Understanding Security APIs

Michael K. Bond

University of Cambridge
Computer Laboratory
Emmanuel College

Jan 2004

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Dedication

To Philip Barnes, who could have stopped me exercising my talent at breaking things, but didn't;

and to Clive Spencer-Bentley, who changed my life both in presence and absence.

Acknowledgements

I am indebted to a whole host of people who have inspired, conspired, cooperated, and supported me and this research since it all started back in 2000.

First, I want to thank my supervisor Ross Anderson who started it all, spotted the significance of the results I was getting, and has backed me up at every twist and turn of my journey. Many thanks are also due to Larry Paulson who supervised me for the first year and a half, and lent a balance and perspective to the work which was invaluable when I was immersed in the technical details of a problem.

I would like to thank my comrades from TG1 – George (*"What are we going to do today George? Same thing do we do every day Mike – try to take over the world!"*), Richard, and Markus: countless conversations, remonstrations, arguments, and even sword-fights have helped me settle the truths of this topic. In more recent times Steven, Stephen, Piotr, Andrei and Sergei have all lent their advice, skills and senses of humour to aid my work.

Out in the big wide world of industry, special thanks go to Nicko Van Someren, Peter Landrock and Leendert Van Doorn, who have all been very generous to me. Ernie Cohen gave me some useful pointers and feedback on my crude efforts in formal analysis. Particular thanks in recent times are also due to Todd Arnold and Dave Ritten.

I must thank my mysterious and generally invisible funding bodies, the EPSRC and Marconi – I hope the discoveries in this thesis return at least some of the investment you have made in me. I should also thank my former director of studies, Neil Dodgson, for (presumably) not writing a damning reference for me just after discovering about the 'vodka in exam' incident!

In my personal struggle to survive this Ph.D. experience I am lost for superlatives to describe the unshaking support I've had from Marianne and from my family. My father in particular put up the money until my funding came through, and has read or heard about (and completely understood) every idea as it arrived hot off the press. Thanks also to numerous friends who have watched with interest and kept me sane: Martin, Mary, Matt, Joe, Steve to mention but a few. A special thank you to Sheila, who I swear is more interested in my work than I am, and who has been a continual source of support and a great friend.

Finally I must thank Jol – I might have quit academic Security API research had the size of the research community in this field not suddenly doubled. Onwards to the future!

Understanding Security APIs

Michael K. Bond

Summary

This thesis introduces the newly-born field of Security API research, and lays the foundations for future analysis, study, and construction of APIs. Security APIs are *application programmer interfaces* which use cryptography to enforce a security policy on the users of the API, governing the way in which they manipulate sensitive data and key material.

The thesis begins by examining the origins and history of Security APIs, and that of Hardware Security Modules – tamper-resistant cryptographic processors which implement the APIs, the study of which goes hand-in-hand with this research. The major manufacturers and their products are covered, and commentaries draw together a few of the more important themes that explain why Security APIs are the way they are today.

The significant original contribution at the heart of the thesis is a catalogue of new attacks and attack techniques for Security APIs. These attacks have had substantial impact on the Security API design community since their original publication. For example, the related-key "meet-in-the-middle" attack compromised every HSM analysed, and differential protocol analysis compromised all financial Security APIs. Historic attacks and brief explanations of very new unpublished attacks are also included.

The thesis goes on to provide a body of advice for Security API design, consisting of heuristics and discussions of key issues, including those most pertinent to modern HSMs such as *authorisation* and *trusted paths*. The advice is linked in with the cautionary tales of Security API failures from the previous chapters.

As the thesis is opening a new field of academic research, its main objective is to build *understanding* about Security APIs, and the conclusions drawn are open-ended and speculative. The different driving forces shaping the development of Security APIs are considered, and Trusted Computing is identified as central to the shaping of Security APIs and to the future relevance of this thesis.

Contents

Chapter 1

Introduction

In today's world, the user of a computer is no longer the only person with a stake in the operation of her device. Modern PCs, mainframes and networks are not just designed to support multiple users, but also operate with code, secrets and data produced by many organisations, each with their own interests. Considering all the conceptual components, today's computers belong to no single individual. Each PC is an autonomous device that tries to enforce many users' policies on the software inside it, and many software packages' policies on the actions of the user.

A Security API is an application programmer interface that uses cryptography to enforce a security policy on the interactions between two entities. Security APIs have many shapes and forms, but the most common API is fairly recognisable. One entity – a corporation – will entrust a piece of software with valuable corporate data, and a policy which carefully controls how users of the software may access that data. Typically the users are the corporation's own less senior employees, but more recently corporations providing services have extended the use cases to include their clients as well.

In the last three years, the Security API industry has been rocked by the discovery of a raft of new attacks that work by executing unexpected sequences of commands to trick the API into revealing secrets in a way that the designers could not possibly have intended. These attacks have turned the financial PIN processing industry on its head, and revealed just how much uncertainty there is about the security of the APIs that are protecting our most valuable key material and secrets.

We are now on the dawn of a new age of control, where individuals and corporations are putting more and more trust into computers as policy enforcers – they have the potential to implement multiple sophisticated policies simultaneously. However, the level of protection of an entity's interests by a policy is only as good as the policy itself, and the translation of policies into efficient cryptographic mechanisms is an undeveloped and poorly understood field.

As Microsoft spearheads the ubiquitous distribution of Security APIs over the home and corporate community with their NGSCB (formerly Palladium) project, it has

become more important than ever for us to develop an *understanding of Security APIs*. We need to understand their abilities and limitations, and the secrets of good design. Only then will we be able to judge whether devolution of policy enforcement to computers is in the best interests of our society.

The aim of this thesis is to build an understanding of Security APIs at both technical, operational and social levels. It makes an in-depth examination of the abilities and weaknesses of today's Security APIs, drawing strongly from financial security applications, and the study of tamper-resistant hardware. It exposes the shortcomings of existing APIs, which are failing in some cases to enforce even the simplest of policies. It examines API failures both in simple two party cases, and complex heterogeneous environments. It proposes starting points for methods of analysis, brings to bear previous innovation from other fields of computer science, and provides the first tentative advice on good design practice.

If Security APIs do become important and ubiquitous, this thesis should lay the groundwork for a new and worthwhile field of study.

1.1 How to Read this Thesis

Chapter 2 introduces Security APIs and describe their origins, then chapter 3 describes the discovery and development of API attacks. Readers who are unfamiliar with the concept of an API attack should find the simplified explanations of chapter 3 particularly helpful to study before tackling the meat of the thesis in chapter 7. Note also that there is a glossary at the back of the thesis to help with all the TLAs.

Chapters 4, 5 and 6 introduce the applications and industrial background to Security APIs in considerable detail. As this thesis is all about *understanding Security APIs*, taking time to read about the background is as important as learning about the attacks, but nevertheless some people may prefer to read more about attacks before coming back to look at the big picture, and should skip past these chapters straight to the analysis of Security API failures in chapter 7.

Chapter 7 is the heart of the thesis. Useful abstractions for analysis of Security APIs are first introduced, followed by a catalogue of attack techniques and actual instances (fairly heavy going!). Chapter 7 finishes with a discussion of formal methods for analysis of Security APIs. Chapter 8 then draws together the common themes behind the failures described in chapter 7 into wisdom and heuristics for good Security API design. It then goes on to discuss the design issues facing modern APIs, in particular *authorisation* and *trusted paths*.

Finally, chapters 9 and 10 speculate about the future of Security APIs, and draw some general conclusions about what it will take for API design and research to mature, and whether or not they will have an important role to play in the future of computer security.

Whatever you intend to get out of this thesis, the author's intention is for you to ***understand*** Security APIs, so you are strongly encouraged to explore, and see the field from as many different perspectives as possible.

1.2 Schedule of Work

The material in this thesis is a subset of research performed by the author between October 2000 and October 2003 in the Security Group at the Computer Laboratory, University of Cambridge. Much of the material has been published in the following papers:

- M. Bond,"Attacks on Cryptoprocessor Transaction Sets", CHES Workshop 2001, Paris, Springer LNCS 2162, pp. 220–234
- M. Bond, R. Anderson, "API-Level Attacks on Embedded Systems", IEEE Computer, Oct 2001, Vol 34 No. 10, pp. 67–75

  All the attacks except the 'XOR to Null Key Attack' were discovered by the author. Anderson co-authored the text.
- R. Clayton, M. Bond, "Experience Using a Low-Cost FPGA Design to Crack DES Keys", CHES Workshop 2002, San Francisco, Springer LNCS 2523, pp. 579–592

  The attack described in the paper was the author's work. Clayton built the majority of the hardware DES cracker that served to accelerate attack, and co-authored the text.
- R. Anderson, M. Bond, "Protocol Analysis, Composability and Computation", Computer Systems: Papers for Roger Needham, Jan 2003

  The author discovered the decimalisation table attack described; Anderson was the primary author of the text.
- M. Bond, P. Zielinski, "Decimalisation Table Attacks for PIN Cracking", Computer Laboratory Technical Report no. 560, Feb 2003

  The author discovered the decimalisation table attack; Zielinski optimised the attack to be adaptive, and co-authored the text.

In addition the work has been presented by invitation at seminars and conferences:

- "Attacks on Cryptoprocessor Transactions Sets"

  Computer Laboratory, University of Cambridge, UK
  COSIC , Katholieke Universitat Leuven, Belgium
  CHES Workshop, Paris, France

- "A Low-cost Hardware Birthday Attack on DES"

  Computer Laboratory, University of Cambridge, UK

- "First Steps in Cryptoprocessor API Analysis"

  "Specification and Verification of Secure Cryptographic Protocols" Workshop, Schloss Dagstuhl, Germany

- "The Benefits and Pitfalls of Cryptographic Hardware"

  Information Security Forum 2002, London, UK

- "The Hazards of Security API Design"

  BCS Advanced Programming Seminar, London, UK

- "Experience Using a Low-Cost FPGA to Crack DES Keys"

  CHES Workshop 2002, San Francisco, USA

- "The Hazards of Security API Design : Special Edition"

  Security PIC, IBM TJ Watson Research Labs, NY, USA

- "Hardware Security Modules : Benefits and Pitfalls"

  EEMA Information Security Solutions Europe conference, France

- "Model Checking Cryptoprocessors : Why I like the British Museum"

  Computer Laboratory, University of Cambridge, UK

- "Differential Protocol Analysis and API-Level Attacks"

  Invited Talk, Security and Protection of Information 2003
  Brno, Czech Republic

- "Security APIs – Digital Battlefields"

  Information Security Group, University of Bristol, UK

The aspects of the research relating directly to ATM security have been at the centre of a number of international news stories:

- In November 2001, the attack on the IBM 4758 CCA in section 7.3.7 was the subject of an exclusive report on BBC Television's 'Newsnight' topical news programme, and was featured in the Financial Times, Daily Telegraph and Independent newspapers. Subscribers to Reuters and Associated Press of America also printed versions of the story. The story was followed up by topical news programmes on the radio internationally.
- In February 2003, the decimalisation table attack in section 7.3.10 was crucial evidence in the South African court case "Diners Club SA vs. Singh", in which the author gave expert testimony. Diner's Club International were alleged to be attempting to stifle disclosure of the attack, and the story ran first in the Sunday Times, then in the Financial Times and in the Guardian. Various international radio coverage followed. The technical report *"Decimalisation Table Attacks for PIN Cracking"* was downloaded over 120000 times in the week the story broke.
- In December 2003, the banking security attacks were the subject of a full page article in the Sunday Business Post, Ireland, followed up by a radio interview on East Coast FM.
- The research has also featured several times in New Scientist, and in relevant trade journals and newspapers, and on numerous online news sites.

Finally, the research has already had several tangible effects upon the security of existing APIs:

- The IBM 4758 CCA attack (section 7.3.7) prompted the release of the CCA Version 2.41.
- The attack on the Prism TSM200 (section 7.3.11) was reported to Prism and fixed.
- The theoretical attack on one-shot authorisation using nCipher smartcards (section 8.4.2) caused a reduction in the timeout period for operator cardset insertion.
- There is circumstantial evidence that the work published in "Attacks on Cryptoprocessor Transaction Sets" prompted a substantial redesign of Atalla's API, including the introduction of a new key block format that addressed 3DES binding issues.
- The attacks have several times been brought (indirectly) before the ANSI X9.8 financial standards committee and are to a limited extent being taken into account in the revised versions of the standard.

Chapter 2

Origins of Security APIs

There are probably several hundred API designers at work today, and this figure will grow as Security APIs become ubiquitous. However, their origins were in the hands of only a few people: small teams of engineers and scientists, first in the US military, and then in Automated Teller Machine (ATM) security. This brief survey attempts to show a unifying logic behind the development of Security APIs (of course, history is not as simple as this, and the individual motivations and perspectives of the parties involved will not necessarily conform).

2.1 Beginnings

Security APIs were born in an age when dedicated hardware was *necessary* in order to do cryptography. The major algorithm of the 70s and 80s – DES – was designed to be efficient to implement in hardware. The computers of the day needed a simple command set to govern communication with this hardware: it might consist of a command to set the key, a command to encrypt, and one to decrypt. Here was the first cryptographic API, though it could not yet be considered a Security API, as there was no policy on usage to enforce.

As digital cryptographic equipment became smaller and more portable, the military adopted in increasing numbers of roles, such as to secure battlefield communications links. Whilst cipher rooms in embassies abroad would have good physical security, the dynamic environment of the battlefield could not offer crypto equipment or the keys within it any long-term safety. Tamper-resistance provided a partial solution to the problems of battlefield capture.

The principle of tamper-resistance was part of military thinking for some time. Military analogue and digital equipment was built with custom-marked components for many years, which made reverse-engineering and re-commissioning of stolen or damaged equipment harder. Tamper-resistance found further applications in nuclear command and control. This was an issue of particular concern in the late 1960s;

it spawned the development of control equipment to make it difficult for soldiers working with nuclear ordinance to deploy weapons without proper authorisation. At the time, the major perceived threat to nuclear ordinance was that of regime change in allied countries entrusted with the weapons. These controls centred around Permissive Action Links (PALs): electronics in the ordinance which enforced access control to the mechanical core of the device.

The conjunction of cryptographic functionality with the security policies inherent in tamper-resistance and access control gave rise to Security APIs as we know them today. However, it took the 'killer app' of ATM security to introduce Security APIs to the commercial world.

2.2 The 'Killer App'

The ATMs which are now a common feature of everyday life have taken several decades to develop. They started out as cash-dispensing devices purely for increased speed of service, and are now quite independent multi-purpose devices that provide banking services in hostile environments. As the international ATM network grew, the concentration points of the networks – the bank mainframes – held more and more valuable secrets, and enforcement of policy on usage of their APIs became crucial.

Some of the early adopters had the security relevant code integrated with application code inside their mainframes. Thus, the highly sensitive keys used to determine customer PINs were stored in the RAM of the mainframe, and there would be a number of employees with sufficient access to see them, if motivated. This risk was then mitigated by identifying and isolating the security-relevant code from the rest of the banking applications, yet within the same mainframe. Some of the first Security APIs were created here – between software modules written by different programmers.

However, as the overall complexity of operating systems rose, it became clear that the keys were at risk from the maintenance staff, who needed full access to all parts of a mainframe to maintain and upgrade the software. Furthermore, it seemed unwise to put keys that were vital to the economic security of the bank at risk from bugs in third-party code (not designed with security considerations) that could compromise the security relevant code. It could also be argued that all mainframe operating systems were so complex as to make bug-free implementation unattainable, regardless of whether they had been designed with security in mind.

The concept of a "Hardware Security Module" (HSM) or "Cryptoprocessor" arose in part to address this problem. These implemented the existing Security APIs, but with physical separation of the computing platforms, enabling separate administration of the devices. To be successful, these modules had to fulfil two aims:

- To isolate and define the code which was security relevant, and to physically separate it into a device not under control of the system administrators.
- To minimise the amount of security relevant code, and to keep the rules governing its use as simple as possible, so that it would be easier to avoid bugs.

In practical terms, financial HSMs were entrusted with the secret keys used for deriving customer PINs from account numbers. Their policy was to ensure that all PINs stayed in encrypted form, and clear PINs would never be available to host programmers (they could only be sent in the clear to a secure printer locked in a cage). As HSMs began to move away from mainframes into high end servers, and onto the desks of operators, physical tamper-resistance was added to the logical Security API. These financial APIs are still in existence today, are used by banks all over the world, and have not been substantially redesigned since their first inception.

