider is
     * installed, to create root and end certificates, and to
     * create smart card and server key stores and a trust
     */
    public static void main(String[] args) {
        String providerName = "BC";
        if (Security.getProvider(providerName) == null) {
            System.out.println(providerName + " provider not installed");
        } else {
            System.out.println(providerName + " is installed.");
        }
        try {
            createKeyStores();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * Create a trust store and key stores for the server and
     * smart card (adapted from Hook D., (2005))
     */
    public static void createKeyStores() throws Exception {
        /**
         * Create a trusted root certificate and trust store
         * Load certificate and private key.
         */
        KeyPair rootKeyPair = generateRSAKeyPair();
        X509Certificate rootCert =
            generateCert(rootKeyPair.getPublic()),

    
}
rootKeyPair.getPrivate(),Utilities.trustCN);
KeyStore trustStore = KeyStore.getInstance("JKS");
trustStore.load(null, null);
trustStore.setCertificateEntry("Trust certificate",
rootCert);
trustStore.setKeyEntry("Trust private key",
rootKeyPair.getPrivate(),
Utilities.TRUST_STORE_PASSWORD,
new X509Certificate[] {rootCert});
trustStore.store(new
FileOutputStream(Utilities.TRUST_STORE_NAME + ".jks"),
Utilities.TRUST_STORE_PASSWORD);
Enumeration en = trustStore.aliases();
while (en.hasMoreElements())
{
String alias = (String) en.nextElement();
System.out.println("Trust Store: " + alias);
}
/**
* Create a smart card certificate and key store.
* load private key and certificate.
*/
KeyPair scKeyPair = generateRSAKeyPair();
X509Certificate scCert = generateCert(scKeyPair.getPublic(),
rootKeyPair.getPrivate(), Utilities.smartcardCN);
KeyStore keyStore = KeyStore.getInstance("JKS");
keyStore.load(null, null);
keyStore.setKeyEntry("Smartcard private key",
scKeyPair.getPrivate(),
Utilities.SMARTCARD_PASSWORD,
new X509Certificate[] {scCert, rootCert});
keyStore.setCertificateEntry("Smartcard certificate",
scCert);
keyStore.store(
new FileOutputStream(Utilities.SMARTCARD_NAME + ".jks"),
Utilities.SMARTCARD_PASSWORD);
Enumeration enSc = keyStore.aliases();
while (enSc.hasMoreElements())
{
String alias = (String) enSc.nextElement();
System.out.println("Smartcard key store: " + alias);
}
/**
* Create a server certificate and key store.
* load private key and certificate
*/
KeyPair sKeyPair = generateRSAKeyPair();
X509Certificate sCert = generateCert(sKeyPair.getPublic(),
rootKeyPair.getPrivate(), Utilities.serverCN);
KeyStore serverKeyStore = KeyStore.getInstance("JKS");
serverKeyStore.load(null, null);
serverKeyStore.setKeyEntry("Server private key",
sKeyPair.getPrivate(),
Utilities.SERVER_PASSWORD,
new X509Certificate[] {sCert, rootCert});
serverKeyStore.setCertificateEntry("Server certificate",
sCert);
serverKeyStore.store(
new FileOutputStream(Utilities.SERVER_NAME + ".jks"),
Utilities.SERVER_PASSWORD);
Enumeration enS = serverKeyStore.aliases();
while (enS.hasMoreElements()) {
    String alias = (String) enS.nextElement();
    System.out.println("Server key store: " + alias);
}

