



<http://researchspace.auckland.ac.nz>

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage.

<http://researchspace.auckland.ac.nz/feedback>

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form.

The Design and Verification of a Cryptographic Security Architecture

Peter Gutmann

A thesis submitted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

The University of Auckland, 2000

Abstract

A cryptographic security architecture constitutes the collection of hardware and software which protects and controls the use of encryption keys and similar cryptovariables. This thesis presents a design for a portable, flexible high-security architecture based on a traditional computer security model. Behind the API it consists of a kernel implementing a reference monitor which controls access to security-relevant objects and attributes based on a configurable security policy. Layered over the kernel are various objects which abstract core functionality such as encryption and digital signature capabilities, certificate management, and secure sessions and data enveloping (email encryption).

The kernel itself uses a novel design which bases its security policy on a collection of filter rules enforcing a cryptographic module-specific security policy. Since the enforcement mechanism (the kernel) is completely independent of the policy database (the filter rules), it is possible to change the behaviour of the architecture by updating the policy database without having to make any changes to the kernel itself. This clear separation of policy and mechanism contrasts with current cryptographic security architecture approaches which, if they enforce controls at all, hardcode them into the implementation, making it difficult to either change the controls to meet application-specific requirements or to assess and verify them.

To provide assurance of the correctness of the implementation, this thesis presents a design and implementation process which has been selected to allow the implementation to be verified in a manner which can reassure an outsider that it does indeed function as required. In addition to producing verification evidence which is understandable to the average user, the verification process for an implementation needs to be fully automated and capable of being taken down to the level of running code, an approach which is currently impossible with traditional methods. The approach presented here makes it possible to perform verification at this level, something which had previously been classed as “beyond A1” (that is, not achievable using any known technology).

The versatility of the architecture presented here has been proven through its use in implementations ranging from 16-bit microcontrollers through to supercomputers, as well as a number of unusual areas such as security modules in ATMs and cryptographic coprocessors for general-purpose computers.

Acknowledgements

This thesis has been a long time in coming. My thesis supervisor, Dr. Peter Fenwick, had both the patience to await its arrival and the courage to let me do my own thing, with occasional course corrections as some areas of research proved to be more fruitful than others. I hope that the finished work rewards his confidence in me.

I spent the last two years of my thesis as a visiting scientist at the IBM T.J. Watson Research Centre in Hawthorne, New York. During that time the members of the global security analysis lab (GSAL) and the smart card group provided a great deal of advice and feedback on my work, augmented by the considerable resources of the Watson research library. Leendert van Doorn, Paul Karger, Elaine and Charles Palmer, Ron Perez, Dave Safford, Doug Schales, Sean Smith, Wietse Venema, and Steve Weingart all helped contribute to the final product, and in return probably found out more about lobotomised flatworms and sheep than they ever cared to know.

Before coming to IBM, Orion Systems in Auckland, New Zealand for many years provided me with a place to drink Mountain Dew, print out research papers, and test various implementations of the work described in this thesis. Paying me wages while I did this was a nice touch, and helped keep body and soul together.

Portions of this work have appeared both as refereed conference papers and in online publications. Trent Jaeger, John Kelsey, Bodo Möller, Brian Oblivion, Colin Plumb, Geoff Thorpe, Jon Tidswell, Robert Rothenburg Walking-Owl, Chris Zimman, and various anonymous conference referees have offered comments and suggestions which have improved the quality of the result. As the finished work neared completion, Charles “lint” Palmer, Trent “gcc -wall” Jaeger and Paul “lclint” Karger went through various chapters and pointed out sections where things could be clarified and improved.

Finally, I would like to thank my family for their continued support while I worked on my thesis. Without them, none of this would have been possible.

Hawthorne, July 2000

Peter Gutmann

John Roebling had sense enough to know what he *didn't* know. So he designed the stiffness of the truss on the Brooklyn Bridge roadway to be *six times* what a normal calculation based on known static and dynamic loads would have called for. When Roebling was asked whether his proposed bridge wouldn't collapse like so many others, he said “No, because I designed it six times as strong as it needs to be, to prevent that from happening”

— *Jon Bentley, “Programming Pearls”*

Preface

A cryptographic security architecture constitutes the collection of hardware and software which protects and controls the use of encryption keys and similar cryptovariables. Traditional security architectures have concentrated mostly on defining an application programming interface (API) and left the internal details up to individual implementers. This thesis presents a design for a portable, flexible high-security architecture based on a traditional computer security model. Behind the API it consists of a kernel implementing a reference monitor which controls access to security-relevant objects and attributes based on a configurable security policy. Layered over the kernel are various objects which abstract core functionality such as encryption and digital signature capabilities, certificate management, and secure sessions and data enveloping (email encryption). This allows them to be easily moved into cryptographic devices such as smart cards and crypto accelerators for extra performance or security. Chapter 1 introduces the software architecture and provides a general overview of features such as the object model and inter-object communications.

