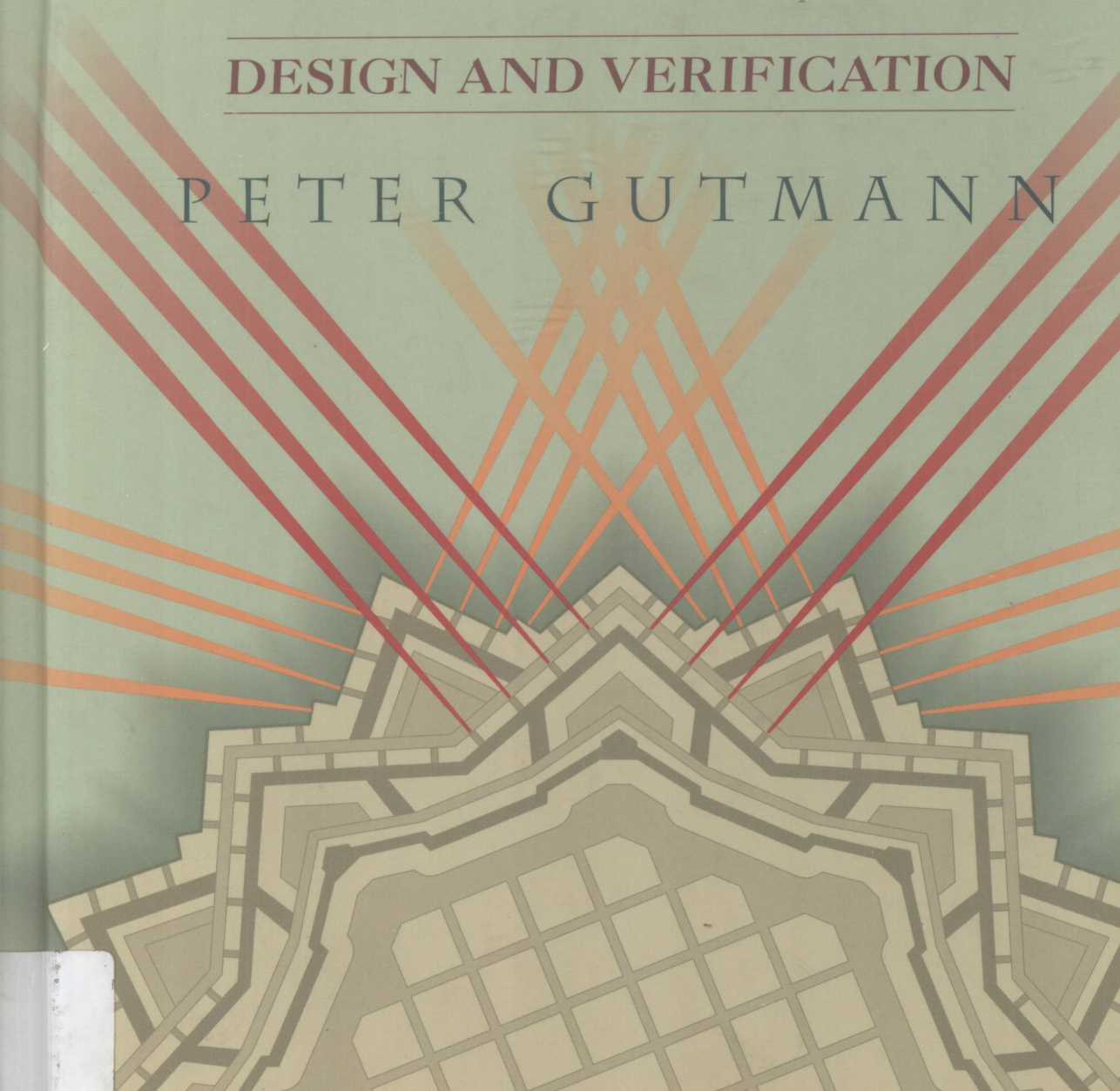


CRYPTOGRAPHIC SECURITY ARCHITECTURE

DESIGN AND VERIFICATION

PETER GUTMANN



TP309
G984

Peter Gutmann

Cryptographic Security Architecture

Design and Verification

With 149 Illustrations



E200400159



Springer

Peter Gutmann
Department of Computer Science
University of Auckland
Private Bag 92019
Auckland
New Zealand
pgut001@cs.auckland.ac.nz
<http://www.cs.auckland.ac.nz/~pgut001/cryptlib/>

Cover illustration: During the 16th and 17th centuries the art of fortress design advanced from ad hoc methods which threw up towers and walls as needed, materials allowed, and fashion dictated, to a science based on the use of rigorous engineering principles. This type of systematic security architecture design was made famous by Sebastien le Prestre de Vauban, a portion of whose fortress of Neuf-Brisach on the French border with Switzerland is depicted on the cover.