/**
 * Generate a sample V1 certificate
 * (adapted from Hook D., (2005))
 */
public static X509Certificate generateCert
    (PublicKey publicKey, PrivateKey privateKey, String CN)
        throws Exception {
    X509V1CertificateGenerator certGen = new
        X509V1CertificateGenerator();
    certGen.setSerialNumber(BigInteger.valueOf(1));
    certGen.setIssuerDN(new X500Principal(CN));
    certGen.setNotBefore
        (new Date(System.currentTimeMillis()));
    certGen.setNotAfter(new Date(System.currentTimeMillis() +
        VALIDITY_PERIOD));
    certGen.setSubjectDN(new X500Principal(CN));
    certGen.setPublicKey(publicKey);
    certGen.setSignatureAlgorithm("SHA1WithRSAEncryption");

    return certGen.generate(privateKey, "BC");
}

/**
 * Generate a random 1024 bit RSA key pair
 * (from Hook D., (2005))
 */
public static KeyPair generateRSAKeyPair() throws Exception {
    KeyPairGenerator kpGen = keyPairGenerator.getInstance("RSA");
    kpGen.initialize(1024, new SecureRandom());

    return kpGen.generateKeyPair();
}
Server

package m801.framework;

import java.io.*;
import java.security.*;
import javax.net.ssl.*;

/**
 * basic SSL Server with client authentication
 * Adapted from Hook D. (2006)
 */

public class Server {

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        /**
         * Create an SSL Context using the server key store and
         * trust store
         */
        KeyManagerFactory mgrFact = 
            KeyManagerFactory.getInstance("SunX509");
        KeyStore serverKeyStore = KeyStore.getInstance("JKS");
        serverKeyStore.load(new FileInputStream("server.jks"),
            Utilities.SERVER_PASSWORD);
        mgrFact.init(serverKeyStore,
            Utilities.SERVER_PASSWORD);

        TrustManagerFactory trustFact = 
            TrustManagerFactory.getInstance("SunX509");
        KeyStore trustStore = KeyStore.getInstance("JKS");
        trustStore.load(new FileInputStream("trustStore.jks"),
            Utilities.TRUST_STORE_PASSWORD);
        trustFact.init(trustStore);

        SSLContext sslContext = SSLContext.getInstance("TLS");
        sslContext.init(mgrFact.getKeyManagers(),
            trustFact.getTrustManagers(), null);

        /**
         * Create an SSL Server socket
         */
        SSLServerSocketFactory fact = 
            sslContext.getServerSocketFactory();
        SSLServerSocket sSock = (SSLServerSocket)
            fact.createServerSocket(Utilities.PORT_NO);
        sSock.setWantClientAuth(true);

        /**
         * Accept a connection from a client and read the
         * request.
         */
        for (;;) {
            SSLSocket sslSock = (SSLSocket) sSock.accept();
            try {
                sslSock.startHandshake();
            } 
            catch (IOException e) {
                e.printStackTrace();
            }
            InputStream in = sslSock.getInputStream();
        }
    }
}
System.out.print("Request: ");
int ch = 0;
int lastCh = 0;
while ((ch = in.read()) >= 0 && (ch != '\n' &&
  lastCh != '\n')) {
    System.out.println((char)ch);
    if (ch != '\r')
      lastCh = ch;
}
System.out.println();
/**
 * Authenticate client.
 */
SSLSession session = sslSock.getSession();
try {
  Principal clientID =
    session.getPeerPrincipal();
  System.out.println("client identified as " +
    clientID);
} catch (SSLPeerUnverifiedException e) {
  System.out.println
    ("client not authenticated");
}
/**
 * Send response.
 */
OutputStream out = sslSock.getOutputStream();
PrintWriter pWrt = new PrintWriter(new
  OutputStreamWriter(out));
pWrt.print("HTTP/1.1 200 OK\r\n");
pWrt.print("Content-Type: text/html\r\n");
pWrt.print("\r\n");
pWrt.print("<html>\r\n");
pWrt.print("<body>\r\n");
pWrt.print("Server connection established\r\n");
pWrt.print("</body>\r\n");
pWrt.print("</html>\r\n");
pWrt.flush();
sslSock.close();
}
SmartCardOS

package m801.framework;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;
import java.nio.ByteBuffer;
import java.security.*;

class SmartCardOS {
    public SecureRandom random;
    public byte[] zSc, zSc1, zH, kH, idSc, addressK, plainText, xChk,
            zSc2, cipherText, returnCode, biometricTemplate, sigBytes;
    public ByteBuffer byteBuffer;
    private Cipher cipher;
    private SecretKeySpec key;
    int ctLength;
    int[] fbz;
    private Mac mac;
    private PrivateKey adminPrivateKey;
    private Signature signature;
    public Utilities utilities;