Since security-related functions which handle sensitive data pervade the architecture, security must be considered in every aspect of the design. Chapter 2 provides a comprehensive overview of the security features of the architecture, beginning with an analysis of requirements and an introduction to various types of security models and security kernel design, with a particular emphasis on separation kernels of the type used in the architecture. The kernel contains various security and protection mechanisms which it enforces for all objects within the architecture, as covered in the latter part of the chapter.

The kernel itself uses a novel design which bases its security policy on a collection of filter rules enforcing a cryptographic module-specific security policy. The implementation details of the kernel and its filter rules are presented in Chapter 3, which first examines similar approaches used in other systems and then presents the kernel design and implementation details of the filter rules.

Since the enforcement mechanism (the kernel) is completely independent of the policy database (the filter rules), it is possible to change the behaviour of the architecture by updating the policy database without having to make any changes to the kernel itself. This clear separation of policy and mechanism contrasts with current cryptographic security architecture approaches which, if they enforce controls at all, hardcode them into the implementation, making it difficult to either change the controls to meet application-specific requirements or to assess and verify them. The approach to enforcing security controls which is presented here is important not simply for aesthetic reasons but also because it is crucial to the verification process discussed in Chapter 5.

Once a security system has been implemented, the traditional (in fact pretty much the only) means of verifying the correctness of the implementation has been to apply various approaches based on formal methods. This has several drawbacks which are examined in some detail in Chapter 4. This chapter covers various problems associated not only with formal methods but with other possible alternatives as well, concluding that neither the application of formal methods nor the use of alternatives such as the CMM present a very practical means of building high-assurance security software.

Rather than taking a fixed methodology and trying to force-fit the design to fit the methodology, this thesis instead presents a design and implementation process which has been selected to allow the design to be verified in a manner which can reassure an outsider that it does indeed function as required, something which is practically impossible with a formally verified design. Chapter 5 presents a new approach to building a trustworthy system which combines cognitive psychology concepts and established software engineering principles. This combination allows evidence to support the assurance argument to be presented to the user in a manner which should be both palatable and comprehensible.

In addition to producing verification evidence which is understandable to the average user, the verification process for an implementation needs to be fully automated and capable of being taken down to the level of running code, an approach which is currently impossible with traditional methods. The approach presented here makes it possible to perform verification at this level, something which had previously been classed as “beyond AI” (that is, not achievable using any known technology). This level of verification can be achieved principally because the kernel design and implementation has been carefully chosen to match the functionality

embodied in the verification mechanism. The behaviour of the kernel then exactly matches the functionality provided by the verification mechanism and the verification mechanism provides exactly those checks which are needed to verify the kernel. The result of this co-design process is an implementation for which a binary executable can be pulled from a running system and re-verified against the specification at any point, a feature which would be impossible with formal-methods-based verification.

The primary goal of a cryptographic security architecture is to safeguard cryptovariables such as keys and related security parameters from misuse. Sensitive data of this kind lies at the heart of any cryptographic system and must be generated by a random number generator of guaranteed quality and security. If the cryptovvariable generation process is insecure then even the most sophisticated protection mechanisms in the architecture won't do any good. More precisely, the cryptovvariable generation process must be subject to the same high level of assurance as the kernel itself if the architecture is to meet its overall design goal, even though it isn't directly a part of the security kernel.

Because of the importance of this process, an entire chapter is devoted to the topic of random number generation for use as cryptovariables. Chapter 6 begins with a requirements analysis and a survey of existing generators, including extensive coverage of pitfalls which must be avoided. It then describes the method used by the architecture to generate cryptovariables, and applies the same verification techniques used in the kernel to the generator. Finally, the performance of the generator on various operating systems is examined.

Although the architecture works well enough in a straightforward software-only implementation, the situation where it really shines is when it is used as the equivalent of an operating system for cryptographic hardware (rather than having to share a computer with all manner of other software, including trojan horses and similar malware). Chapter 7 presents a sample application in which the architecture is used with a general-purpose embedded system, with the security kernel acting as a mediator for access to the cryptographic functionality embedded in the device. This represents the first open-source cryptographic processor, capable of being built from off-the-shelf hardware controlled by the software which implements the architecture.