Library of Congress Cataloging-in-Publication Data
Gutmann, Peter.

Cryptographic Security Architecture / Peter Gutmann.

p. cm.

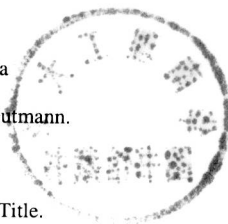
Includes bibliographical references and index.

ISBN 0-387-95387-6 (alk. paper)

1. Computer security. 2. Cryptography. I. Title.

QA76.9.A25 G88 2002

005.8—dc21



2002070742

ISBN 0-387-95387-6

Printed on acid-free paper.

© 2004 Springer-Verlag New York, Inc.

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

SPIN 10856194

Typesetting: Pages created using the author's Word files.

www.springer-ny.com

Springer-Verlag New York Berlin Heidelberg
A member of BertelsmannSpringer Science+Business Media GmbH

Cryptographic Security Architecture

Springer

New York

Berlin

Heidelberg

Hong Kong

London

Milan

Paris

Tokyo

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

Overview and Goals

This book describes various aspects of cryptographic security architecture design, with a particular emphasis on the use of rigorous security models and practices in the design. The first portion of the book presents the overall architectural basis for the design, providing a general overview of features such as the object model and inter-object communications. The objective of this portion of the work is to provide an understanding of the software architectural underpinnings on which the rest of the book is based.

Following on from this, the remainder of the book contains an analysis of security policies and kernel design that are used to support the security side of the architecture. The goal of this part of the book is to provide an awareness and understanding of various security models and policies, and how they may be applied towards the protection of cryptographic information and data. The security kernel design presented here uses a novel design that 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 that has been selected to allow the implementation to be verified in a manner that can reassure an outsider that it does indeed function as required. In addition to producing verification evidence that 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 that is currently impossible with traditional methods. The approach presented here makes it possible to perform verification at this level, something that had previously been classed as “beyond A1” (that is, not achievable using any known technology).

Finally, two specific issues that arise from the design presented here, namely the generation and protection of cryptovariables such as encryption and signature keys, and the application of the design to cryptographic hardware, are presented. These sections are

intended to supplement the main work and provide additional information on areas that are often neglected in other works.

Organisation and Features

A cryptographic security architecture constitutes the collection of hardware and software that 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 book 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 that controls access to security-relevant objects and attributes based on a configurable security policy. Layered over the kernel are various objects that 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 that 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 that it enforces for all objects within the architecture, as covered in the latter part of the chapter.

The kernel itself uses a novel design that 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 that, if they enforce controls at all, hardcode them into the implementation, making it difficult either to change the controls to meet application-specific requirements or to assess and verify them. The approach to enforcing security controls that 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 book instead presents a design and implementation process that has been selected to allow the design to be verified in a manner that can reassure an outsider that it does indeed function as required, something that is practically impossible with a formally verified design. Chapter 5 presents a new approach to building a trustworthy system that 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 that should be both palatable and comprehensible.

In addition to producing verification evidence that 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 that is currently impossible with traditional methods. The approach presented here makes it possible to perform verification at this level, something that had previously been classed as “beyond A1” (that is, not achievable using any known technology). This level of verification can be achieved principally because the kernel design and implementation have 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 that 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 that 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 generating random number for use as cryptovariables. Chapter 6 begins with a requirements analysis and a survey of existing generators, including extensive coverage of pitfalls that 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, and is capable of being built from off-the-shelf hardware controlled by the software that 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 that are not feasible in a general-purpose software implementation. The chapter covers some of the threats that 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 that 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.

Intended Audience

This book is intended for a range of readers interested in security architectures, cryptographic software and hardware, and verification techniques, including:

- Designers and implementers: The book discusses in some detail design issues and approaches to meeting various security requirements.
- Students and researchers: The book is intended to be both a general tutorial for study and an in-depth reference providing links to detailed background material for further research.

Acknowledgements

This book (in its original thesis form) 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 book. 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 that 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. After its completion, the current book form was prepared under the guidance and direction of Wayne Wheeler and Wayne Yuhasz of Springer-Verlag. During the reworking process, Adam Back, Ariel Glenn, and Anton Stiglic provided feedback and suggestions for changes. The book itself was completed despite Microsoft Word, with diagrams done using Visio.