    SmartCardOS(PrivateKey privateKey, Signature sig) {
        // Initialise smart card data.
        // Code to read data from the smart card and to
        // exchange data between the smart card and Host
        // should be inserted.
        //
        utilities = new Utilities();
        idSc = new byte[] {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
                0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                0x10, 0x20, 0x30, 0x40};
        kH = new byte[] {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
                0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
        biometricTemplate = new byte [] {0x00, 0x01, 0x02, 0x03,
                0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
                0x0d, 0x0e, 0x0f, 0x10, 0x20, 0x30, 0x40};
        fbz = new int[] {0, 3, 3, 3};
        returnCode = new byte[] {0x00};
        //
        // The administrator's private key and a signature are
        // read for test purposes only. They would normally
        // be written to the smart card on initial card
        // configuration.
        //
        adminPrivateKey = privateKey;
        signature = sig;
    }

    public byte[] reset() {
        return idSc;
    }
}
/*@ public normal_behavior
@ requires true;
@ assignable random, zSc;
@ ensures \fresh (zSc);
@*/
public byte[] askZ() {
    // return a random number
    //
    random = new SecureRandom();
    zSc = new byte[8];
    random.nextBytes(zSc);
    return zSc;
}

public byte[] ack(byte[] cipherText) {
    // Decrypt message
    try {
        cipher = Cipher.getInstance("RC4");
        SecretKeySpec key = new SecretKeySpec(kH, "RC4");
        cipher.init(Cipher.DECRYPT_MODE, key);
        plainText = new byte[20];
        int ptLength = cipher.update(cipherText, 0, 20,
                                    plainText, 0);
        ptLength += cipher.doFinal(plainText, ptLength);
    } catch (NoSuchAlgorithmException e2) {
        e2.printStackTrace();
    } catch (NoSuchPaddingException e2) {
        e2.printStackTrace();
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (ShortBufferException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (BadPaddingException e) {
        e.printStackTrace();
    }
    // Extract message items
    byteBuffer = ByteBuffer.wrap(plainText);
    zSc1 = new byte [8];
    zH = new byte [8];
    addressK = new byte [4];
    byteBuffer.get(zSc1, 0, 8);
    byteBuffer.get(zH, 0, 8);
    byteBuffer.get(addressK, 0, 4);
    // Validate misuse counter, nonce and key address
    if (fbz[1] == 0) {
        System.out.println("Card disabled - contact your card
                        provider");
        System.exit(1);
    }
    utilities.checkNonce(plainText, 0, zSc);
    if (getKsc(addressK).equals(null)) {
        System.out.println("Invalid smart card key address");
    }
System.exit(1);

// Compose return message
byte [] cipherxChd = new byte[16];
xChk = new byte[16];
byteBuffer = ByteBuffer.wrap(xChk);
SecureRandom random = new SecureRandom();
zSc2 = new byte[8];
random.nextBytes(zSc2);
byteBuffer.put(zSc2, 0, 8);
byteBuffer.put(zH, 0, 8);
// Encrypt and send Xchd message, and derive new key
try {
    ctLength = cipher.update(xChk, 0, 16, cipherxChd, 0);
    ctLength += cipher.doFinal(cipherxChd, ctLength);
} catch (ShortBufferException e1) {
    e1.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
}
return cipherxChd;

private SecretKeySpec getKsc(byte [] addressK) {
    // Get smart card session key using storage address.
    // Smart card I/O not implemented in prototype so key
    // is not retrieved from addressK but instead a default
    // value is used. In a real system, if address was
    // invalid then no SecretKeySpec would be returned.
    byte [] defaultKey = new byte[]{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x20, 0x30, 0x40};
    SecretKeySpec keySpecSC = new SecretKeySpec(defaultKey, "RC4");
    return keySpecSC;
}