2.3 The Present

The next significant advance was when Security APIs were used to enforce more complex policies than just protecting secrecy of keys. The introduction of electronic credit tokens for prepayment services such as electricity meters (and most recently mobile phones) gave rise to a new application – credit dispensing. A Security API could have a policy allowing the repeated execution of a credit dispensing command, deducting an amount from a *credit counter* each time.

The mid-nineties has seen new applications for Security APIs: securing *certification authorities* and *internet payment infrastructures*. In addition, embedded devices such as smartcards, which are much smaller and more numerous than hardware security modules (HSMs), are beginning to add policy enforcement to their APIs.

So security and cryptography are no longer restricted to the military and banking worlds. Now that corporations and individuals have seized the initiative, wherever competition arises in life, the tools of the security world can go to work protecting people's interests. Security considerations are about to become ubiquitous: every major operating system ships with substantial crypto facilities, and so do mobile phones. With the rise of wireless communications, we will see "home area networks" becoming established, and even washing machines and fridges will use embedded cryptographic controllers to communicate securely over the airwaves.

–And with every application that uses security comes a 'Security API', forming the interface between the application code and the security relevant code.

2.4 Key Dates

Year	Events
1960	Development of Permissive Action Links (PALs) started, to protect nuclear weapons
1973	Bell-LaPadula Multi-Level Security policy proposed
1974	IBM launches 3600 series finance system supporting the deployment of the first ATMs
1976	DES released by NBS (NBS was predecessor to NIST)
1977	**IBM launches first crude HSM: 3845 DES encryption unit**
1978	RSA invented
1985	**VISA Security Module introduced**
1987	IBM introduces 4755 HSM, including μABYSS tamper mesh
1989	**IBM introduces Transaction Security System (TSS), which includes the "Common Cryptographic Architecture" (CCA)**
1989	Visa Security Module clones start to appear on market
1991	Eracom enters HSM market, producing "secure application modules"
1992	Longley and Rigby publish first work on automated analysis of an HSM
1994	Chrysalis-ITS founded
1993	*"Why Cryptosystems Fail"* published by Anderson
1995	**NIST FIPS 140-1 security module validation programme started**
1996	nCipher founded
1996	**IBM launches 4758** (as replacement for 4755)
1996	**RSA launches PKCS#11 standard**
1997	Rainbow enters HSM market
1998 Nov	IBM 4758 becomes the first HSM validated FIPS 140-1 Level 4
1999 Jan	Racal SPS introduces "Websentry" HSM
2000	**First academic research into Security APIs**
2000	Rainbow launches CryptoSwift HSM
2000 Dec	Racal SPS rebranded Thales
2001 Apr	nCipher launches SEE (Secure Execution Engine), allowing custom code to be run on their platform
2001 May	***"Attacks on Cryptoprocessor Transaction Sets"* published at CHES Workshop in Paris**
2001 Nov	Clulow discovers information leakage attacks
2002	nCipher goes public; enters financial HSM market
2003 Jan	***"Decimalisation Table Attack"* and information leakage attacks published by Bond and Clulow**
2003	Chrysalis-ITS taken over twice in one year
2003	Thales finally launches successor to RG7000
2003 Oct	nCipher launches next generation netHSM

Chapter 3

Origins of Security API Attacks

This chapter summarises the history of discovery and publication of API attacks on HSMs. It explains what an API attack is, how the attacks were discovered, and shows the core ideas behind them. The attacks described have been built up into the toolkit described in section 7.2. For simplicity, the story of their discovery is told only in the context of financial security systems, though the same techniques have been successfully applied to a range of other non-financial applications.

3.1 Early Security API Failures

Anderson was one of the first to introduce hardware security module failures to the academic community. After spending a number of years working in financial security, in 1992 he became involved in a class action law suit in the UK, pertaining to so-called 'phantom withdrawals': unexplained losses of money from customer accounts. Anderson condensed much of his understanding into an academic paper "Why Cryptosystems Fail" [3]. This paper focussed on the known failure modes of ATM banking systems, including several procedural and technical failures in the use of security modules. A cryptographic binding error was typical of the failures Anderson described:

"One large UK bank even wrote the encrypted PIN to the card strip. It took the criminal fraternity fifteen years to figure out that you could change the account number on your own card's magnetic strip to that of your target, and then use it with your own PIN to loot his account."

However, the paper stopped short of including a description of what we would nowadays call an API attack. Several years later, Anderson described in "Low Cost Attacks on Tamper Resistant Devices" [6], an incident where a dangerous transaction was deliberately added to a security module API.

Many banks at the time calculated customer PINs by encrypting the customer's Primary Account Number (PAN) with a secret key, then converting the resulting

ciphertext into a four digit number. If customers wished to change their PIN, the bank stored an offset representing the difference between the customer's new and old PIN in their database. For example, if the customer's issued PIN was `3566` and she changed it to `3690`, the offset `0134` would be stored.

One bank wished to restructure their customer PANs, maybe to make space for future expansion. Unfortunately, changing the PAN would change the original PIN issued to customers, and the bank did not wish to force all its customers to accept new PINs. The bank commissioned a security module transaction that would adjust all the stored offsets so that a customer's account number could change, yet each could retain the PIN he or she had chosen. The manufacturer produced a transaction of the following form, warning that it was dangerous and should only be used to perform a batch conversion, then removed from the API.

```
Host -> HSM  : old_PAN , new_PAN , offset
HSM  -> Host : new_offset
```

Somehow the warnings were forgotten, and the transaction was never removed from the API. A year or so later, a programmer spotted how this transaction might be abused. If he fed in his own account number as the `new_PAN`, the command would duly calculate and return the difference between any customer's issued PIN and his own original PIN! In the published paper, Anderson characterised this as a protocol failure.

In 2000 Anderson gave a talk at the Cambridge Security Protocols workshop, titled "The Correctness of Crypto Transaction Sets" [1]. He re-iterated a description of the above failure, which pertained to a single bad transaction, but this time he asked the question: "So how can you be sure that there isn't some chain of 17 transactions which will leak a clear key?".

The idea of an API attack was born as ***an unexpected sequence of transactions which would trick a security module into revealing a secret in a way the designers couldn't possibly have intended***. Shortly afterwards Anderson took a second look at the API of the 'VISA Security Module' and came up with an attack.

3.2 A Second Look at the Visa Security Module

The 'VISA Security Module' (VSM) was one of the earliest financial HSM designs, which VISA commissioned to improve PIN processing security, so that member banks might be encouraged to permit processing of each other's customer PINs. It was a large metal box that talked to a bank mainframe via an RS232 or IBM channel interface. No pictures of the VSM are currently in the public domain, but the RG7000 pictured in figure 3.1 is very similar.

Figure 3.1: The RG7000 Hardware Security Module

3.2.1 XOR to Null Key Attack

Until recently ATMs had to support offline operation, so when banks set up new ATMs, they needed a way to securely transfer the *PIN derivation keys* used to calculate customer PINs from PANs. The VSM used a system of dual control to achieve this. The idea was that two service engineers would each take one *component* of a master key to the ATM, and enter it in. Once both components were entered, the ATM could combine the components using the XOR function. The resulting 'Terminal Master Key' (TMK) would be shared with the VSM and could be used for communicating all the other keys. A transaction was first run twice at the VSM to generate the components:

```
HSM -> Printer : TMK1                (Generate Component)
HSM -> Host    : { TMK1 }Km

HSM -> Printer : TMK2                (Generate Component)
HSM -> Host    : { TMK2 }Km
```

The VSM only had very limited internal storage, yet there might be many different ATMs it needed to hold keys for. The paradigm of working with encrypted keys evolved: instead of keeping keys internally, the VSM only held a few master keys, and other keys were passed in as arguments to each transaction encrypted under one of these master keys. So, in response to the above transaction, the VSM returned an *encrypted copy* of the component to the host computer, encrypted under its master key, `Km` (and of course printed a clear copy onto a special sealed mailer for the

service engineer). In order for the VSM to recreate the same key as the ATM, it had a command to XOR two encrypted components together, as shown in figure 3.2.

```
Host -> HSM  : { TMK1 }Km , { TMK2 }Km          (Combine Components)
HSM  -> Host : { TMK1 ⊕ TMK2 }Km
```

The attack

```
Host -> HSM  : { TMK1 }Km , { TMK1 }Km          (Combine Components)
HSM  -> Host : { TMK1 ⊕ TMK1 }Km
```

```
TMK1 ⊕ TMK 1 = 0
```

Figure 3.2: The XOR to Null Key Attack

Anderson made the following observation: if the same component is fed in twice, then because the components are combined with XOR, a key of binary zeroes will result. This known key could then be used to export the *PIN derivation key* in the clear. Anderson described a slightly more complex completion of the attack in [1] than was strictly necessary, but the core idea was the same. This attack was the first true Security API attack, as (unlike the offset calculation attack) it was unintentional, and was composed of more than one transaction. In this thesis, it is named the "XOR to Null Key Attack", and is described fully in section 7.3.1.

3.2.2 Type System Attack

In late 2000, working with Anderson, the author examined the transaction set and found that there were more vulnerabilities: the VSM also had problems with keeping keys used for different purposes separate. The *Terminal Master Keys* used to send other keys to ATMs, and the *PIN Derivation Keys* used to calculate customer PINs were stored by the VSM encrypted with the same master key – `Km`. Two example transactions using these keys are shown below. `PDK1` is a PIN derivation key, and `TMK1` is a terminal master key.

The first transaction encrypts a customer PAN with the PIN derivation key, but sends the PIN to a secure printer (for subsequent mailing to the customer); the second transaction encrypts the PIN derivation key under a TMK belonging to an ATM. Though they perform quite different functions which are not connected, their inputs were sent in under the same master key.

```
Host -> HSM     : PAN , { PDK1 }Km              (Print PIN Mailer)
HSM  -> Printer : { PAN }PDK1
```

```
Host -> HSM     : { PDK1 }Km , { TMK1 }Km       (Send PDK to ATM)
HSM  -> Host    : { PDK1 }TMK1
```

However, the designers did recognise a clear difference between 'Terminal Communications' keys (TCs) and PIN derivation keys or TMKs. `TC1` is a terminal communications key, and `Km2` is a second master key that was used to encrypt keys of this type, keeping them separate from the rest. They were kept separate because terminal communications keys were not considered to be as valuable as PIN derivation keys – and there needed to be a transaction to enter a chosen TC key.

```
Host -> HSM    : TC1                                (Enter clear TC Key)
HSM  -> Host   : { TC1 }Km2
```

TCs needed to be communicated to ATMs in the same way as PIN derivation keys, so there was a command that worked in a very similar way, encrypting the chosen TC under a chosen TMK corresponding to a particular ATM.

```
Host -> HSM     : { TC1 }Km2 , { TMK1 }Km       (Send TC Key to ATM)
HSM  -> Host    : { TC1 }TMK1
```

However, the author spotted that when these two transactions were used together, given the lack of differentiation between PIN derivation keys and TMKs, there was a simple attack. It was to enter in a customer PAN, claiming it to be a TC key, and substitute a PIN derivation key for a TMK in the "send to ATM" transaction.

The Attack

```
Host -> HSM  : PAN                                (Enter clear TC Key)
HSM  -> Host : { PAN }Km2

Host -> HSM  : { PAN }Km2 , { PDK1 }Km          (Send TC Key to ATM)
HSM  -> Host : { PAN }PDK1
```

Of course, `{ PAN }PDK1` is simply the customer's PIN. The full details of this attack are in section 7.3.2. Just like Anderson's 'XOR to Null Key Attack', this vulnerability had gone unnoticed for over a decade. How many more attacks were waiting to be found?

3.3 Development of the Attack Toolkit

3.3.1 Meet-in-the-Middle Attack

The author began a systematic exploration of the VSM API, and also examined the financial API for IBM's 4758 HSM, called the Common Cryptographic Architecture (CCA). The CCA manual was available on the web [26], and when the author studied it, a number of new attack techniques rapidly emerged.

The author observed that both the CCA and the VSM had transactions to generate 'check values' for keys – a number calculated by encrypting a fixed string under the key. When keys were exchanged between financial institutions in components, these check values were used to ensure that no typing mistakes had been made during key entry. The input to the check value encryption was usually a block of binary zeroes.

```
Host -> HSM  : { TMK1 }Km                          (Generate Check Value)
HSM  -> Host : { 0000000000000000 }TMK1
```

Another intriguing feature was that both HSMs stored their keys on the host computer, and only held master keys internally. Due to this external storage, a user could generate an almost unlimited number of conventional keys of a particular type. It was well known that the check values could be used as known plaintext for a brute force search to find a key, but a full search of the 56-bit DES key space was considered prohibitively expensive. But what if the attacker did not need to search for a particular key, but if any one of a large set would suffice? The attack went as follows:

1. Generate a large number of terminal master keys, and collect the check value of each.
2. Store all the check values in a hash table
3. Perform a brute force search, by guessing a key and encrypting the fixed test pattern with it
4. Compare the resulting check value against all the stored check values by looking it up in the hash table (an $O(1)$ operation).

With a 2^{56} keyspace, and 2^{16} target keys, a target key should be hit by luck with roughly $2^{56}/2^{16} = 2^{40}$ effort. The author named the attack the 'meet-in-the-middle' attack with reference to how the effort spent by the HSM generating keys and the effort spent by the brute force search checking keys meet-in-the-middle[1]. The time-memory trade-off has of course been described several decades ago, for example

[1]The "meet-in-the-middle" terminology is not to be confused with cryptographic meet-in-the-middle attacks on block ciphers. More appropriate terminology from published literature includes 'key collision' and 'parallel key search'.

in the attack against 2DES proposed by Diffie and Hellman [19], neither is the idea of parallel search for multiple keys new (Desmedt describes parallel key search machine in [18]). However, it seems the author was the first to apply the technique to HSMs. It was extremely successful, and compromised almost every HSM analysed – sections 7.2.2, 7.3.3 and 7.3.7 have more details.

3.3.2 3DES Key Binding Attack

In the nineties, financial API manufacturers began to upgrade their APIs to use triple-DES (3DES) as advancing computing power undermined the security of single DES. IBM's CCA supported two-key 3DES keys, but stored each half separately, encrypted under the master key in ECB mode. A different *variant* of the master key was used for the left and right halves – achieved by XORing constants representing the types `left` and `right` with the master key `Km`.

```
Host -> HSM  : { KL }Km⊕left , { KR }Km⊕right , data   (Encrypt)
HSM  -> Host : { data }KL|KR
```

The CCA also had support for single DES in a special legacy mode: a 'replicate' 3DES key could be generated, with both halves the same. 3DES is encryption with $K1$, followed by decryption with $K2$, then encryption with $K1$, so if $K1 = K2$ then $E(K1, D(K1, E(K1, data))) = E(K1, data)$, and a replicate key performs exactly as a single DES key.

```
Host -> HSM  : (Generate Replicate)
HSM  -> Host : { X }Km⊕left , { X }Km⊕right
```

The flaw was that the two halves of 3DES keys were not bound together with each other properly, only separated into left and right. There was a clear CRC of the key token, but this was easily circumvented. A large set of replicate keys could be generated and cracked using the meet-in-the-middle attack, then a known 3DES key could be made by swapping the halves of two replicate keys. This known key could then be used to export other more valuable keys.

```
Host -> HSM  : (Generate Replicate)
HSM  -> Host : { X }Km⊕left , { X }Km⊕right

Host -> HSM  : (Generate Replicate)
HSM  -> Host : { Y }Km⊕left , { Y }Km⊕right
```

Known key : `{ X }Km⊕left , { Y }Km⊕right`

This key binding attack effectively reduced the CCA's 3DES down to only twice as good as single DES, which was by then widely considered insufficient. Several attacks exploiting the key binding flaw are described in sections 7.3.6 and 7.3.7.

The attack techniques and implementations in the last few sections were published at the "Cryptographic Hardware and Embedded Systems" workshop in Paris in 2001 [8], and later in IEEE Computer [9]. The CHES paper inspired Clulow to examine the PIN verification functionality of financial APIs more closely, and he discovered half a dozen significant new attacks, which he detailed in his MSc thesis "The Design and Analysis of Cryptographic APIs for Security Devices" [15].