Contents

Preface	vii
Overview and Goals.....	vii
Organisation and Features.....	viii
Intended Audience	x
Acknowledgements	x
1 The Software Architecture	1
1.1 Introduction	1
1.2 An Introduction to Software Architecture.....	2
1.2.1 The Pipe-and-Filter Model.....	3
1.2.2 The Object-Oriented Model	4
1.2.3 The Event-Based Model.....	5
1.2.4 The Layered Model	6
1.2.5 The Repository Model.....	6
1.2.6 The Distributed Process Model.....	7
1.2.7 The Forwarder-Receiver Model.....	7
1.3 Architecture Design Goals	8
1.4 The Object Model	9
1.4.1 User ↔ Object Interaction	10
1.4.2 Action Objects.....	12
1.4.3 Data Containers.....	13
1.4.4 Key and Certificate Containers	14
1.4.5 Security Attribute Containers.....	15
1.4.6 The Overall Architectural and Object Model.....	15
1.5 Object Internals	17
1.5.1 Object Internal Details	18
1.5.2 Data Formats	20
1.6 Interobject Communications	21
1.6.1 Message Routing.....	23
1.6.2 Message Routing Implementation.....	25
1.6.3 Alternative Routing Strategies	26
1.7 The Message Dispatcher	27
1.7.1 Asynchronous versus Synchronous Message Dispatching	30
1.8 Object Reuse	31
1.8.1 Object Dependencies.....	34

1.9 Object Management Message Flow	35
1.10 Other Kernel Mechanisms.....	37
1.10.1 Semaphores	38
1.10.2 Threads.....	38
1.10.3 Event Notification	39
1.11 References.....	39
2 The Security Architecture	45
2.1 Security Features of the Architecture.....	45
2.1.1 Security Architecture Design Goals	46
2.2 Introduction to Security Mechanisms	47
2.2.1 Access Control	47
2.2.2 Reference Monitors.....	49
2.2.3 Security Policies and Models	49
2.2.4 Security Models after Bell–LaPadula.....	51
2.2.5 Security Kernels and the Separation Kernel	54
2.2.6 The Generalised TCB.....	57
2.2.7 Implementation Complexity Issues	59
2.3 The cryptlib Security Kernel.....	61
2.3.1 Extended Security Policies and Models	63
2.3.2 Controls Enforced by the Kernel.....	65
2.4 The Object Life Cycle.....	66
2.4.1 Object Creation and Destruction	68
2.5 Object Access Control.....	70
2.5.1 Object Security Implementation.....	72
2.5.2 External and Internal Object Access	74
2.6 Object Usage Control.....	75
2.6.1 Permission Inheritance	76
2.6.2 The Security Controls as an Expert System	77
2.6.3 Other Object Controls	78
2.7 Protecting Objects Outside the Architecture	79
2.7.1 Key Export Security Features	81
2.8 Object Attribute security	82
2.9 References	83
3 The Kernel Implementation	93
3.1 Kernel Message Processing.....	93
3.1.1 Rule-based Policy Enforcement	93
3.1.2 The DTOS/Flask Approach.....	94
3.1.3 Object-based Access Control	96
3.1.4 Meta-Objects for Access Control.....	98
3.1.5 Access Control via Message Filter Rules.....	99
3.2 Filter Rule Structure.....	101

3.2.1 Filter Rules	102
3.3 Attribute ACL Structure	106
3.3.1 Attribute ACLs	108
3.4 Mechanism ACL Structure	112
3.4.1 Mechanism ACLs	113
3.5 Message Filter Implementation	117
3.5.1 Pre-dispatch Filters	117
3.5.2 Post-dispatch Filters	119
3.6 Customising the Rule-Based Policy	120
3.7 Miscellaneous Implementation Issues	122
3.8 Performance	123
3.9 References	123
4 Verification Techniques	127
4.1 Introduction	127
4.2 Formal Security Verification	127
4.2.1 Formal Security Model Verification	130
4.3 Problems with Formal Verification	131
4.3.1 Problems with Tools and Scalability	131
4.3.2 Formal Methods as a Swiss Army Chainsaw	133
4.3.3 What Happens when the Chainsaw Sticks	135
4.3.4 What is being Verified/Proven?	138
4.3.5 Credibility of Formal Methods	142
4.3.6 Where Formal Methods are Cost-Effective	144
4.3.7 Whither Formal Methods?	145
4.4 Problems with other Software Engineering Methods	146
4.4.1 Assessing the Effectiveness of Software Engineering Techniques	149
4.5 Alternative Approaches	152
4.5.1 Extreme Programming	153
4.5.2 Lessons from Alternative Approaches	154
4.6 References	154
5 Verification of the cryptlib Kernel	167
5.1 An Analytical Approach to Verification Methods	167
5.1.1 Peer Review as an Evaluation Mechanism	168
5.1.2 Enabling Peer Review	170
5.1.3 Selecting an Appropriate Specification Method	170
5.1.4 A Unified Specification	173
5.1.5 Enabling Verification All the way Down	174
5.2 Making the Specification and Implementation Comprehensible	175
5.2.1 Program Cognition	176
5.2.2 How Programmers Understand Code	177
5.2.3 Code Layout to Aid Comprehension	180