Because the kernel is now running in a separate physical device it is possible for it to perform additional actions and checks which are not feasible in a general-purpose software implementation. The chapter covers some of the threats which a straightforward software implementation is exposed to, and then examines ways in which a cryptographic coprocessor based on the architecture can counter these threats. For example it can use a trusted I/O path to request confirmation for actions such as document signing and decryption which would otherwise be vulnerable to manipulation by trojan horses running in the same environment as a pure software implementation.

Finally, the conclusion looks at what has been achieved, and examines avenues for future work.

Contents

1. The Software Architecture	1
1. Introduction	3
2. An Introduction to Software Architecture	4
2.1. The Pipe and Filter Model	5
2.2. The Object-Oriented Model	5
2.3. The Event-based Model	6
2.4. The Layered Model	6
2.5. The Repository Model	7
2.6. The Distributed Process Model	7
2.7. The Forwarder-Receiver Model	7
3. Architecture Design Goals	8
4. The Object Model	9
4.1. User « Object Interaction	9
4.2. Action Objects	10
4.3. Data Containers	11
4.4. Key and Certificate Containers	12
4.5. Security Attribute Containers	12
4.6. The Overall Architectural and Object Model	13
5. Object Internals	14
5.1. Object Internal Details	15
5.2. Data Formats	17
6. Inter-object Communications	17
6.1. Message Routing	18
6.2. Message Routing Implementation	19
6.3. Alternative Routing Strategies	20
7. The Message Dispatcher	21
7.1. Asynchronous vs. Synchronous Message Dispatching	23
8. Object Reuse	24
8.1. Object Dependencies	26
9. Object Management Message Flow	26
10. Other Kernel Mechanisms	28
10.1. Semaphores	28
10.2. Threads	28
10.3. Event Notification	29
11. References	29
2. The Security Architecture	33
1. Security Features of the Architecture	35
1.1. Security Architecture Design Goals	36
2. Introduction to Security Mechanisms	37
2.1. Access Control	37
2.2. Reference Monitors	38
2.3. Security Policies and Models	38
2.4. Security Models after Bell-LaPadula	39
2.5. Security Kernels and the Separation Kernel	41
2.6. The Generalised TCB	43
2.7. Implementation Complexity Issues	44

3. The cryptlib Security Kernel.....	46
3.1. Extended Security Policies and Models.....	48
3.2. Controls Enforced by the Kernel.....	49
4. The Object Life Cycle.....	51
4.1. Object Creation and Destruction.....	52
5. Object Access Control.....	53
5.1. Object Security Implementation.....	54
5.2. External and Internal Object Access.....	55
6. Object Usage Control.....	56
6.1. Permission Inheritance.....	57
6.2. The Security Controls as an Expert System.....	58
6.3. Other Object Controls.....	59
7. Protecting Objects Outside the Architecture.....	59
8. Object Attribute security.....	60
9. References.....	61
3. The Kernel Implementation.....	67
1. Kernel Message Processing.....	69
1.1. Rule-based Policy Enforcement.....	69
1.2. The DTOS/Flask Approach.....	70
1.3. Meta-Objects for Access Control.....	71
1.4. Access Control via Message Filter Rules.....	72
2. Filter Rule Structure.....	73
2.1. Filter Rules.....	74
3. Attribute ACL Structure.....	77
3.1. Attribute ACLs.....	78
4. Mechanism ACL Structure.....	80
4.1. Mechanism ACLs.....	81
5. Message Filter Implementation.....	83
5.1. Pre-dispatch Filters.....	83
5.2. Post-dispatch Filters.....	85
6. Customising the Rule-based Policy.....	85
7. Miscellaneous Implementation Issues.....	87
7.1. Object Locking Implementation.....	87
8. Performance.....	89
9. References.....	89
4. Verification Techniques.....	89
1. Introduction.....	91
2. Formal Security Verification.....	91
2.1. Formal Security Model Verification.....	93
3. Problems with Formal Verification.....	94
3.1. Problems with Tools and Scalability.....	94
3.2. Formal Methods as a Swiss Army Chainsaw.....	95
3.3. What Happens when the Chainsaw Sticks.....	96
3.4. What is being Verified/Proven?.....	98
3.5. Credibility of Formal Methods.....	101
3.6. Where Formal Methods are Cost-Effective.....	102
3.7. Whither Formal Methods?.....	103
4. Problems with other Software Engineering Methods.....	104