3.3.3 Decimalisation Table Attack

In late 2002 the author and Clulow independently made the next significant advance in attack technology – the discovery of information leakage attacks. Clulow had discovered the problems an entire year earlier, but was unable to speak publicly about them until late 2002, when he gave seminars at RSA Europe, and the University of Cambridge. Early in the next year the author published details of the 'decimalisation table attack', and Clulow published his M.Sc. thesis.

The decimalisation table attack (explained fully in section 7.3.10) exploited flexibility in IBM's method for calculating customer PINs from PANs. Once the PAN was encrypted with a PIN derivation key, it still remained to convert the 64-bit binary block into a four digit PIN. A natural representation of the block to the programmers was hexadecimal, but this would have been confusing for customers, so IBM chose take the hexadecimal output, truncate it to the first four digits, then decimalise these using a lookup table, or *'decimalisation table'*, as shown in figure 3.3.

```
Account Number        4556 2385 7753 2239
Encrypted Accno       3F7C 2201 00CA 8AB3

Shortened Enc Accno   3F7C

                  0123456789ABCDEF
                  0123456789012345

Decimalised PIN       3572
```

Figure 3.3: IBM 3624-Offset PIN Generation Method

Originally the decimalisation table was a fixed input – integrated into the PIN generation and verification commands, but somehow it became parameterised, and by the time the VSM and CCA APIs were implemented, the decimalisation table was

an input that could be specified by the user. If a normal PIN verification command failed, it discounted a single possibility – the incorrect guess at the PIN. However, if the decimalisation table was modified, much more information could be learnt. For example, if the user entered a trial PIN of `0000`, and a decimalisation table of all zeroes, with a single `1` in the `7` position – `0000000100000000` – then if the verification succeeded the user could deduce that the PIN did not contain the digit `7`. Zielinski optimised the author's original algorithm, revealing that PINs could be determined with an average of 15 guesses [10].

3.4 Attacks on Modern APIs

Many of today's Security APIs have been discovered to be vulnerable to the same or similar techniques as those described in this chapter. However, there are some more modern API designs which bear less resemblance to those used in financial security applications. In particular, the main issues relating to the security of PKI hardware security modules are *authorisation* and *trusted path*. These issues have only very recently been explored, and there have been no concrete attacks published. Chapter 8 includes a discussion of the issues of authorisation and trusted path, and describes several hypothetical attacks.

Finally, if the reader is already thoroughly familiar with the attacks described in this chapter, attention should be drawn to several brand new Security API attacks which have been outlined in section 7.3.12, which were developed by the author as a result of analysis of nCipher's payShield API.

Chapter 4

Applications of Security APIs

This chapter gives an overview of the people and organisations that work with Security APIs, and discusses their current applications. Some observations are made on the interactions between the different parties in the industry, and upon when Security APIs should and should not be used.

4.1 Automated Teller Machine Security

ATMs were the 'killer application' that got cryptography into wide use outside of military and diplomatic circles. Cryptography was first used following card forgery attacks on early machines in the late 1960s and early 1970s, when IBM developed a system whereby the customer's *personal identification number* (PIN) was computed from their account number using a secret DES key, the *PIN derivation key*. This system was introduced in 1977 with the launch of the 3614 ATM series [25] and is described further in [2, 3]. Along with electronic payments processing, it is one of the highest volume uses of tamper-resistant hardware.

HSMs are today used to control access to the PIN derivation keys, and also to keep PINs secret in transit through the network:

- Acquisition of PINs inside the ATMs is handled by special HSMs integrated into the keypads, normally built around a secure microcontroller such as a Dallas DS5002.
- Verification requests and responses are then transferred from the ATM across a network of 'switches', each having a conventional HSM (such as an RG7000) translating between encryption and MAC keys on incoming and outgoing links.
- PIN verification requests are finally resolved by an HSM at the issuing bank.
- There may also be HSMs at card embossing and mass PIN mailing sites which banks occasionally contract out to help with high volume reissue of cards.

Because the ATM network is link based architecture, rather than end-to-end, the increasing user-base increases the encryption workload several fold.

4.1.1 Targets of Attack

The ATM system security in banks has a quite conventional threat model which shares aspects with that of many other data-processing and service-providing corporations, because there are no secrets which are absolutely *mission-critical*. Whilst the keys used for deriving PINs from account numbers are extremely valuable, even a theft of tens of millions of pounds is not enough to collapse a bank for financial reasons (their brand name, however, can certainly be considered mission-critical).

The crucial secret is the customer PIN. Sometimes PIN information exists as explicit data which must be kept secret from an attacker, and sometimes it is kept as a PIN derivation key. The authorisation responses sent to ATMs, although not secret, are also valuable. If the integrity of these responses can be compromised, and a *no* turned to a *yes*, money can be withdrawn from a particular cash machine, without knowledge of the correct PIN. Service codes for disabling ATM tamper-resistance, or for test dispensing of cash are also of course valuable and must be kept secret.

It is actually quite easy to quantify the financial loss associated with PIN theft. 'Velocity checking' limits mean that maybe a maximum of £300 can be withdrawn per calendar day, and if monthly statements to the customer are checked, then the fraud can perpetuate for at most one month. Thus each stolen PIN is worth up to £9300 to the attacker – maybe on average £5000. There have been cases where velocity checking was bypassed, or not even present [20]; in these circumstances, as a rough guide, one person can withdraw about £25000 per day working full time.

Unfortunately for the bank, because their prime business of storing money for people is strongly built upon trust, the effects of fraud on the bank's image and the trust of its customer base have to be factored in. These are very difficult for an outsider to assess, let alone consider quantitatively.

If handled shrewdly, the loss may not be costly at all, for instance if the bank declares the customer to be liable. If the amount is small enough, the customer will be tempted to give up rather than wasting time and effort to retrieve a small sum. However, the potential loss of revenue from bad handling of a fraud worth, say £10000, could be in the tens of millions in terms of long-term business lost.

The primary motive for attacking a bank is obviously financial gain. Some attackers seek *short-term financial gain*, for instance by extracting PINs for several hundred accounts and looting them in a couple of weeks. Others may plan for *mid-term financial gain*, for instance by selling a PIN extraction service for stolen cards at £50 per card; this slow trickle of cash may be easier for the attacker to conceal in his finances.

Higher-level goals maybe to achieve financial gain with the explicit desire that the bank knows it is being robbed: *extortion* or *blackmail*. On the other hand, the goal might not include financial gain, as with *revenge*, or attacker may simply desire to create *anarchy*.

4.1.2 Threat Model

The primary threats to the above targets of attack are as follows:

- *PIN observation*. Secretly observing someone enter their PIN, or 'shoulder-surfing' as it has been dubbed, is an easy way to discover a small number of PINs. It is unlikely PIN observation attacks will ever be completely eradicated.
- *PIN guessing*. PINs are weak secrets, so there is always the possibility that they can be guessed outright. Guessing can be made easier by exploiting weaknesses in the algorithms used to determine PINs, offsets on the magstripe [31], or information in the wallet of the victim (in the case of a stolen card).
- *HSM assisted PIN guessing*. Bank insiders may have the opportunity to exploit direct access to the HSM verifying PINs. Normally this does not present a large window for abuse, requiring thousands of transactions per PIN, plus a visit to an ATM machine up-front. However, new techniques such as the decimalisation table attack (see section 7.3.10) make this a much more realistic type of attack.
- *HSM key material compromise*. Compromise of key material in an HSM, through API attacks or through collusion can reveal the keys for calculating customer PINs, or keys under which correct encrypted PINs are stored. This type of attack is only available to a programmer with sufficient access.
- *HSM encrypted PIN block compromise*. Encrypted PIN blocks travel all around ATM networks, holding trial PINs from valid customers at ATMs and correct PINs on their way to PIN mailer printing sites. These blocks can be decoded through key material compromise higher up the hierarchy, or can be compromised themselves with new information leakage attacks and differential protocol analysis (see section 7.2.6).
- *Authorisation response forging*. Once a PIN is verified by an HSM, it is the responsibility of the host computer to pass back an authorisation response to the ATM, which will then dispense the cash. These responses are supposed to have their integrity assured by MACs on the messages on each link, but the messages are still vulnerable to modification within the hosts, during MAC translation, and sometimes on the links themselves, when network operators simply don't bother to enable the MACs on the messages.

- *Procedural control bypass.* Procedural controls can be subtly modified to allow key material change or discovery by one or a few of the authorised parties in collusion. They can only be completely bypassed by targeting weak links further up the chain. For instance, older banking HSMs use key switches for authorisation. If there is a reissue procedure for lost keys, a requisition form could be forged by the attacker, completely bypassing the rest of the procedures protecting the bank's legitimate copy.

- *Key material discovery.* Occasionally gross misunderstandings of the secrecy requirements on key components are made. There are reports of key components being stored in public correspondence files [3], instead of being destroyed after use. An attacker could exploit a failure such as this to discover a transport key, and unpick the rest of the system from there.

- *Brute force cryptographic attacks.* Older bank systems still using DES are nowadays vulnerable to brute force key material guessing attacks. Insiders with HSM access may be able to use techniques such as a meet-in-the-middle attack (see section 7.2.2) to bring the key search within range of a desktop PC. Outsiders may need to invest roughly £10000 to build equipment that can crack DES keys in a reasonable time period.

- *Falsely disputed transactions.* Every bank customer has a simple strategy available to defraud the bank – make some ATM withdrawals outside their normal pattern of usage, and claim they are phantoms. This strategy relies upon the legal precedent making customer reimbursement likely.

- *HSM vulnerability disclosure.* Discovering and disclosing vulnerabilities without ever being in a clear position to exploit them can support goals of extortion and blackmail.

4.2 Electronic Payment Schemes

The existing electronic payment schemes based on magstripe cards have used HSMs for some time to protect communications links between banks, and to hold keys which are used to verify the genuineness of a card presented to a Point of Sale (POS) machine (using the CVV values). These contain secure microcontrollers such as the Dallas DS5002. In large supermarkets and chain stores, HSMs may also be used as concentrators for networks of tills handling card and PIN information. HSMs are also an integral part of the back-end systems at banks which process these transactions, preventing operations centre employees from exploiting their positions.

The EMV standard is currently being rolled out, which aims to replace current magstripe technology with PIN on chip smartcards. The EMV standard is named after the three member organisations that created it: Europay, Mastercard and

VISA. It also permits end-to-end communication between the smartcard and the HSM at the issuing bank.

The next generation of electronic payment schemes are well on the way. Although home banking is already deployed, for it to become ubiquitous designers need to find better ways to establish an island of trust in part of the home user's PC – some are waiting for the general promises of "trusted computing" to materialise, and others are developing smartcards and reader solutions, or special disconnected tamper-resistant authorisation devices (similar to the RSA SecurID) to provide a Security API which they hope will be resistant to attack by malicious code running on the home user's PC.

Other electronic payment schemes are of the digital cash genre. Various of them rely upon trusted third parties to mint electronic tokens, and these third parties can make use of Security APIs to control the production and distribution processes of electronic cash.

4.3 Certification Authorities

Certification Authorities and Public Key Infrastructures (PKIs) were the crucial application that prompted development of the current generation of HSMs. HSMs are now found underlying corporate IT services, certification authorities and virtual private networks. In this new application area, establishing the threats to security is more difficult than for older applications such as ATM security and credit dispensing. The typical goals of an attacker such as personal profit remain largely unchanged, but the path to achieving this goal is more complex, and will be dependent upon the semantics of the certificates processed, and the PKI to which they belong. In particular, there may be no secret information obtainable or bogus information insertable that could be directly exchanged for money.

The HSMs used at Certification Authorities have developed some way on from simple acceleration units that sped up the modular math operations. They are now key management devices and their function is access control: to protect and control the use of the private keys in PKIs. Through the use of procedural controls HSMs can help enforce more sophisticated and stringent policies on the circumstances of key usage. They can enforce dual control policies on the most valuable keys in a CA, can help supervisors monitor the activities of large numbers of human operators efficiently, and keep signed audit trails of activities to allow retrospective monitoring of access control.

4.3.1 Public Key Infrastructures

Certification authorities are all about enabling identification and communication with people, and are closely tied into public key infrastructures that contain the

certificates they produce. It is easy for an HSM to provide protection for the CA's signing keys in the face of simple threats such as theft of computer equipment or hacking of the corporate network, but they also have potential to process policy components and genuinely assist in the operation of a Certification Authority – but to do this the HSM's Security API policy must take into account the threats, which are linked to the purpose and value of the certificates which the CA produces. Studying the underlying PKI that the certificates belong to can help build an understanding of the threat.

PKIs are dynamic collections of keys and certificates (documents signed by keys) which reflect a management structure or usage policy. PKIs are used to add structure and preserve the integrity of systems with large numbers of principals, where the structure is constantly changing. Examples include

- Logon authentication for large companies, maybe with 10000 or more employees and offices in different countries with different IT staff running computer systems in each country.
- Delegating and managing code signing ability to ensure that programmers contributing to a large website such as `www.microsoft.com` can only upload bona fide code.
- Transaction authorisation – establishing digital identities for people, and enforcing signature policies to validate transactions (e.g. a stock trade must be authorised by one trader and one manager).

Typical tasks performed in a certification authority are *registration & certification*, *certificate renewal*, and *certificate revocation*.

4.3.2 Threat Model

Most of the sensitive information within a PKI has no intrinsic value: certificates and keys are means to an end, worthless when divorced from their context. The one exception is the *root key* for an API, which could be considered to have intrinsic value developed by expending effort securely distributing it to millions of end-users. A corporate brand name is a good analogue for a PKI root key – it can be absolutely *mission-critical*. Adequate protection for the PKI root key is vital.

Lower down the hierarchy of keys and certificates, there may be no intrinsic value to key material, but revocation and reissue could still be expensive. It may be permissible to detect abuse rather than absolutely prevent it. The cost of extra protection must be weighed up against perceived threat, and cost of key material reissue. Late detection strategies are also favourable when the security is only to achieve "due diligence" – buying products which satisfy accreditation procedures and insurance requirements in order to shift liability.

There are three sorts of threat to key material in a PKI:

- *Theft of key material.* If intrinsically valuable key material is stolen, the attacker could put it to any possible use. This is the toughest attack to achieve, almost certainly requiring collusion or deceit, but would enable any of the goals above, and leave little evidence of the theft afterwards.
- *Admission of bogus key material.* Admission of bogus key material (i.e. key material which is known or chosen by the attacker) can achieve the same goals as theft of existing key material, but the attack is likely to take much longer, as newly admitted key material would have to slowly grow from having no intrinsic value. Admitting bogus key material might immediately enable theft of confidential data, but forged certificate sales and extortion would be harder. Residual evidence of attack is of course inherent in admitting bogus material, but auditing to spot bogus material is non-trivial.
- *Abuse of key material.* Finding ways to gain unauthorised access to existing key material could permit sale of forged certificates, and theft of confidential information, but blackmail would be very difficult. Access has to be maintained to the security processor for the abuse to continue. The attacker is thus at a significant disadvantage and risk. But if there is very valuable key that can be abused, a one-off event may still be worthwhile. Key abuse attacks will have an active component, and maybe involve deceit.

In summary, a Security API in a certification authority should carefully protect the root key, try to limit reliance on audit to spot abuse of mid-value key material, and in particular, resist the threat of systematic abuse of access control. Facilitating these goals is at least as important as preserving the secrecy of private keys themselves.

A range of types of employees will be in a position to attempt one of the above sorts of attack:

- Senior manager (controls employees and policy)

  Senior management are of course in a strong position. A senior manager can influence policy, and as is demonstrated later, only subtle changes are necessary to break procedural controls. He would only come under fire if his changes break fundamental and established practices or codes of conduct for that industry. Uninterrupted access to the HSM, and other resources including modules for experimentation would be within reach. However, experimenting with the API would be out of the question during office hours. Attacks involving collusion are likely to be between senior management and a lower level employee who has everyday access to the module.

- Single security officer (in an *m-of-n* scheme)

  As a team, the security officers are all-powerful. A single security officer is already part way to exercising this total power, which would straight away yield an attack. He or she is in a good position to deceive the other security officers, and is likely to be trained and experienced in operation of the module. However, he is under scrutiny from all levels – management, colleagues and users – as they are aware that deviation from established procedures gives him full privilege. He will have good host access, but is unlikely to be able to operate the host every day.