5.2.4 Code Creation and Bugs.....	182
5.2.5 Avoiding Specification/Implementation Bugs	183
5.3 Verification All the Way Down	184
5.3.1 Programming with Assertions	186
5.3.2 Specification using Assertions	188
5.3.3 Specification Languages	189
5.3.4 English-like Specification Languages	190
5.3.5 Spec	192
5.3.6 Larch	193
5.3.7 ADL	194
5.3.8 Other Approaches	197
5.4 The Verification Process	199
5.4.1 Verification of the Kernel Filter Rules.....	199
5.4.2 Specification-Based Testing.....	200
5.4.3 Verification with ADL	202
5.5 Conclusion	203
5.6 References	204
6 Random Number Generation.....	215
6.1 Introduction	215
6.2 Requirements and Limitations of the Generator	218
6.3 Existing Generator Designs and Problems.....	221
6.3.1 The <i>Applied Cryptography</i> Generator.....	223
6.3.2 The ANSI X9.17 Generator	224
6.3.3 The PGP 2.x Generator	225
6.3.4 The PGP 5.x Generator	227
6.3.5 The /dev/random Generator	228
6.3.6 The Skip Generator	230
6.3.7 The ssh Generator	231
6.3.8 The SSLeay/OpenSSL Generator.....	232
6.3.9 The CryptoAPI Generator.....	235
6.3.10 The Capstone/Fortezza Generator.....	236
6.3.11 The Intel Generator	238
6.4 The cryptlib Generator	239
6.4.1 The Mixing Function.....	239
6.4.2 Protection of Pool Output.....	240
6.4.3 Output Post-processing	242
6.4.4 Other Precautions	242
6.4.5 Nonce Generation	242
6.4.6 Generator Continuous Tests	243
6.4.7 Generator Verification	244
6.4.8 System-specific Pitfalls	245
6.4.9 A Taxonomy of Generators.....	248
6.5 The Entropy Accumulator.....	249

6.5.1 Problems with User-Supplied Entropy	249
6.5.2 Entropy Polling Strategy	250
6.5.3 Win16 Polling	251
6.5.4 Macintosh and OS/2 Polling	251
6.5.5 BeOS Polling.....	252
6.5.6 Win32 Polling	252
6.5.7 Unix Polling	253
6.5.8 Other Entropy Sources	256
6.6 Randomness-Polling Results.....	256
6.6.1 Data Compression as an Entropy Estimation Tool.....	257
6.6.2 Win16/Windows 95/98/ME Polling Results	259
6.6.3 Windows NT/2000/XP Polling Results	260
6.6.4 Unix Polling Results	261
6.7 Extensions to the Basic Polling Model	261
6.8 Protecting the Randomness Pool.....	263
6.9 Conclusion	266
6.10 References	267
7 Hardware Encryption Modules	275
7.1 Problems with Crypto on End-User Systems	275
7.1.1 The Root of the Problem	277
7.1.2 Solving the Problem.....	279
7.1.3 Coprocessor Design Issues.....	280
7.2 The Coprocessor	283
7.2.1 Coprocessor Hardware	283
7.2.2 Coprocessor Firmware	285
7.2.3 Firmware Setup.....	286
7.3 Crypto Functionality Implementation	287
7.3.1 Communicating with the Coprocessor	289
7.3.2 Communications Hardware.....	289
7.3.3 Communications Software	290
7.3.4 Coprocessor Session Control	291
7.3.5 Open versus Closed-Source Coprocessors.....	293
7.4 Extended Security Functionality	294
7.4.1 Controlling Coprocessor Actions	294
7.4.2 Trusted I/O Path	295
7.4.3 Physically Isolated Crypto	296
7.4.4 Coprocessors in Hostile Environments	297
7.5 Conclusion	299
7.6 References	299
8 Conclusion	305
8.1 Conclusion	305

8.1.1 Separation Kernel Enforcing Filter Rules	305
8.1.2 Kernel and Verification Co-design	306
8.1.3 Use of Specification-based Testing.....	306
8.1.4 Use of Cognitive Psychology Principles for Verification	307
8.1.5 Practical Design	307
8.2 Future Research.....	308
9 Glossary	309
Index.....	317