4.1. Assessing the Effectiveness of Software Engineering Techniques	105
5. Alternative Approaches	107
5.1. Extreme Programming.....	108
5.2. Lessons from Alternative Approaches.....	109
6. References	109
5. Verification of the cryptlib Kernel.....	119
1. An Analytical Approach to Verification Methods	121
1.1. Peer Review as an Evaluation Mechanism	122
1.2. Enabling Peer Review	123
1.3. Selecting an Appropriate Specification Method.....	123
1.4. A Unified Specification	125
1.5. Enabling Verification All the way Down.....	126
2. Making the Specification and Implementation Comprehensible	126
2.1. Program Cognition	127
2.2. How Programmers Understand Code	128
2.3. Code Layout to Aid Comprehension	130
2.4. Code Creation and Bugs.....	131
2.5. Avoiding Specification/Implementation Bugs	132
3. Verification All the way Down	133
3.1. Programming with Assertions.....	134
3.2. Specification using Assertions.....	135
3.3. Specification Languages.....	136
3.4. English-like Specification Languages.....	137
3.5. Spec	138
3.6. Larch.....	139
3.7. ADL.....	140
3.8. Other Approaches.....	142
4. The Verification Process	143
4.1. Verification of the Kernel Filter Rules	143
4.2. Specification-based Testing.....	144
4.3. Verification with ADL.....	145
5. Conclusion.....	146
6. References	146
6. Random Number Generation	153
1. Introduction	155
2. Requirements and Limitations of the Generator	157
3. Existing Generator Designs and Problems	159
3.1 The Applied Cryptography Generator	160
3.2 The ANSI X9.17 Generator.....	161
3.3 The PGP 2.x Generator	161
3.4 The PGP 5.x Generator	162
3.5 The /dev/random Generator.....	163
3.6 The Skip Generator	164
3.7 The ssh Generator.....	165
3.8 The SSLeay/OpenSSL Generator	166
3.9 The CryptoAPI Generator	167
3.10 The Capstone/Fortezza Generator	168
3.11 The Intel Generator	169
4. The cryptlib Generator	170
4.1 The Mixing Function	170

4.2 Protection of Pool Output	171
4.3 Output Post-processing	172
4.4 Other Precautions	172
4.5 Nonce Generation	172
4.6 Generator Continuous Tests	172
4.7 Generator Verification	173
4.8 System-specific Pitfalls	174
4.9 A Taxonomy of Generators	176
5. Polling for Randomness	176
5.1 Problems with User-supplied Entropy	177
5.2 Entropy Polling Strategy	177
5.3 Win16 Polling	178
5.4 Macintosh and OS/2 Polling	178
5.5 BeOS Polling	178
5.6 Win32 Polling	179
5.7 Unix Polling	180
5.8 Other Entropy Sources	181
6. Randomness Polling Results	182
6.1 Data Compression as an Entropy Estimation Tool	182
6.2 Win16/Windows'95/98 Polling Results	183
6.3 Windows NT/2000/XP Polling Results	184
6.4 Unix Polling Results	185
7. Extensions to the Basic Polling Model	185
8. Protecting the Randomness Pool	186
9. Conclusion	188
10. References	189
7. Hardware Encryption Modules	193
1. Problems with Crypto on End-user Systems	195
1.1 The Root of the Problem	196
1.2 Solving the Problem	198
1.3 Coprocessor Design Issues	198
2. The Coprocessor	200
2.1 Coprocessor Hardware	201
2.2 Coprocessor Firmware	202
2.3 Firmware Setup	203
3. Crypto Functionality Implementation	203
3.1 Communicating with the Coprocessor	204
3.2 Coprocessor Session Control	206
3.3 Open vs. Closed-source Coprocessors	207
4. Extended Security Functionality	207
4.1 Controlling Coprocessor Actions	208
4.2 Trusted I/O Path	209
4.3 Physically Isolated Crypto	209
4.4 Coprocessors in Hostile Environments	209
5. Crypto Hardware Acceleration	211
5.1 Conventional Encryption/Hashing	211
5.2 Public-key Encryption	212
5.3 Other Functionality	213
6. Conclusion	213
7. References	214

8. Conclusion.....	217
1. Conclusion.....	219
1.1. Separation Kernel enforcing Filter Rules	219
1.2. Kernel and Verification Co-design.....	219
1.3. Use of Specification-based Testing	220
1.4. Use of Cognitive Psychology Principles for Verification.....	220
1.5. Practical Design.....	220
2. Future Research	221