- Single operator (in an *m-of-n* scheme)

  Like the security officer, an operator holding an access card is already part way authorised to perform sensitive actions. He or she is in a good position to deceive other operators, and it is likely that the training and skill of the other operators will be lower than that of a security officer. A card-holding operator would regularly have access to the host in the normal line of business.

- Operator (access to host, no cards)

  An operator who does not have rights over any of the access tokens still has scope to perform attacks. He is in a good position to subvert the host or harvest passwords and PINs from other operators. As a colleague he would be in a position to deceive card-holding operators, or possibly even security officers.

4.4 Prepayment Electricity Meters

HSMs are an important and integral part of the prepayment electricity meter systems used to sell electric power to students in halls of residence, to the third-world poor, and to poor customers in rich countries [5]. They are typical of the many systems that once used coin-operated vending, but have now switched to tokens such as magnetic cards or smartcards. The principle of operation is simple: the meter will supply a certain quantity of energy on receipt of an encrypted instruction – a 'credit token', then interrupt the supply. These credit tokens are created in a token vending machine, which contains an HSM that knows the secret key in each local meter. The HSM is designed to limit the loss if a vending machine is stolen or misused; this enables the supplier to entrust vending machines to marginal economic players ranging from student unions to third-world village stores.

The HSM inside the vending machine thus needs to be tamper-resistant, and protect the meter keys and a value counter. The value counter enforces a credit limit; after that much electricity has been sold, the machine stops working until it is reloaded. This requires an encrypted message from a controller one step up higher the chain of

control – and would typically be issued by the distributor once they have been paid by the machine operator. If an attempt is made to tamper with the value counter, then the cryptographic keys should be erased so that it will no longer function at all. Without these controls, fraud would be much easier, and the theft of a vending machine might compel the distributor to re-key all the meters within its vend area. There are other security processors all the way up the value chain, and the one at the top – in the headquarters of the power company – may be controlling payments of billions of dollars a year.

4.5 SSL Security and Acceleration

Secure Sockets Layer (SSL) is an widespread protocol used in conjunction with HTTP to secure communications on the web. It supports sensitive web services such as secure payments and electronic banking. Public keys embedded in internet browsers are used to authenticate a chain of certificates that attest to a relationship between a particular domain name and a public key used in the SSL protocol. The user relies on this certificate chain to be sure that she is communicating directly with the webserver of the site in question – a merchant or electronic banking service, for example.

A webserver supporting SSL commonly performs a private key exponentiation for every connection attempted, so it has a considerable strain placed upon its processors by the modular arithmetic. This need to accelerate the public key operations spawned the application field of SSL accelerators. The companies providing the products also realised that the potential risks to a business of compromise of their SSL private key were considerable (undermining the trust of their customer base is very serious, even if the private key is not used for large scale spoofing or eavesdropping). A second function thus arose of the cryptographic coprocessor – to protect the private keys from theft, should the webserver be compromised. This defence is in some senses just security through obscurity – most people who hack webservers will probably not know how to hack SSL accelerators, but it is reasonable to believe that designers of dedicated crypto acceleration equipment will be better at getting the protection right than authors of webserver and O/S software, for whom the crypto is not a speciality.

4.6 Digital Rights Management

Digital Rights Management is the control of distribution and use of copyrighted works. Since the introduction of digital media, lossless copying has become possible, and publishers are seeking to protect their revenue streams. DRM can also be used for controlling access to media to enable alternative marketing strategies, such as a subscription service for access to media.

The most widely deployed DRM mechanisms are currently in consumer electronics. For example, Sony Memory Sticks – Flash based digital storage media – have 'protection bits', that mark certain files with 'do not copy', or 'only retrievable by authorised Sony devices' tags. DVDs also have DRM mechanisms for 'region control', a finer market segmentation method which allows geographically staggered release dates, and makes trafficking illegal bitwise copies slightly harder. None of this technology is highly tamper-resistant, though modern miniaturised circuitry does discourage straightforward physical attack.

There are more DRM mechanisms that begin to look like Security APIs. They are usually software based, and just like their hardware counterparts they rely upon obscurity and obfuscation. Third party copy-protecting encapsulation has been used for distribution of electronic books, and similar techniques have been tried for audio and video media. A common paradigm is to have support software entrench itself in the operating system, storing protected media in encrypted form, and hiding the keys. This O/S add-on then presents a Security API to some graphical user interface which the end-user will use to manage his or her media collection. These third-party packages are often general-purpose, so have properties much like general-purpose crypto APIs. They will support binding of media to a particular PC or system configuration, expiring licences, and process feature codes which change the access control policies, or unlock further content. Additional features and repeat subscriptions can thus be sold to users without the need for further downloads of large binaries.

Examples of these packages include:

- Folio Rights Publisher [46]
- Microsoft Rights Management System [47]
- Infraworks Intether Suite [48]

Some manufacturers have produced generic rights management components that add into Windows and control not just media for entertainment, but also electronic documents and email. These aim to enable control of information flow within organisations, to prevent leaks and make theft of intellectual property more difficult. In late 2003 Microsoft released its new Office suite of applications, with *Information Rights Management* (IRM) facilities integrated into all the applications. These use the Microsoft Rights Management architecture to enable restriction of document flow, expiry, and controls on editing.

DRM technology within tamper-resistant hardware is not currently widely deployed. IBM's Enhanced Media Management System (EMMS) has an optional secure hardware component [30], but most existing solutions are not specifically tamper-resistant. The crucial DRM mechanism of the future could be Microsoft's *Next Generation*

Secure Computing Base (NGSCB), previously called 'Palladium' – a whole set of technologies to be integrated into the next generation of PCs, including a tamper-resistant cryptographic coprocessor, and a special security kernel for Microsoft operating systems. NGSCB proponents hope that they will experience the same success controlling PC media as that of media read by consumer electronics. But the more complex APIs presented by DRM software on PCs will be hard to get right, whether or not they are supported by back-end tamper-resistant hardware. It will not be a surprise if the first generation of DRM hardware for PCs is only resistant to casual attack.

4.7 Military Applications

There is unfortunately little information in the public domain about military applications of Security APIs (and there are no guarantees that the military knowhow that has been published is not misleading, unintentionally or otherwise). *Nuclear command and control* appears to have employed hardware enforcing simple Security APIs that perform access control on arming of weapons. In particular, these APIs implemented dual control policies – the arming must be authorised by multiple top-ranking officials – that were permeated right through the command infrastructure down to the devices themselves. There would then be no weak-spot where a single person could seize control of an array of weapons. This application area certainly seems to have catalysed the development of tamper-resistance, which is described further in section 6.1. There is plenty of speculation about nuclear command and control on the internet [44].

Battlefield radios certainly use cryptography to secure communications, and there is indication that some military COMSEC equipment loads keys from a physical token – a "crypto ignition key". Such devices could be considered to have simple Security APIs that ensure that keys can only be loaded and never extracted. A simple policy such as this could however substantially mitigate the consequences of loss or capture of the equipment.

4.8 Specialist Applications

Bills of Lading – Securing high-value business transactions such as sale or transfer of goods in transit across the Atlantic, is a hard problem. When the goods cannot be handed over in synchrony with the sale, there are a number of potential frauds, such as selling the same goods twice. It may be months after a sale when an oil tanker's course finally becomes unique in differentiating between two probable destinations, and these months can be used to launder the profit from a double sale and disappear. Trading authorities have adopted a measure of computer security to uphold simple

policies on transactions (e.g. the same goods cannot be sold twice). A natural way to implement such a policy is with an HSM.

Key-Loading Sites – Applications which deploy thousands of HSMs into a public environment, such as pay-TV smartcards and prepayment electricity meters all require an infrastructure to initialise them before delivery. Generic manufacturing processes are often followed by a key-loading phase, which may happen at an entirely different facility from that where the manufacture took place. These key-loading facilities can hold master keys which are used to derive the keys for a particular device, or valuable signing keys used to certify the authenticity of a device. The same API technology which protects the devices during deployment is thus used to secure the loading process from potentially untrustworthy staff.

Chapter 5

The Security API Industry

This chapter introduces the major manufacturers of Hardware Security Modules, focussing on HSMs that have well-developed Security APIs. It is intended as a non-authoritative background reference to flesh out the reader's understanding of the companies that designed the major APIs discussed in this thesis. It also provides a starting point for those seeking more APIs and material for future analysis.

5.1 People and Organisations using Security APIs

- *HSM Manufacturers* are in the business of producing the hardware that runs code implementing a Security API. They may package their hardware in a tamper-resistant enclosure, and will have to write firmware for loading the Security API code. There are 10–20 such companies worldwide. Many of them nowadays try to market entire *solutions*, so have departments that consult on Security API usage as well the production and R&D departments. The business model can get quite complicated: for example, the IBM 4758 hardware and firmware was designed at the T.J. Watson Research & Development labs, is manufactured in Italy, and then sold at a subsidised rate (circa $2000-3000 per unit). The subsidy is then made back by licensing third-party code development kits and performing specialist consulting. Over half of the HSMs available on the market now support loading of third-party code – so a custom Security API can be implemented. Other manufacturers (e.g. Thales) do not advertise third-party customisation, but will provide it themselves as a service if a client has a specific request.

- *Product Integrators* provide integrated solutions for corporate end users. They design or customise both the Security API and the host software which interfaces to it. Their business model may include selling continued support, services and training for the integrated solution they have built. They may develop their own code on top of the base services provided by an HSM, or

customise existing transaction sets. Whilst there are only a handful of HSM manufacturers, there are a multitude of product integrators. The HSM manufacturers licence out development kits themselves, and may do some of their own product integration for larger clients.

- *End User Corporations* are those that buy an HSM solution for a standard application, or commission a product integrator to make a specialist solution. The employees of the company will likely interact with the API using some host software supplied with it in the case of PKI solutions, or written by an in-house programming team in the case of more specialist APIs. There may be only one or two people in the corporation with a technical understanding of the API, who will be responsible for maintaining the corporation's host software. The role of the in-house programmers is usually not to replace the key management facilities – but to supply a further abstracted and simplified API to other programmers. These simplified APIs are usually not security critical and repackage only common commands (e.g. encryption, decryption, signing). Alternatively the in-house team might make facilities available to employees via a graphical interface. Note that the scope of API repackaging may only cover the more basic commands; HSM manufacturers tend to keep master key handling procedures – initialisation, cloning of modules, and recovery from backup – quite distinct from those used by the normal applications which interact with the HSM.

- *Employees*

  Employees at corporations that use cryptographic equipment (excluding programmers themselves) may end up interacting with the Security API every day. For example, staff at a certification authority that produces certificates in correspondence with paper forms will be interacting with the Security API to make every signature, as well as for authentication of their identity. Some staff may even be aware of high-level rules enforced by the Security API, e.g. bank tellers will be aware that money cannot be created nor destroyed – just transferred between accounts. The interface presented to the staff members themselves can also be considered a Security API if it has particular visible policies controlling how the staff process sensitive information. Examples of policies at this level are the anonymisation of individuals in research databases, and account management functionality for bank tellers. This sort of high level API was once attacked when an unscrupulous teller discovered that address changes were not audited by the API. He first changed a customer's address to his own, reissued a card and PIN, then changed the address back again.

- *Individuals, Customers, Clients*

  Individuals (and small companies such as merchants) may interact with Security APIs embedded in equipment they use as part of their interaction with other businesses of which they are customers. For an individual this may come in the form of a smartcard for electronic banking; for a merchant it might be the capture device for credit cards. They will likely be shielded from most of the interaction with the API, but will probably be aware of some of the overriding principles from the API's policy which affect how they must interact with it. For example the user of the smartcard may know that a PIN has to be presented to unlock access for certain commands, but they wouldn't know the specific function of these commands.

- *Government Agencies* such as NIST (National Institute of Standards and Technology) have a role producing cryptographic standards. NIST produces the FIPS (Federal Information Processing Standard) series and involved with the internationally recognised suite of 'Common Criteria' standards. The more clandestine government agencies once kept themselves to themselves, but now the NSA in particular has involvement in the commercial security community, and maintains an interest in standards and evaluation. The NSA recently released code for the collaborative Secure Linux project [54], which has a special *security module* (a module of code) which is privileged and enforces a set of policies on how the applications can access the rest of the operating system.

- *Validation Labs* are a newer arrival to the Security API world – increasing demand for conformance to government quality specifications from commercial industry has led to the establishment of these private organisations, which check commercial products against government standards. Previously, when the military was the sole customer for cryptography, validation was performed by the military, for the military. The availability of third-party evaluation labs has led to an interesting shift in the way validation is done. Companies now stipulate that they will only consider validated products, but the manufacturer actually commissions the evaluation of the product. The validation labs then make bids for evaluation, and (presumably) the contract is won by the lowest bidder.

5.2 Corporate Timeline

5.2.1 1970 to 1990

Year	Events
1972	NBS (precursor to NIST) requests standardised crypto algorithm
1974	IBM submits Lucifer algorithm to NBS
1974	IBM launches 3600 series finance system – ATM, Teller, PIN software
1976	DES released by NBS
1976	Diffie and Hellman invent public key crypto
1977	**IBM 3845 DES encryption unit launched**
1977	IBM extends its 3614 ATMs to use PIN derivation
1978	RSA invented
1979	IBM 3848 DES encryption unit released, which is faster than the 3845 and supports 3DES
1979	IBM product development site in Charlotte opened
1979	Mykotronx founded in California, working for the NSA
1979	**Eracom founded in Australia**, producing IBM channel connected HSMs
1981	IBM 4700 series finance system – ATM, teller, PIN software
1984	Rainbow Technologies founded in California, 'Sentinel' anti-piracy product highly profitable
1984	Eracom expands into Germany
1985	**VISA Security Module introduced**
1985	Eracom produces PC crypto card
1987	**IBM μABYSS mesh introduced**
1987	Rainbow floats
1989	**IBM Transaction Security System (TSS) products introduced, including CCA, 4755 high-speed crypto card, 4573 channel-attached crypto unit**
1989	VSM clones appearing on market

5.2.2 1990 to 2000

Year	Events
1991	IBM S/390 ICRF (Integrated CRyptographic Facility) launched
1991	Mykotronx gets US funding for Clipper Chip development
1991	Eracom produces first "secure application modules"
1992	IBM TSS adds public key crypto support
1993	Jan IBM 4753 Enhanced: add key distribution management, & session level encryption.
1994	**Chrysalis-ITS founded**
1995	**NIST FIPS 140-1 validation programme started**
1995	Rainbow acquires Mykotronx
1995	Chrysalis ships first products
1996 May	Racal SPS adds smartcard LMK loading, public key crypto
1996	**nCipher founded**
1996	**IBM launches 4758-001** (as replacement for 4755)
1997 May	Racal SPS introduces RG7110 high-speed ethernet HSM
1997 Jun	IBM 4758 announced
1997 Oct	Chrysalis Luna-1 token validated FIPS 140-1 Level 2
1997	**Rainbow enters HSM market**
1997	S/390 CCF single-chip CMOS crypto unit replaces ICRF, and adds public key crypto
1998 Sep	nCipher nFast validated FIPS Level 3
1998 Oct	Chrysalis Luna CA3 validated FIPS Level 3
1998 Nov	**IBM 4758-001 first module validated FIPS Level 4**
1999 Jan	IBM S/390 CMOS Cryptographic Coprocessor validated Level 4
1999 Jan	Racal SPS introduces "Websentry" HSM
1999 Nov	IBM 4758-013 validated FIPS Level 3
1999	IBM 4758-002 launched

5.2.3 2000 to Present

Year	Events
2000	**Racal taken over by Thomson-CSF,**
	HSM manufacturing offshoot was 'Zaxus'
2000	Rainbow launches CryptoSwift eCommerce accelerator
2000 Jan	nCipher nShield validated FIPS 140-1 Level 3
2000 Jan	nCipher Keysafe launched
2000 Jun	IBM 4753 & 4755 discontinued
2000 Sep	IBM 4758-002 Validated FIPS Level 4
	IBM 4758-023 Validated FIPS Level 3
2001	Eracom opens US sales offices, merges with
	Theissen Security Systems AG and others, keeps name
2001 Apr	**Zaxus becomes Thales eSecurity**
2001 Apr	**nCipher launches SEE (Secure Execution Engine)**
2001 Jul	Eracom CSA8000 validated FIPS Level 3
2001 Aug	Rainbow CryptoSwift HSM FIPS Level 3
2001	IBM zSeries servers release 'PCICC' custom
	crypto platform, including a 4758
2002	nCipher floats on London Stock Exchange
2002 Mar	nCipher CodeSafe launched
2002 Sep	**nCipher payShield (financial HSM) launched**
2003	Rainbow acquires Chrysalis-ITS
2003 Jul	Atalla Cryptographic Engine FIPS Level 3
2003 Oct	**nCipher netHSM launched**
2003	IBM launches PCIXCC crypto card designed to replace
	all previous crypto cards

5.3 Summary of HSM Manufacturers

5.3.1 IBM

`http://www-3.ibm.com/security/cryptocards`

IBM have been at the core of commercial cryptography from the start, and are the largest company with an HSM department. In 1974 they introduced the first ATM transaction processing systems for banks, and in 1977 developed the very first HSM supporting DES encryption (the IBM 3845) that formed part of the transaction processing system. This was followed by the IBM 3848, which supported asymmetric key usage. Their present day 47 series HSMs are well known, but do not dominate any market in particular. Of this series, the most recent is the IBM 4758. It started life as a research project at T.J. Watson Labs, NY, to provide a general purpose secure processing platform, and to supersede the ageing 4755 HSM. IBM went on to achieve the first FIPS 140-1 Level 4 security evaluation for this device in 1998 – the highest commercial evaluation available at the time. The sophisticated code-loading and authentication architecture, state-of-the-art tamper-resistance, and efforts to formally validate parts of the firmware put IBM's 4758 in a league of its own. In the closing few months of 2003 IBM has released word of the successor to the 4758, which is called the 'PCIXCC'. Unfortunately there is very little information about its architecture and design at this time.