private void v() {
    // Derive new key from zSc and zH nonces.
    // Initialise cipher and MAC
    key = new SecretKeySpec(xChk, "RC4");
    try {
        cipher.init(Cipher.DECRYPT_MODE, key);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    }
    try {
        mac = Mac.getInstance("HmacSHA1");
        mac.init(key);
    }
catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
} catch (InvalidKeyException e) {
    e.printStackTrace();
}

public byte[] valPin(byte[] cipherText) {
    // Validate PIN and return 0 = OK or 1 = invalid.
    // Code to retrieve PIN on Smart Card and compare it
    // with user-entered PIN is not implemented:
    // code of 0 always returned.
    // Decrypt valpin message and check MAC
    //
    plainText = new byte[mac.getMacLength() + 4];
    try {
        cipher.init(Cipher.DECRYPT_MODE, key);
        int ptLength = cipher.update(cipherText, 0,
                mac.getMacLength() + 4, plainText, 0);
        ptLength += cipher.doFinal(plainText, ptLength);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (ShortBufferException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (BadPaddingException e) {
        e.printStackTrace();
    }
    utilities.checkMac(plainText, 4, mac);
    // Construct RetValPin message
    //
    cipherText = new byte[mac.getMacLength() + 1];
    try {
        mac.update(returnCode);
        cipher.init(Cipher.ENCRYPT_MODE, key);
        ctLength = cipher.update(returnCode, 0, 1, cipherText, 0);
        ctLength += cipher.doFinal(mac.doFinal(), 0,
                mac.getMacLength(), cipherText, ctLength);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (ShortBufferException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (BadPaddingException e) {
        e.printStackTrace();
    }
    return cipherText;
}

public void setFBZ (byte[] message) {
    //
    // Validate MAC and reset appropriate misuse counter
    //
    utilities.checkMac(message, 1, mac);
    fbz[message[0]] = 3;
public byte[] getFBZ (byte[] message) {
    //
    // Validate MAC and return counter value, received
    // nonce and new MAC.
    //
    utilities.checkMac(message, 9, mac);
    message[0] = (byte) fbz[message[0]];
    mac.update(message, 0, 9);
    try {
        mac.doFinal(message, 9);
    } catch (ShortBufferException e) {
        e.printStackTrace();
    } catch (IllegalStateException e) {
        e.printStackTrace();
    }
    return message;
}

public void decFBZ (byte[] message) {
    //
    // Decrement misuse counter
    //
    utilities.checkMac(message, 1, mac);
    fbz[message[0]]--;
}

public byte [] getData (byte [] message) {
    //
    // Validate MAC and return it, biometric template and
    // hash of template and id.
    //
    // Validate MAC
    // Construct signature if id and biometric template
    //
    utilities.checkMac(message, 8, mac);
    try {
        try {
            signature.initSign(adminPrivateKey);
            signature.update(idSc);
            signature.update(biometricTemplate);
            sigBytes = signature.sign();
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        } catch (SignatureException e) {
            e.printStackTrace();
        }
    }
Encrypt and send xchd message

cipherText = new byte [28+sigBytes.length];
try {
    cipher.init(Cipher.ENCRYPT_MODE, key);
    ctLength =
        cipher.update(message, 0, 8, cipherText, 0);
    ctLength += cipher.update(biometricTemplate, 0, 20,
                              cipherText, 8);
    ctLength += cipher.doFinal(sigBytes, 0, sigBytes.length,
                                cipherText, ctLength);
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (ShortBufferException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
}
return cipherText;
}

public void close() {
    //
    //  Connection to Smart Card terminated.
    //
}
}
public class UserBioscanInstructions extends Frame implements ActionListener {

    private TextField txtA;
    private Label lb;
    private boolean acknowledged = false;

    UserBioscanInstructions (String s) { // Display instructions to user
        super(s);
        setLayout(new GridLayout(1,2));
        lb = new Label("Put your finger on the scanner and press 'Enter'");
        txtA = new TextField(" ", 10);
        txtA.addActionListener(this);
        add(lb);
        add(txtA);
        setSize(600,50);
        setAlwaysOnTop(true);
        setVisible(true);

        public void actionPerformed(ActionEvent e) {
            acknowledged = true;
            setVisible(false);
        }

        public boolean getAck() {
            return acknowledged;
        }
    }
}
UserPIN

package m801.framework;

import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class UserPIN extends Frame implements ActionListener {
    private TextField txtA;
    private byte[] pin;
    private Label lb;
    static final long serialVersionUID = 0;