The most frequent application to be run on the 4758 is IBM's "Common Cryptographic Architecture" – a framework for financial cryptography which was introduced as part of their Transaction Security Services (TSS) suite in 1989. To many programmers, this product constitutes the Security API of the 4758 and there is little need to interact with the 4758 firmware after initial loading (the CCA team themselves dispensed with the outbound authentication functionality of the 4758's firmware API). The 4758 CCA is also sold as part of the 4753 – an HSM designed to interact with mainframes, consisting of a stripped down PC with a 4758 and IBM channel adaptor card.

5.3.2 Thales / Zaxus / Racal

`http://www.thales-esecurity.com`

Also known as:
Racal Secure Payment Systems (Racal SPS)
Zaxus
Thales eSecurity

Thales is the current owner of the Racal RG7000 series hardware security module – probably the most widely deployed device in the banking community, which dominates European markets. Their product brochures claim that 70% of the world's

ATM transactions travel through RG7000 series devices. Racal SPS originally had the contract to produce the VISA Security Module, but once VISA disassociated their brand name with the device, it developed into the RG series HSMs. In 2000, the HSM department split off to fend for itself, under the name 'Zaxus', but was bought after a year or so by the multi-national conglomerate 'Thales'. The original Racal product lines have not been radically updated since their conception in the 80s – smartcards and keyswitches have augmented passwords for access control to the HSMs, but the key management architecture in the RG7000 remains very dated. The RG series devices are also not currently strongly tamper-resistant – their main line of defence is a lid-switch whilst others such as IBM have had wire meshes and even tamper-resistant membranes. In the RG series API lies a somewhat paradoxical reputation: it must be respected for its continued dominance of the financial security market, even though their technical product is way behind the other vendors.

Thales also produce less famous, but more advanced modules for electronic payments and PKI security, for example, their 'Websentry' product line, which uses the PKCS#11 API and its own custom key management interface.

5.3.3 nCipher

`http://www.ncipher.com`

nCipher is one of the youngest HSM manufacturers out there, founded in 1996. Their API is uniquely modern in that it is centred around public key cryptography. Their first product was a hardware cryptography accelerator card – the 'nFast' – which was designed to be fitted to web servers to increase their SSL capabilities, as well as protect the private SSL keys, should the webserver be compromised. Their focus is on performance and API security, not tamper-resistance. nCipher argues that sophisticated tamper-resistance is overkill given the physical access controls on the server rooms where these devices typically reside. Current nCipher devices are available in potted form, and do have a tamper-responding component, but they only claim tamper-evidence for their devices. Most of their products have achieved FIPS 140-1 level 3 evaluation. As the success of their SSL acceleration cards grew, nCipher released products including more and more key management facilities, and redesigned their API to reflect this. They introduced the 'nForce' and 'nShield' devices – available in PCI and SCSI forms – which competed with products from Baltimore and Chyrsalis to provide back-end protection for key material in Certification Authorities.

Since 2001, nCipher have incorporated the Secure Execution Engine (SEE) technology into their devices. This facility allows third-party code to run within their tamper-evident boundary, thus their HSMs are suitable devices for implementing completely custom Security APIs. In early 2002 nCipher floated on the London Stock Exchange; they compete with manufacturers such as IBM in the general pur-

pose crypto platform market, and since late 2002 have entered into the (already saturated) financial security market alongside Thales and the others.

nCipher's enduring informal trade mark is a blue LED on their modules which flashes error codes in morse!

5.3.4 HP Atalla

`http://www.atalla.com`

Hewlett-Packard now owns the US based HSM manufacturer 'Atalla', who retain a significant share of the US financial security market. They are the US equivalent of Racal SPS – one of the manufacturers that has been there from the beginning. Their financial security product range consists of *Network Security Processors* (NSPs), but they have other product lines, including custom programmable modules. Atalla has since 2001 been developing and marketing new key block formats which go some way to addressing the key binding flaws first publicly identified the same year, in the author's paper "Attacks on Cryptoprocessor Transaction Sets". Atalla runs a series of web seminars linked from their website, which are a useful (if sales-oriented) source of information about the financial security climate. Atalla's new proprietary key storage data formats are also being influential in the development of a new ANSI X.98 standard for PIN processing.

5.3.5 Chrysalis-ITS

Chrysalis-ITS	`http://www.chrysalis-its.com`
Rainbow	`http://www.rainbow.com`
SafeNet	`http://safenet-inc.com`

Chrysalis is a younger HSM vendor based in Toronto, Canada. They have been floundering since the bursting of the dot com bubble: they were bought in 2002 by Rainbow Technologies, which is now in the process of merging with SafeNet Inc. During their early years, they were very closely involved with the design of the PKCS#11, which is in fact modelled on their underlying 'Luna API'.

They produce small PCMCIA form factor cryptographic tokens which use the PKCS#11 interface for access to the keys. There are a range of tokens available, with essentially the same potential functionality, but customised for particular end user applications; their main business is supporting certification authorities. Their small form factor 'Luna' tokens have quite low performance, so they also market separate accelerator devices which temporarily take custody of the key material from the tokens.

5.3.6 Prism Payment Technologies

`http://www.prism.co.za`

Prism Payment Technologies is a South African HSM manufacturer that has swallowed up a large portion of the SA crypto market. They have two major roots. The first was the company 'Linkdata', which started out building electronic funds transfer switches for routing PIN processing transactions. After a management buyout, the name 'Prism' was created. They moved from there to focus substantially on the prepayment electricity market, where they developed the TSM200 (see section 6.3.6 and section 7.3.11 for a fuller description including an attack on the API). They were very successful in the SA prepayment electricity meter market, and also made POS devices using the TSM200.

In late 2000 Prism acquired 'Nanoteq' from Comparex holdings. Nanoteq was another SA HSM manufacturer that started out as a local builder and supplier of communications security equipment for the SA military. South Africa was able to support a number of different manufacturers during apartheid because of import and export restrictions, and the concern that imported devices may have trapdoors. Nanoteq was founded in the 80s, and later expanded into the commercial market with POS devices, smartcards and a low-performance HSM called the SM500. In 1998 Comparex acquired Nanoteq, a small HSM manufacturer called CAT, and part of Mosaic Software. Their product ranges were merged, and Mosaic software wrote a financial API for use with CAT's HSMs. It was this product range that Prism acquired, rebranding it the 'Incognito' series.

The Incognito series includes the older TSM credit dispensing HSMs, and several flavours of custom-programmable general purpose HSMs, which support authenticated code loading, and performance similar to the IBM 4758.

5.3.7 Eracom

`http://www.eracom-tech.com`

Eracom is an Australian HSM manufacturer, founded in 1979. It has produced general purpose cryptographic equipment such as a DES accelerator for block-level disk encryption in 1985, and since 1991 has been producing programmable HSMs. They have a strong secondary base of operations in Germany, and supply to various continental European nations from there. Their HSM ranges are called 'ProtectHost' and 'ProtectServer', implementing Eracom's proprietary 'ProtectToolkit' API.

5.3.8 Baltimore

`http://www.baltimore.com`

Baltimore was one of the first PKI vendors. They augmented their CA software by acquiring an HSM manufacturer 'Zergo' (previously 'Data Innovations'), and produced HSMs to integrate into their CA solution. Their major product line was the 'Sureware Keyper', which used PKCS#11 for access to key material, and had an integrated display and keypad. Baltimore collapsed when the dot com bubble burst, and their various acquisitions have been sold off.

5.3.9 Jones-Futurex

`http://www.futurex.com`

Jones-Futurex is a lesser known financial security HSM provider. Their modules are not really *coprocessors* (as most HSMs are) but in fact entire PCs enclosed within steel boxes. The host computer will communicate via a serial interface or via a network connection, and key management tasks are performed via integrated keyboard and monitor. They sell the majority of their units in the USA and Asia.

5.3.10 Other Security API Vendors

Microsoft

`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/security/security/cryptography_portal.asp`

Microsoft produce a software-only cryptographic API called 'MSCAPI' which is integrated into Windows. It is a general-purpose API which provides crypto services to applications and O/S components. Third-party developers can write Cryptographic Service Providers (CSPs) which augment or replace the functionality of the default CSPs.

RSA

`http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html`

PKCS#11 is RSA's generic interface standard for providing cryptographic services to applications. It was developed by RSA in cooperation with various industry partners, but the design borrows most from Chrysalis ITS's 'Luna API'.

PKCS#11 has been widely adopted since its release in c. 1996, and is very successful in the sheer number of applications that use it as an interface to gain access to their cryptographic services. It has a rather worse reputation however with HSM

manufacturers, because the standard leaves important policy issues unsettled, and has coarse-grained and flawed key management facilities. Its usability has also been called into question. Despite these worries, most manufacturers produce a device which conforms in spirit to the PKCS#11 API, though there may be crucial semantic differences that result in error messages if dangerous actions are attempted that the specification appears to allow. Manufacturers implementing PKCS#11 also tend to put the more complex aspects of key management in an orthogonal interface, and only use the PKCS#11 API to provide simple encryption, decryption, signing and verification services to the application.

The paper "On the Security of PKCS#11" [14], presented at CHES 2003, catalogues some of the many weaknesses of the standard. It constitutes one of the first public airings of these weaknesses, which have been muttered about by manufacturers, but gone unaddressed for some time.

Cryptlib

`http://www.cs.auckland.ac.nz/~pgut001/cryptlib`

Cryptlib is an API created by Peter Guttman, who was tired of the low quality and poor usability of software only implementations of Security APIs, such as PKCS#11 and MSCAPI, and decided to conceive his own design. It claims to have more elegant 'enveloping' and session management abstractions that make it powerful and easy to use. Cryptlib is freely available on the internet and is open source.

5.3.11 Odds and Ends

The additional manufacturers listed here are less significant *in the context of the discussion in this thesis.*

- "BBN" produce a range of HSMs and crypto accelerators, in particular their SignAssure HSM, which is FIPS 140-1 Level 3 validated. Their website is `http://www.bbn.com`.
- The "Trusted Computing Group" is an alliance of large software, hardware and embedded systems vendors, whose goal is to develop open standards for increasing the security of all manner of computing devices. It has grown out of the "Trusted Computing Platform Alliance", and contains the same big five founding members: Compaq, HP, IBM, Intel and Microsoft, but now also Sony, AMD and Sun Microsystems. Broadly speaking, the aim is to develop better operating system security through hardware support (including native crypto support) and architectural redesign. Microsoft is also busy making its own trusted computing architecture, which may or may not turn out to be supportable by the hardware developing in the TCG initiative.

The Trusted Computing Group will likely turn out to be a highly influential organisation in the next five years, and produce some important standards, but while these standards are in very early draft at the moment, their impact is minimal. Their websites are `http://www.trustedcomputinggroup.org` and `http://www.trustedcomputing.org`.

- "Wave" is a manufacturer of lightweight embedded HSM systems, and vendor of PC application software which utilises the Trusted Platform Modules (TPMs) that have already been integrated into some PC motherboards as part of the TCPA initiative. They use these facilities to offer secure document storage, signing, and try to deal with trust management issues too. Their website is `http://www.wave.com`.

- "Prime Factors" make the Band Card Security System (BCSS) which is a suite of software for providing back-end functionality for issuing ATM cards to customers, and includes software only implementation of the functionality provided by a typical Visa Security Module clone. Their website is `http://www.primefactors.com`.

- "Trusted Security Solutions" (TSS) produces the A98 product range – consisting of front and back end software for more securely establishing initial keys at ATMs and Point-Of-Sale (POS) devices. One of their products is particularly interesting: it is an automated interactive voice telephone answering service to allow ATM service engineers to report servicing and request and acknowledge the flows of printed key material. It is unusual in that as well as having a Security API to the host and the HSM itself, the interactive voice service is essentially a sophisticated trusted path system. Their website is `http://trustedsecurity.com`.

5.4 Interacting with Vendors

5.4.1 Buying from Vendors

In the PIN processing market, most HSM vendors sell directly to the banks, who are in effect the end users. This because financial HSMs have a standard PIN processing security policy hardwired which needs little if any modification to be appropriate for any bank. Thales and Atalla provide a consultancy service to provide modifications to the API specific to the requirements of certain banks doing unusual things. IBM provides consultancy services, but also has created its UDX (User Defined eXtension) toolkit [50] which allows their financial API, the CCA, to be extended with special transactions to meet specific customer needs. They also substantially subsidise the price of their 4758s, presumably making the money back in consultancy fees.

A bank will have a reasonable choice of vendors, so long as they do not have old equipment that they want to remain compatible with. The PIN processing API commands themselves vary little across manufacturers as they are so closely tied in with the ISO PIN block standards, but the key management architectures have varying degrees of complexity and granularity of access control. There are also a wide range of authentication mechanisms to choose from. There is a mounting catalogue of vulnerabilities in the current PIN processing architecture, few of which have been addressed by anyone, so an absolute comparison of security has little value. The default approach of comparing manufacturers on price and customer service seems quite sensible. One pitfall is that performance figures are extremely warped. Almost all of the vendors artificially limit their PIN processing transaction rates, so that they can sell more HSMs. In the real world processing power continues to double every 18 months, in accordance with Moore's law; however, PIN processing power has been put across as only linearly increasing, if that.

The big issues when buying PIN processing equipment may end up having little to do with security, speed or price, but instead focus on legacy compatibility, both with older PIN processing devices and with old mainframe software which still speaks those protocols, and is very expensive to maintain because of the shortage of programmers skilled with these older languages. Thales inherited the particularly powerful legacy API associated with the Racal RG7000 series HSMs, and their continued success as the preferred choice for banks over immeasurably technically superior and cheaper devices, gives pause for thought as to the importance of compatibility and the power of brand names.

In the PKI market, the HSM choice is wider than that for PIN processing, but the API choice tends to be more limited. End-users will typically be running heavyweight certification authority software that only talks to the HSM for providing the actual signing operations, and taking final authorisation. Manufacturers such as Baltimore provided both HSMs and CA software during the dot com boom; since then the hardware and software suppliers have separated. The common API that sits between the CA software and the HSM is PKCS#11. All the manufacturers have versions of their HSMs which support the PKCS#11 API, but this API is more of a framework than a concrete specification, and needs to be coupled with a "conformance profile" document. The level of detail and availability of conformance profile documents varies a lot. The key issues for the buyer to consider here are performance and authorisation method. Public key operations are quite expensive, especially generation, so supporting peak workload could be an issue. Chrysalis-ITS's Luna tokens are particularly poor in performance, but their PCMCIA form factor can be considered advantageous in that they can be locked away in a safe (the flip-side is that they can be more easily stolen or substituted for fakes).