    UserPIN (String s) {
        // Read PIN entered by user
        super(s);
        setLayout(new GridLayout(1,2));
        lb = new Label("Enter PIN");
        txtA = new TextField (4);
        txtA.addActionListener(this);
        add(lb);
        add(txtA);
        setSize(600,50);
        setAlwaysOnTop(true);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        String sPin = txtA.getText();
        pin = new byte [4];
        pin = sPin.getBytes();
        setVisible(false);
    }

    public byte[] getPin() {
        return pin;
    }

    public void giveUserInstruction () {
        // Tell user to put finger on scanner
        lb.setText("Put your finger on the scanner and press Enter");
        txtA.setText(" ");
        setVisible(true);
    }
}

UserURL

package m801.framework;

import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;
import java.awt.Checkbox;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class UserURL extends Frame implements ActionListener {

    private TextField txtA;
    private String url = "https://localhost:9020";
    private String userURL;
    private Label lb;
    static final long serialVersionUID = 0;

    UserURL (String s) {
        super(s);
       .setLayout(new GridLayout(2,2));
        lb = new Label("Enter URL");
        txtA = new TextField (url, 25);
        txtA.addActionListener(this);
        Checkbox smartcard = new Checkbox("Use smartcard certificate for client authentication", true);
        add(lb);
        add(txtA);
        add(smartcard);
        setSize(600,100);
        setAlwaysOnTop(true);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        userURL = txtA.getText();
        setVisible(false);
    }

    public String getUserURL() {
        return userURL;
    }
}
Utilities

package m801.framework;

import java.nio.ByteBuffer;
import javax.crypto.Mac;
//@ model import org.jmlspecs.models.*;

public class Utilities {

    public static final String SERVER_NAME = "server";
    public static final char[] SERVER_PASSWORD =
        "serverPassword".toCharArray();

    public static final String SMARTCARD_NAME = "smartCard";
    public static final char[] SMARTCARD_PASSWORD =
        "smartcardPassword".toCharArray();

    public static final String TRUST_STORE_NAME = "trustStore";
    public static final char[] TRUST_STORE_PASSWORD =
        "trustStorePassword".toCharArray();

    public static final String trustCN = "CN=Test Certificate";

    public static final String smartcardCN = "CN=Test Certificate";

    public static final String serverCN = "CN=Test Certificate";

    static final String HOST = "localhost";
    static final int PORT_NO = 9020;

    public byte[] calcMac, receivedMac, zH, zH1;

    public void utilities () {
    }

    //@ public normal_behavior
    @ requires length > 0 && length <= message.length;
    @ assignable \nothing;
    @ ensures (\forall int x; 0 <= x &&
        x < mac.getMacLength();
    @
        calcMac[x] == receivedMac[x]);
    @*/

    public void checkMac (byte[] message, int length, Mac mac) {
        // Check received MAC in byte 'length'+ of the
        // message is the same as the calculated MAC of the
        // value in the first 'length' bytes of the message.
        //
        byte receivedMac [] = new byte [mac.getMacLength()];
        System.arraycopy(message, length, receivedMac, 0,
            mac.getMacLength());
        mac.update(message, 0, length);
        byte calcMac [] = new byte [mac.getMacLength()];
        calcMac = mac.doFinal();
    }
}
for (int i = 0; i < mac.getMacLength(); i++) {
    byte a = receivedMac[i];
    byte b = calcMac[i];
    if (a != b) {
        System.out.println("Invalid MAC received by Smart card");
        System.exit(1);
    }
}