The SSL and crypto acceleration market is probably the one most amenable to direct comparison. Competition is based primarily around performance, but also ease of integration. nCipher and Rainbow are major players. Performance figures will be

available, but should be carefully analysed, as tricks to boost apparent performance figures are common, for example quoting a high rate for exponentiation, which turns out to be based around the common small exponent case.

The two big players in the custom platform market are IBM and nCipher. Both have good performance native crypto, and can be programmed in C. nCipher's secure execution engine is advantageous in that functionality can be built on top of the nCore API, which makes building custom APIs much quicker, and keys and data from the custom API can be manipulated using conventional commands issued to the HSM as well as by talking to the custom command dispatcher. The 4758's advantage is its multi-layered approach to code loading (O/S and application can be swapped in and out independently), and its outbound authentication facilities, which allow you to remotely establish a secure session with the software running inside the device. Many of the other smaller vendors do have custom programmable devices, including Atalla, and also Prism whose programmable HSMs may predate the 4758 in having such sophisticated functionality.

5.4.2 Reporting Faults to Vendors

From time to time users of Security APIs discover weaknesses (often while debugging a failing application, or designing an interoperable device), and there is now continuing academic research at the University of Cambridge specifically looking for weaknesses in current APIs. Most vendors give a positive reception to information fed to them about security flaws in their products, but they tend to be much more cautious about providing feedback, and dealing with potential publicity.

If the aim of reporting the fault is to get the fault fixed, it is very important to have the right points of contact within the vendor's organisation. Dual entry points are ideal – someone with the technical understanding to appreciate the attack, and someone senior enough to be able to instigate changes (and maybe even make a comment outside the company). When the fault that has significant real-world consequences goes unaddressed by an unresponsive vendor, another option is to encourage the media to champion the cause.

Media that speaks to a vendor's client base is extremely influential, and can pressure a vendor into making quite radical changes of policy. However, once the media is involved, the battle is fought on an entirely higher (some would argue 'inferior') plane – that of public relations. A key factor in a good news story is that it has two sides, so an effective tactic for the vendor is to dead-wall journalists and researchers by refusing to make any comment about vulnerabilities. If the story does go ahead, the implication is that the discoverers of the flaw are so hopelessly uninformed about the real-life implementation issues that their theories are irrelevant. However, a well-informed and technologically savvy journalist can usually cut through this and elicit a response. The drawback of pressuring vendors through the media is that

it is crude and unpredictable, and may end up doing more harm than good. In particular, developing an adversarial relationship between finders of exploits and the security architects trying to build them inhibits communication and does not necessarily result in fixed systems which are better.

Many industries are not faced with high-visibility resourceful enemies however, and the burst of attention surrounding media coverage can sometimes be considered beneficial in shaking things up in these sorts of slow-moving industries. For example, the PKCS#11 standard when read literally is really hopelessly insecure, and vendor's implementations now have much much more custom semantics to stop bad things from happening. Here, the users of the standard seem to be well aware of the shortcomings, but even with the right lines for feedback, gaining enough momentum to push changes through is difficult. The publication of Clulow's paper "On the Security of PKCS#11" [14] might catalyse some significant revisions in the next version of the standard. Another example is the author's own experience publicising a particular attack on the IBM 4758 CCA (see [51] for a summary of the publicity, and section 7.3.7 for details of the attack itself). This publicity prompted the early release of a new version of the CCA, version 2.41.

Chapter 6

Hardware Security Modules

6.1 A Brief History of HSMs

Hardware Security Modules (HSMs) have their roots in a military environment, just as Security APIs do. It must be noted that the information in the public domain about military usage of tamper-resistance is limited, and what information there is may be misleading (whether unintentionally or through malice). This section attempts only to give a flavour of the origins of tamper-resistant HSMs, which adds some context to the devices we see implementing today's Security APIs.

Early military HSMs were concerned with nuclear command and control, and communications security. During the cold war, the nuclear arsenals of the two superpowers grew and grew, and were deployed in allied countries as well as on home soil, and along frontiers. In this new environment, an increased danger of ordinance falling into the wrong hands was perceived. Following the Cuban missile crisis, President Kennedy ordered that all US nuclear ordinance be got under "positive control". *Permissive Action Links*, also known as *Prescribed Action Links* (PALs), were introduced to try to achieve this. The function of a PAL was to prevent the device from detonating without a valid authorisation code being entered, and to this end they were embedded deep within the warhead.

To prevent being bypassed, the PALs needed tamper sensors and tamper reactions. HSMs usually take tamper-reaction for granted as information is relatively easy to destroy, but for nuclear devices careful thought had to go into designing tamper-reactive mechanisms that would render the bomb useless should the tamper sensors be triggered. A range of tamper-reactions have been speculated (follow up [44] for more info), for example

- gas injections into the core to turn the plutonium or uranium into a hydride not dense enough to achieve critical mass.
- preventing uniform implosion of the core by varying the yield of the conventional detonating explosives. During correct detonation, the yield variations

are compensated for by staggered timing of detonation. The timing information is in effect a secret key, and is erased on tamper.

- soft detonation (i.e. non-nuclear detonation).

Figure 6.1: A control device for an early PAL

So in the field of military security, tamper-resistance developed to a high standard, although policies that their HSMs enforced never became particularly sophisticated.

It is difficult to tell whether knowledge was really carried over from military applications, or whether the same technology was reinvented. Some lines of commercial HSMs may have had the ability to build on aspects of military knowhow, for instance Rainbow acquired the NSA defence contractor Mykotronx, and some of the early South African HSM manufacturers supplied both military and commercial clients. However, the initial needs for physical security in the commercial environment were not perceived as very great, so most early tamper-resistance measures were ad hoc. The Visa Security Module initially had simply a lid microswitch that would cut the power to the RAM if the lid was removed. This served as an effective deterrent to an inquisitive employee, as if triggered the master keys would be erased and require reloading to bring the module back online – an event that would certainly be audited. There was still a threat though: the HSM service engineer whose job was to change the battery every six months was not a bank employee. He would be able to legitimately open the device, then locate and disable the lid-switch on his first visit. The next time he came to change the battery he could read out the RAM contents trivially!

Tamper-resistance was gradually adopted to counter the service engineer threat, and to provide greater protection in the event of physical theft of the device –

an increasing concern as the devices became smaller and more portable. IBM's 4755 (a predecessor to the well-known 4758) used their "μABYSS" design – a fine mesh of wires surrounding the module, which would trigger erasure of key material if interfered with. This was probably the first commercially available tamper-detection system not based around the lid-switch and steel enclosure paradigm.

HSM tamper-resistance continued to develop reactively as new physical attack techniques were identified, although the counter-measures were often well developed by the time knowledge of the threat reached the public domain. Temperature sensors to combat data remanence attacks, and radiation sensors to combat data burning attacks are both examples of counter-measures specifically designed to counter newly discovered attack techniques. On chip tamper-resistance was also being developed for smartcards at the same time.

In parallel to the increasing sophistication of tamper-resistance, the power and speed of security modules increased as well. Due to the hardware-optimised design of DES and the limited processing power available in mainframes, a common early perspective was that the dedicated cryptographic co-processor was simply responsible for speeding up the crypto. IBM's earliest hardware security modules – the 3845 and 3848 – were simply known as "DES encryption units". An incentive to increase these products in power gradually formed as it became more reasonable to use encryption for security, and the sizes of data files encrypted went up. The idea of encryption of files at a device driver level was also introduced; by 1985 Eracom was marketing a "PC encryptor" ISA card, which allowed block by block encryption of floppy disks.

The catalyst for the next generation of hardware security modules was added in the late nineties during the dot com boom. Public key cryptography was widely hailed as the solution to the key exchange problems of the past, and dozens of companies entered the PKI solutions market. However, big-integer modular arithmetic was a problem poorly suited to traditional CPU architectures, and it dramatically increased the load on general purpose processors in servers and desktops, that had easily taken onboard symmetric cryptography. The next generation of HSMs were thus accelerator cards for public key crypto, built first around embedded processors, and eventually including dedicated hardware acceleration for modular arithmetic. 1997 saw the release of the nCipher nForce and the IBM 4758: both had hardware acceleration for public key crypto.

The buzz-phrase for the very latest generation of HSM is "network attached". Physical security can more and more be piggy-backed on the inevitable physical security of server rooms containing valuable equipment, and if the crypto resources can be located there they need not be so tamper-resistant. This in turn allows more heat dissipation and higher performance. Chrysalis and nCipher both are now pushing their network attached HSMs which essentially perform exactly the same function as the old devices, but centralise the physical resources. The future of HSM design looks set to be a battle between online centralised resources such as these, and integration of crypto functionality onto normal PC motherboards (and eventually into

processors themselves).

6.2 Physical Tamper-resistance

An HSM's physical tamper-resistance can be a combination of both active and passive measures. *Active* tamper-resistance measures detect an attack taking place and foil it by destroying the target of the attack (usually cryptographic keys). *Passive* measures seek to make it harder to inspect and manipulate the active components of the tamper-resistance, or simply make it harder to reach the key material.

Passive Measures

- *Steel casing* remains a popular and easy to understand tamper-resistant measure. The idea of an HSM as a PC in a safe is appealing to many in the financial community. If an HSM is only as secure as the safe that its administrator keys are locked in, why not lock the whole thing in a safe? Whilst steel casing may seem absurd compared with the high-tech state of the art, it is particularly effective when combined with reasonable on-site physical security – it is difficult to sneak an angle grinder into a server room. The sheer weight of the casing also serves as an anti-theft mechanism, and some HSMs built according to this philosophy include extra weight deliberately.

Figure 6.2: A prototype PCI card potted in epoxy between two glass sheets

- *Potting* is nowadays one of the most common tamper-resistance measures, as well as a quite effective tamper-evidence layer. Choice of potting compound will affect thermal dissipation (designing high-performance tamper-resistant coprocessors is a nightmare), ease of manufacture, and of course ease of removal. Figures 6.2 and 6.3 show examples of potted HSMs. nCipher's PCI HSMs (section 6.3.3) also have the potting clearly visible.

Figure 6.3: An IBM 4758-001 part potted in urethane, showing membrane and interior (courtesy F. Stajano)

Active Measures

- *Wire meshes* were one of the first significant tamper-responding improvements to replace lid switches. There are almost always coupled with a potting compound which makes it difficult to move the wires, or apply patch leads. IBM's μABYSS appears to be the first commercial mesh design, which was used for the 4755. The tamper response can be conductive – if wires are shorted or broken the sensor is triggered – or inductive, where the sensor is triggered if the inductance of one of the loops changes. Conductive meshes respond rapidly to tampering but can be defeated by patching, whereas inductive meshes are more difficult to patch, but are slower to respond. In practice, inductive meshes tend to be harder for an attacker to defeat, as the slowing in response time is only of use for very fast attacks such as those using shaped charges.

- *Conductive Membranes* are the modern successor to wire meshes. The most well-known producer of membranes for the HSM industry is Gore Industries. Gore Industries 3D membrane is made of a special type of polyurethane which

is chemically very similar to their recommended potting compound, to make it difficult to selectively dissolve away the potting without destroying the membrane. It has multiple zig-zagging lines which are layered on top of each other, made by selectively doping the membrane. A sample of the unpotted membrane can be seen in figure 6.4. This technology is the commercial state of the art.

Figure 6.4: The Gore Industries membrane

- *Thermal Sensors* protect against attacks involving cooling. Severe cooling causes data remanence effects in RAM [37].
- *X-Ray Sensors* aim to detect radiation and erase key material before it can be 'burned' into the RAM by the intense radiation.
- *Power Glitch Sensors* detect fault induction attacks. They are important for HSMs with constrained form factors, which cannot hold smoothing capacitors internally, or must rely on an external clock, as with most smartcards. Detection of glitches could trigger full key material erasure, or a more measured response such as performing a full reset of the processor, to prevent any subsequent changes to the data or execution path of the code being acted upon.
- *PRNG Monitoring* is also a consideration where hardware sources of randomness are employed. If the hardware effect can be altered by bathing it in radiation, or changing the temperature, then the adverse affect on the entropy of the source must be detected, otherwise deliberately weak keys can be introduced into the system. If the API lets you generate keys with special privileges to export others, this could lead to a serious compromise.

- *Light Sensors* can be coupled with sealed housings to make disassembly more difficult. A simple light sensor can be trivially defeated by working for example in infrared, until the sensor has been identified and covered. More sophisticated designs involving wide band sensors or feedback loops transmitting and receiving are considerably more difficult to bypass.

- *Trembler Switches* have been historically included in metal box devices as an anti-theft measure more than a tamper-resistance measure. They are impractical in devices other than those that stay put in server rooms, and under those conditions, the physical security is likely to prevent the theft easily anyway.

- *Lid Switches* were the first hint of tamper-resistance applied to commercial modules such as the VSM. They are of course trivial to defeat if their locations are known. However, the unknown locations (or even the uncertain existence) of lid switches can provide a valuable deterrent to the otherwise opportunistic employee, especially when there is only a single device, and rendering this device inoperative would be immediately detected and investigated. One of two lid switches on the Luna XL cryptographic accelerator is shown in figure 6.5.

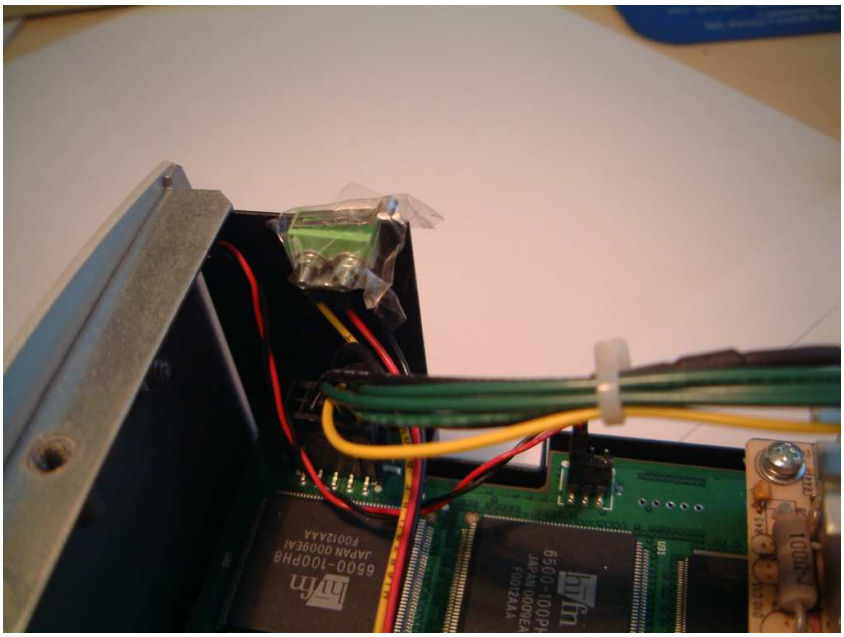


Figure 6.5: Lid switch on the Chrysalis Luna XL Accelerator

Sophisticated Measures

- *Custom-pressure Air Gaps* are probably the hardest physical tamper-resistance measure to defeat. An air-gap encases the entire board at the core of the device, or a particular intermediate layer, which contains pressure sensors connected to the tamper-responding loop. A random pressure can be chosen, and the

device configured to trigger should this pressure not be maintained. In order to defeat the device, either all pressure sensors must be disabled using some other attack technique, or the device must be disassembled within a pressure chamber. Herein lies the problem as pressure chambers are expensive, rare, difficult for humans to work inside, and tend not to fit other necessary equipment (e.g. focussed ion beam workstations) inside. Even then, the task of measuring the air pressure in the gap without affecting it will be a difficult one. Although appealing, custom-pressure air gaps are rarely deployed, probably due to practical difficulties of the design (e.g. compensating for the effect of changing temperatures on air pressure), and simply because in the commercial world they are in excess of security requirements.