/*@ public normal_behavior
@ requires offset >= 0 &&
@ offset <= message.length - 8;
@ assignable 
nothing;
@ ensures (\forall int x; 0 <= x && x < 8;
@ zH[x] == zH1[x]);
@*/
public void checkNonce
(byte[] message, int offset, byte[] zH) {
    //
    // Check received nonce beginning at byte offset in
    // the message is the same as the nonce zH
    //
    ByteBuffer byteBuffer = ByteBuffer.wrap(message);
    byte zH1[] = new byte[8];
    byteBuffer.position(offset);
    byteBuffer.get(zH1, 0, 8);
    byte a, b;
    for (int i = 0; i < 8; i++) {
        a = zH[i];
        b = zH1[i];
        if (a != b) {
            System.out.println("Nonces " + zH + " and "+ zH1 + "
             do not match: possible attack");
            System.exit(1);
        }
    }
}
Validator

package m801.framework;

import javax.net.ssl.SSLSession;
import javax.net.ssl.HostnameVerifier;
import javax.security.auth.x500.X500Principal;
/**
 * Override HostnameVerifier to verify Common Name of Server.
 */
class Validator implements HostnameVerifier {
    public boolean verify(String hostName, SSLSession session) {
        try {
            X500Principal hostID = (X500Principal) session.getPeerPrincipal();
            return hostID.getName().equals(Utilities.serverCN);
        } catch (Exception e) {
            return false;
        }
    }
}
Appendix E  JML Signature specification comparison with javadoc documentation

getInstance (Algorithm)

JML:

//@ public normal_behavior
//@ requires (* algorithm is available *);
//@ assignable \nothing;
//@ ensures \fresh(result) && result.isPristine() &&
//@ result.algorithm == algorithm;

javadoc: Returns a Signature object that implements the specified signature algorithm. This method traverses the list of registered security Providers, starting with the most preferred Provider. A new Signature object encapsulating the SignatureSpi implementation from the first Provider that supports the specified algorithm is returned. Note that the list of registered providers may be retrieved via the Security.getProviders() method.

The javadoc is more comprehensive. The JML says the method returns a new Signature that is ‘pristine’ and implements the signature algorithm. However it is not clear what pristine means and it is not defined in the JML reference manual. Cheon and Perumandha (2006) use it to mean the initial state of the object. The JML only informally specifies that the algorithm must be available. The JML does not describe how the method works or how the registered providers can be listed.

initSign (PrivateKey)

JML:

//@ public normal_behavior
//@ requires (* privateKey is valid *);
//@ assignable data, key, state;
//@ ensures this.data.isEmpty()&&this.key ==
//@ privateKey&&this.state == java.security.Signature.SIGN;

javadoc: Initialize this object for signing. If this method is called again with a different argument, it negates the effect of this call.

Once understood, the JML is more comprehensive because it states that the ‘data’ field from which the signature is generated must be empty, the private key is stored and the Signature object’s state is changed to initialised for signing. It is clear from this description that if the method was called again with a different argument then the new private key would be stored.
**update (byte)**

JML:

```java
//@ public normal_behavior
//@ requires data.length > 0;
//@ requires this.state == java.security.Signature.VERIFY ||
//@ this.state == java.security.Signature.SIGN;
//@ assignable this.data;
//@ ensures this.data.equals
//@   (\old(this.data.concat(toSeq(data))));
```

javadoc: **Updates the data to be signed or verified, using the specified array of bytes.**

The JML is again more comprehensive: there must be at least one input byte; the signature must be in signing or verifying mode before the method is called; and the input bytes are concatenated with the previously stored bytes.

**sign()**

JML:

```java
//@ public normal_behavior
//@ requires this.state == java.security.Signature.SIGN;
//@ assignable data;
//@ ensures this.data.isEmpty()&&
//@   (* \result is signature of \old(data) *);
```

javadoc: **Returns the signature bytes of all the data updated. The format of the signature depends on the underlying signature scheme. A call to this method resets this signature object to the state it was in when previously initialized for signing via a call to initSign(PrivateKey). That is, the object is reset and available to generate another signature from the same signer, if desired, via new calls to update and sign. Returns: the signature bytes of the signing operation's result.**

There is little difference. JML adds that the signature must have been initialised for signing. However it only informally describes part of the result and only implies that the object is available to generate another signature by saying the stored bytes from which the signature is derived are erased. javadoc describes the returned object twice.