- *Shaped Charges* are an appealing tamper-response, but seem to be the preserve of the military. The same sort of technology used for armour piercing is used on a smaller scale. There are a number of designs for the charges, that all create a partial implosive effect. An example is a cylinder containing high explosives, closed at one end, and with an inset cone of metal at the other, typically copper or aluminium. Upon detonation, the copper progressively implodes upon itself, forming an extremely hot and narrow plasma jet, which is highly effective at rapidly destroying key material, and large amounts of RAM or circuitry in general. Unless the action of drilling or opening somehow triggers the charges, they are usually a *tamper-responding* component of the system rather than actually a sensor. [35] shows a numerical simulation of a copper jet erupting from a shaped charge.

Problems with Tamper-Resistance

Tamper-resistance can be costly to provide. The manufacturing process is slower and more intricate, and of course testing and diagnostics may be more difficult with a tamper-resistant device. The tamper-responding sensors themselves may not be cheap. However, the big problem that designers run into is heat dissipation. Fast processors run hot these days, and the thought of not having a fan, let alone not even having a heat sink with air flow is ridiculous to some designers. Celestica [45] develop packaging technology for tamper-resistant devices, and describe thermal simulation methods for checking the properties of potted devices before prototyping in their paper [12]. It seems that IBM paid this price willingly: their 4758-002 has only an Intel 80486 processor running at 100MHz. Its successor, the PCIXCC, has a faster processor, as well as many dedicated hardware crypto acceleration chips. Allegedly, if all the chips function at once, the device could physically melt out of its potting!

6.2.1 Tamper-Evidence

Hardware Security Modules may be tamper-resistant and/or tamper-evident. Tamper-resistance is easy to understand – the device is designed such that any attempt to modify its operation or learn secrets held within is thwarted. The device's usual response to tampering is to erase its memory, destroying cryptographic key material and in many designs, rendering the device wholly nonfunctional. There are devices such as alarm systems that strive to remain functional as well as be tamper-resistant, but this behaviour is normally composed of conventional ceasing to function upon tampering coupled with multiple independent systems checking up on each other.

Tamper-evidence is more subtle. One might desire for it to be possible to check whether a particular boundary has been violated, that contains signing keys within. The keys can then be revoked if this is the case, and only limited damage would be done. However, whilst tamper-resistance is a property that a device holds with respect to the rest of its environment, tamper-evidence requires a notion of device identity, thus is defined in relation to an owner who must be able to identify the device itself.

Figure 6.6: A crude tamper-evident seal on a Chrysalis device

This is a hard problem. The most extreme case is for an attacker to extract all keys from a tamper-evident device, then load them legitimately into an identical looking device. This new device will be able to perfectly emulate the old one; it need then only look physically identical. A lot of deployed tamper-evidence measures are quite cursory, for example sticky label seals over screws as in figure 6.6. More rigorous measures such as engraving serial numbers and affixing holograms can go part way to making the physical device difficult to copy, but it might always be

feasible to construct a copy of a tampered device given enough care and attention. Tamper-evidence thus encapsulates some of the goals of anti-forgery methods. There may also be quite a difference between the visibility of tampering which succeeds in extracting key material, and the visibility of tampering which simply changes the device's operation to re-route its service requests elsewhere. In the case of a PCI card HSM, this secondary link could take the form of a radio transmitter hidden within the "tamper-evident" boundary. Alternatively, it might constitute a sabotaged driver chip outside the tamper-evident enclosure which was assumed not to be of security significance, that routes requests for the module back to a software emulator on the host itself.

6.3 HSM Summary

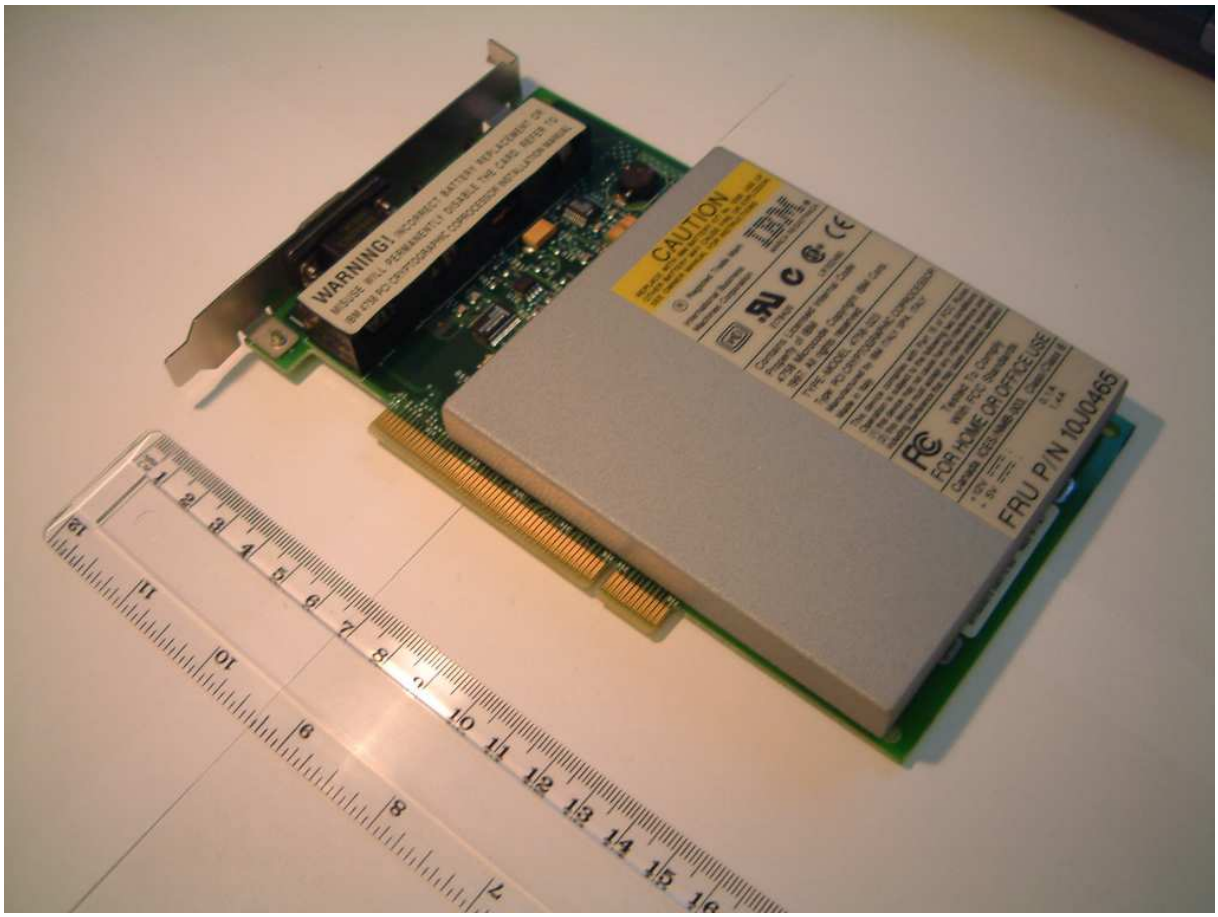
This section introduces the HSMs implementing the APIs analysed in this thesis. The survey is not exhaustive, and in particular only a sample of HSMs from manufacturers with large product lines are included.

6.3.1 IBM 4758-001

API	CCA / CCA+UDX / PKCS#11 / Full Custom
Processor	80486 DX2 66
Introduced	1997
Features	Level 4 Tamper-resistant enclosure
	Multi-layer code loading architecture
	Hardware DES and public key crypto acceleration
Form-factor	PCI Card
FIPS Validation	Level 4
Tamper-resistance	Gore Industries Membrane, Urethane potting
	Temperature & X-ray sensors, PRNG monitor
	Dual nested aluminium casings , PSU smoothing

6.3.2 IBM 4758-002

API	CCA / CCA+UDX / PKCS#11 / Full Custom
Processor	80486 DX4 100
Introduced	2000
Features	Level 4 Tamper-resistant enclosure
	Multi-layer code loading architecture
	Hardware DES and public key crypto acceleration
	Outbound Authentication
Form-factor	PCI Card
FIPS Validation	Level 4
Tamper-resistance	Gore Industries Membrane, Urethane potting
	Temperature & X-ray sensors, PRNG monitor
	Dual nested aluminium casings , PSU smoothing



6.3.3 nCipher nForce

API	nCore API / PKCS#11
Processor	ARM
Introduced	1999ish
Form-factor	PCI card / 5 1/2” drive bay (SCSI)
FIPS Validation	Level 3
Tamper-resistance	Potting, other measures unknown

6.3.4 nCipher nShield

API	nCore API / PKCS#11 / Full Custom
Processor	ARM
Introduced	1999ish
Form-factor	5 1/2” drive bay (SCSI)
FIPS Validation	Level 3
Tamper-resistance	Potting, other measures unknown

6.3.5 nCipher netHSM

API	nCore API / PKCS#11 / Full Custom
Processor	Power PC (Unconfirmed)
Introduced	Oct 2003
Form-factor	19” Rack-Mount 1x height
FIPS Validation	Not yet validated
Tamper-resistance	Internal HSM potted, other measures unknown

6.3.6 Prism TSM200

API	Proprietary
Processor	Dallas (uncertain)
Introduced	Mid 1990s
Form-factor	ISA Card
FIPS Validation	Not Validated
Tamper-resistance	Unknown

6.3.7 Thales RG7000

API	Proprietary
Processor	Unknown
Introduced	Late 1980s
Form-factor	19” Rack Mount 3x height (big heavy box)
FIPS Validation	Not validated
Tamper-resistance	Steel casing with mechanical lock lid-switch, optional trembler sensor

6.3.8 Atalla NSP10000

API	Proprietary
Processor	Unknown
Introduced	Mid 1990s
Form-factor	19” Rack-Mount 1x height
FIPS Validation	Not Validated (unclear)
Tamper-resistance	Steel casing with mechanical lock other measures unknown

6.3.9 Chrysalis-ITS Luna CA3

API	PKCS#11
Processor	StrongARM
Introduced	Late 1990s
Form-factor	PCMCIA Token
FIPS Validation	Level 3
Tamper-resistance	Some cards potted, PRNG monitor, other measures unknown

6.3.10 Visa Security Module

API	Proprietary
Processor	6502 (uncertain)
Introduced	Mid 1980s
Form-factor	19” Rack Mount 3x height (big heavy box)
FIPS Validation	Not Validated
Tamper-resistance	Steel casing with mechanical lock, lid-switch

Chapter 7

Analysis of Security APIs

7.1 Abstractions of Security APIs

At the core of any analysis technique is a condensed and efficient representation of the design that is to be reasoned about. It must be easy for the analyst to visualise and manipulate it in his head. This section describes several useful abstractions of Security APIs, each of which captures a slightly different aspect of API design.

7.1.1 Describing API Commands with Protocol Notation

It is easy to describe API commands using the conventional protocol notation that has been popular since the time of the BAN logic paper [11]. The notation used here is introduced with the oft-quoted Needham-Schroeder Public Key Protocol. It is a slightly simplified curly bracket notation, which does not bother with subscripts.

`A -> B : { Na , A }Kb`	$A \longrightarrow B: \quad \{N_A, A\}_{K_B}$
`B -> A : { Na , Nb }Ka`	$B \longrightarrow A: \quad \{N_A, N_B\}_{K_A}$
`A -> B : { Nb }Kb`	$A \longrightarrow B: \quad \{N_B\}_{K_B}$

In addition to understanding how to represent operations such as encryption and pairing, fairly standard conventions are in use for putting semantics into the variable names – `Na` is a nonce generated by `A`, `Ka^-1` may represent the private key of `A`. Similar conventions are required to concisely describe Security API commands.

K_M or `Km` is the master key of the HSM. HSMs with multiple master keys have each master key named after the type it represents. So

```
{ K1 }TC
```

Accepting a clear key value to become a 'TC' key

```
User -> HSM  : K1
HSM  -> User : { K1 }TC
```

Translating a key from encryption under key X to key Y

```
User -> HSM  : { K1 }X , { X }KM , { Y }KM
HSM  -> User : { K1 }Y
```

Adding together encrypted values

```
User -> HSM  : { A }KM , { B }KM
HSM  -> User : { A+B }KM
```

Verifying a password

```
User -> HSM  : { GUESS }KM , { ANS }KM
HSM  -> User : if GUESS=ANS then YES else NO
```

Figure 7.1: Example commands in protocol notation

represents a key `K1`, encrypted with the master key used for storing 'terminal communications' keys. Therefore `TC` itself is not the terminal communications key – `K1` is. This distinction needs to be held in mind when multiple layers of key are in use.

Transactions are represented by two lines of protocol, one describing arguments send, the other describing the result. A few examples are shown in figure 7.1.

The protocol representation may also borrow from other fields of mathematics and include pseudocode, as seen in last two examples in figure 7.1. The semantics of the command are hopefully still clear.

However, protocol notation gets into trouble because it represents the HSM parsing and decryption of the inputs implicitly. Many weaknesses in transaction sets arise because of interactions during decryption, and poor handling of error states. In these situations, using extra pseudocode in protocol notation becomes cumbersome once too many conditional actions have to be represented. The protocol lines in figure 7.2 show two common commands from the IBM 4758 CCA transaction set.

The first describes the final step of construction of a key encryption key (KEK), where the user provides the third of three component parts, and it is XORed together with the previous two. The key used to encrypt the first two components is the CCA master key, XORed with a control vector `imp/kp`. The '`/`' symbol in the control vector represents the use of XOR (shown elsewhere as '$\oplus$') specifically to combine control vectors and keys together; the semantics identical to '$\oplus$', but the visual cue

First Command: Key Part Import

```
User -> HSM  : KP3 , { KP1 ⊕ KP2 }KM/imp/kp , imp/kp
HSM  -> User : { KP1 ⊕ KP2 ⊕ KP3 }KM/imp
```

```
KEK1 = KP1 ⊕ KP2 ⊕ KP3
KEK2 = KP1 ⊕ KP2 ⊕ KP3 ⊕ (pmk ⊕ data)
```

Second Command: Key Import (in normal operation)

```
User -> HSM  : { PMK1 }KEK1/pmk , { KEK1 }KM/imp , pmk
HSM  -> User : { PMK1 }KM/pmk
```

Second Command: Key Import (when the attack is performed)

```
User -> HSM  : { PMK1 }KEK1/pmk , { KEK2 }KM/imp , data
HSM  -> User : { PMK1 }KM/data
```

Explanation of actions performed
1. HSM decrypts `{ KEK2 }KM/imp` with master key and implicitly claimed type `imp`
2. HSM decrypts `{ PMK1 }KEK1/pmk` using KEK2 and *explicitly* claimed type `data`
3. HSM encrypts `PMK1` with master key and explicitly claimed type `data`

Figure 7.2: The CCA typecasting attack represented in protocol notation

can help the reader identify the control vector. The control vector `imp/kp` represents the fact that it is a key part (`kp`), and that the finished key is to be of type *importer* (`imp`). On completion of the final command the combined key is returned in a ready to use form – encrypted under `KM`, with control vector `imp`.

The protocol notation in figure 7.2 appears to be performing well, but while it does describe a normal input and output of the transaction, it doesn't actually capture the semantics of what happens. Firstly, a fourth input is implicitly present – control information to let the HSM know that this is the last component, and it is to proceed with completing the key. If this control information were not supplied, the command would respond with `{ KP1 ⊕ KP2 ⊕ KP3 }KM/imp/kp`, which would not be usable. Secondly, the third input is an explicit naming of the control vector associated with the encrypted key part. If we assume that the HSM can explicitly identify the input data type (as is often the case in security protocols), we do not need this input, but we lose the semantics of the command, and we lose the opportunity to discover attacks which hinge on this.

When an operational key is imported under a KEK, its key type has to be explicitly stated as before. However, this time, the explicitly stated control vector can interact with the value of the KEK, which the attacker may have some degree of control over. If the attacker can formulate `KEK2`, then `PMK1` will be correctly decrypted and re-encrypted under the master key, except with a new type associated. The attack

works because the modification to the claimed KEK for import cancels out the error which would have otherwise occurred from wrongly specifying the type of the key to be imported as `data`. This attack is described in section 7.3.4. The important issue here is not to understand the attack, but to realise that the protocol notation not only fails to convey certain parts of transaction semantics, but *cannot* represent the semantics even when the author deliberately intends it to.

Representing decryption implicitly by matching keys and then stripping away encryption braces is not effective for describing many problems with APIs.

In conclusion, protocol notation is useful for two things:

1. explaining what a command achieves, and what it is supposed to do but, **not** how it works, and
2. showing the inputs and outputs of a command in a specific instance.

Discussion of the limitations of the notation is not a prerequisite to understanding the attacks described in this thesis, but is intended to serve as a warning to take care about notation for transaction sets when describing them to others. In section 7.4.4, a tool developed by the author for API analysis is described: its notation is sufficiently explicit to avoid these problems. However, even that notation is somewhat cumbersome when representing transactions with lots of conditional processing.

7.1.2 Key Typing Systems

Assigning and binding type information to keys is necessary for fine-grained access control to the key material and transactions. Designers must think carefully about key types and permitted usage when turning a high-level policy into a concrete API design – it is useful to envisage these restrictions as a type system.

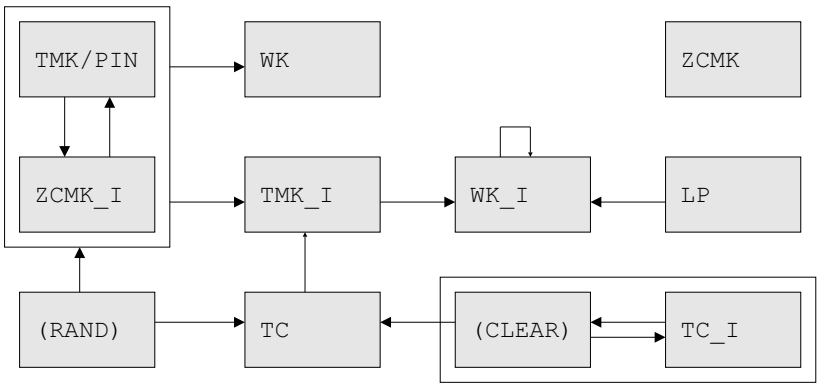


Figure 7.3: Example type system from the VSM

Many transactions have the same core functionality, and without key typing an attacker may be able to abuse a transaction he has permission to use, in order to

achieve the same functionality as one that is denied to him by the access control. For example, deriving a PIN for an account with a financial security API is simply encryption with DES, just as you would do with a data key for communications security. Likewise, calculation of a MAC can be equivalent to CBC encryption, with all but the last block discarded. A well-designed type system can prevent the abuse of the similarities between transactions.

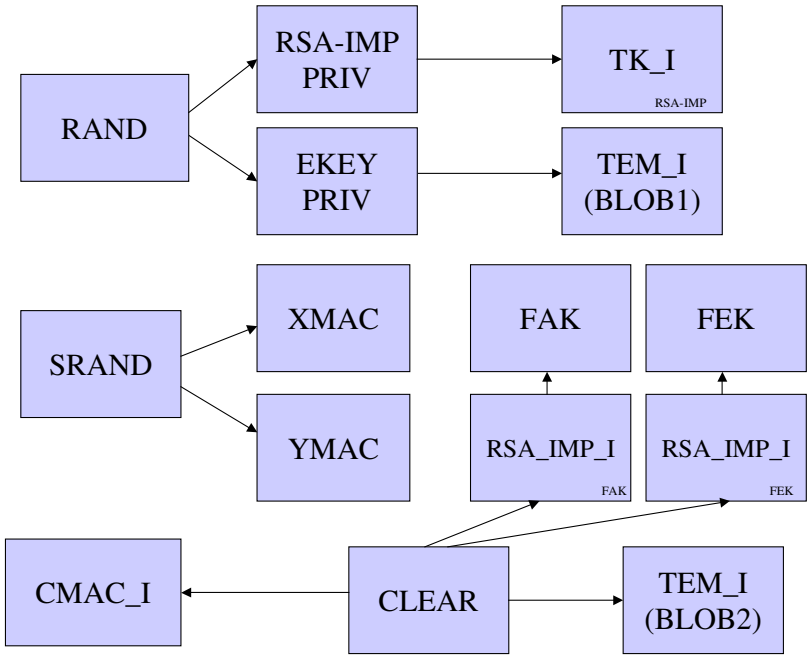


Figure 7.4: Example type system from real world application

Figures 7.3 and 7.4 show examples of a type system represented graphically. The transactions are shown as arrows linking two types together, and can represent only the major flow of information in the transaction. Consider figure 7.3. The labelled boxes represent the Security API's types. For a typical monotonic API a box will contain keys encrypted under the key named in the box label. For instance, a working key `W1` is considered to be of type `WK`, and will be presented to the box in encrypted forms as `{ W1 }WK`. The main point that must be grasped is that `WK` refers both to the type *Working Key*, and to the master key under which working keys are themselves encrypted.

Some labels have the suffix '`_I`' appended. This stands for the type of data encrypted under an *instance* of a particular type. Take for example `WK_I`. This type represents data that is encrypted under a particular working key e.g. `{ DATA }W1` where `W1` is the particular working key, and is presented to the HSM in the form `{ W1 }WK`. The box marked `WK_I` thus represents not really one type, but in fact a whole set of types.

Certain box names have specific meanings in all the diagrams: `CLEAR` represents unprotected, unencrypted clear data or key material that can be chosen by the user, `RAND` represents a random number generated by the HSM which is unknown to the user, `SRAND` represents an unknown but reproducible random number – such as one derived from encrypting some data of the user's choice.

Just like protocol notation, the graphic type system notation is not a formal one, but it can capture a lot of the semantics of a type system in quite a small space.

IBM's Common Cryptographic Architecture (CCA) deals with key typing in an interesting way. The CCA name for the type information of a key is a *control vector*. Rather than using completely disjoint master keys for types, the system of control vectors binds type information to encrypted keys by XORing the control vector with a single master key used to encrypt, and appending an unprotected copy (the *claimed type*) for reference.

$$E_{Km \oplus CV}(KEY) \ , \ CV$$

This control vector is simply a bit-pattern chosen to denote a particular type. If a naive attacker were to change the clear copy of the control vector (i.e. the claimed key type), when the key is used, the HSM's decryption operation would simply produce garbage.

$$D_{Km \oplus CV\_MOD}(E_{Km \oplus CV}(KEY)) \neq KEY$$

This mechanism is sometimes called *key diversification*, or *key variants*; IBM holds a number of patents in this area. The implementation details are in "Key Handling with Control Vectors" [33], and "A Key Management Scheme Based on Control Vectors" [34].

7.1.3 Key Hierarchies

Storage of large numbers of keys becomes necessary when protecting data from multiple sources, or originating from multiple users with differing levels of trust, as it limits damage if one key is compromised. Keys are commonly stored in a hierarchical structure, giving the fundamental advantage of efficient key sharing: access can be granted to an entire key set by granting access to the key at the next level up the hierarchy, under which the set is stored.

Confusion arises when the hierarchy serves more than one distinct role. Some HSMs infer the *type* of a key from its position in the hierarchy, or use hierarchies simply to increase their effective storage capacity when they can only retain a top-level key within their tamper-resistant enclosure.

Figure 7.5 shows a common key management hierarchy with three layers of keys. The top layer contains *'master keys'* which are never revealed outside the HSM, the middle layer *transport keys* or *key-encrypting-keys (KEKs)* to allow sharing between processors, and the bottom layer working keys and session keys – together known as *operational keys*, The scope of some HSMs extends to an even lower layer, containing data encrypted with the operational keys.

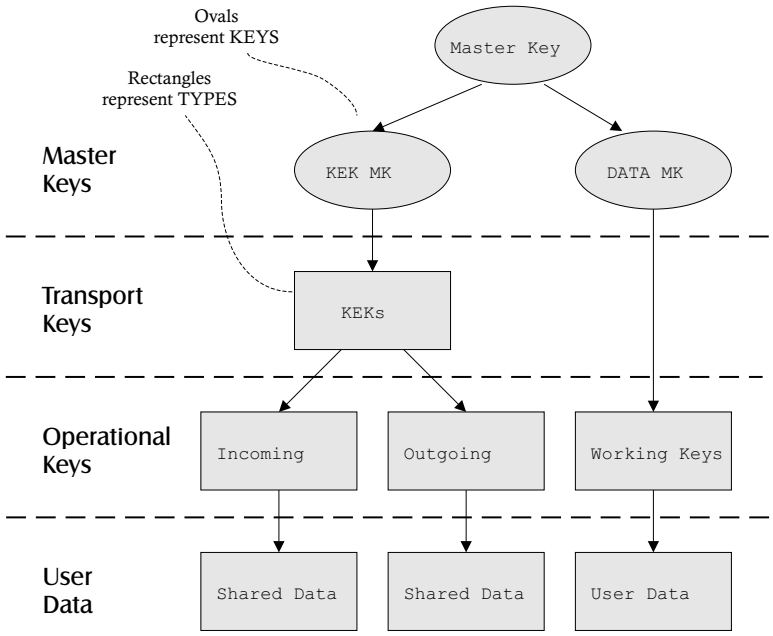


Figure 7.5: An example key hierarchy

7.1.4 Monotonicity and Security APIs

Many Security APIs contain very little internal state. The state of the system as a whole is usually stored as a set of encrypted terms, any of which can be presented as inputs to the commands. Once a new output has been produced by executing a command on a particular combination of inputs, this output can be added to the set of all terms known (referred to as the *Knowledge Set* of the user). If this set increases in size monotonically, it will always be possible to present a set of inputs again and retrieve the same output at a later time, and the ability to use a piece of knowledge can never be lost.

When a model of a Security API can demonstrate this property, certain sorts of analysis become much easier, as the problem becomes one of reachability – whether a particular piece of knowledge be obtained. The data structures for storing monotonically increasing knowledge can be much simpler and more condensed.

In real life, some APIs do come very close to perfect monotonicity. The Visa Security Module and similar designs have only the master key for the device as internal state. Monotonicity is broken during a master key update operation, but this is a privileged

command not available within the threat model of a realistic attacker, so as far as the attacker is concerned, the API is still monotonic.

APIs have to break this property to implement certain useful functionality such as counters. Counters are very useful to allow HSMs to dispense limited amounts of credit tokens (e.g. prepayment electricity meters, mobile phone top-up codes), and to limit the number of signatures which can be performed with a key. An API simply cannot be monotonic if counters are used, as they specifically break the property that a command execution can be repeated at any time.

One API – that of the Prism TSM200 – has a fundamentally non-monotonic design. Keys are stored in one hundred different internal slots, and once loaded are accessed by slot identifier rather than by presenting an encrypted input. The keys are arranged in a hierarchy, each key recording the identifier of its parent. Actions that change the contents of one slot trigger a cascading erasure of keys in slots which record the changed slot as their parent. Thus there is no simple guarantee that a key once available will remain available for use.

7.2 The Attacker's Toolkit

This section discusses the vulnerabilities found in Security APIs analysed during the course of the author's research. Some of the vulnerabilities are easily turned into attack implementations, whilst others are building blocks, which must be used in conjunction with other weaknesses to crystallise an attack. Section 7.3 describes attacks constructed from applications of these techniques.

7.2.1 Unauthorised Type-casting

Commonality between transactions makes the integrity of the type system almost as important as the access controls over the transactions themselves. Once the type constraints of the transaction set are broken, abuse is easy (e.g. if some high security key encrypting key (KEK) could be retyped as a data key, keys protected with it could be exported in the clear using a standard data deciphering transaction).

Certain type casts are only 'unauthorised' in so far as that the designers never intended them to be possible. In IBM's CCA, it is difficult to tell whether a given opportunity to type cast is a bug or a feature! IBM in fact describes a method in the appendix of the manual for their 4758 CCA [16] to convert between key types during import, in order interoperate with earlier products which used a more primitive type system. The manual does not mention how easily this feature could be abused. If type casting is possible, it should also be possible to regulate it at all stages with the access control functions.

The problem is made worse because HSMs which do not maintain internal state about their key structure have difficulty deleting keys. Once an encrypted version of a key has left the HSM it cannot prevent an attacker storing his own copy for later reintroduction to the system. Thus, whenever this key undergoes an authorised type cast, it remains a member of the old type as well as adopting the new type. A key with membership of multiple types thus allows transplanting of parts of the old hierarchy between old and new types. Deletion can only be effected by changing the master keys at the top of the hierarchy, which is radical and costly.

7.2.2 The Meet-in-the-Middle Attack

The idea behind the meet-in-the-middle attack is to perform a brute force search of a block cipher's key space to find not a particular key, but *any one of a large set of keys*. If you can encrypt some test pattern under a set of keys, and can check for a match against any ciphertext in this set simultaneously (for instance using a hash table), you have all you need. The maths is common sense: the more keys that you attack in parallel, the shorter the average time it takes to discover one of them by luck.

The technique is particularly powerful against security modules with monotonic APIs. Users will typically be able to generate as many keys as they wish, and store them locally on the hard drive. Once one of these keys has been discovered, it can normally be selected as the key is used to protect the output of a command, provided it is of the correct type. This is the price of using a type system to specify permitted actions: if even just one key within a type is discovered, a catastrophic failure can occur – select the cracked key, export the rest under it.

The attacker first generates a large number of keys. 2^{16} (65536) is a sensible target: somewhere between a minute's and an hour's work for the HSMs examined. The same test vector must then be encrypted under each key, and the results recorded. Each encryption in the brute force search is then compared against all versions of the encrypted test pattern. Checking each key may now take slightly longer, but there will be many fewer to check: it is much more efficient to perform a single encryption and compare the result against many different possibilities than it is to perform an encryption for each comparison.

In practical use, the power of the attack is limited by the time the attacker can spend generating keys. It is reasonable to suppose that up to 20 bits of key space could be eliminated with this method. Single DES fails catastrophically: its 56-bit key space is reduced to 40 bits or less. A 2^{40} search takes a few days on a home PC. Attacks on a 64-bit key space could be brought within range of funded organisations. The attack has been named a 'meet-in-the-middle' attack because the brute force search machine and the HSM attack the key space from opposite sides, and the effort expended by each meets somewhere in the middle.

Meet-in-the-Middle Maths

The average time between finding keys in a brute force search can be calculated by simple division of the search space by the number of target keys. However, it is more useful to consider the time to find the first key and this requires a slightly more complex model of the system using a Poisson distribution. The probability that the first r guesses to find a key will all fail is $e^{-\lambda r}$ where λ is the probability any given attempt matches (e.g. when trying to search for one of 16384 DES keys λ will be: $2^{14}/2^{56} = 2^{-42}$). An example calculation of the expected time to finding the first key using a hardware meet-in-the-middle DES cracker is in [53].

7.2.3 Key Conjuring

Monotonic HSM designs which store encrypted keys on the host computer can be vulnerable to unauthorised key generation. For DES keys, the principle is simple: simply choose a random value and submit it as an encrypted key. The decrypted result will also be random, with a 1 in 2^8 chance of having the correct parity. Some

early HSMs actually used this technique to generate keys (keys with bad parity were automatically corrected). Most financial APIs now check parity but rarely enforce it, merely raising a warning. In the worst case, the attacker need only make trial encryptions with the keys, and observe whether key parity errors are raised. The odds of 1 in 2^{16} for 3DES keys are still quite feasible, and it is even easier if each half can be tested individually (see the binding attack in section 7.2.5).

7.2.4 Related Key Attacks

Allowing related keys to exist within an HSM is dangerous, because it creates dependency between keys. Two keys can be considered *related* if the bitwise difference between them is known. Once the key set contains related keys, the security of one key is dependent upon the security of all keys related to it. It is impossible to audit for related keys without knowledge of what relationships might exist – and this would only be known by the attacker. Thus, the deliberate release of one key might inadvertently compromise another. *Partial relationships* between keys complicate the situation further. Suppose two keys become known to share certain bits in common: compromise of one key could make a brute force attack feasible against the other. Related keys also endanger each other through increased susceptibility of the related group to a brute force search (see the meet-in-the-middle attack in section 7.2.2). Note that the concept of related keys can be extended past partial relationships to purely *statistical relationships*. There is a danger during analysis that an architectural weakness gets spotted, but only one concrete manifestation of the unwanted relationship is removed, and the statistical relationship remains.

Keys with a *chosen* relationship can be even more dangerous because architectures using *key variants* combine type information directly into the key bits. Ambiguity is inevitable: the combination of one key and one type might result in exactly the same final key as the combination of another key and type. Allowing a *chosen difference* between keys can lead to opportunities to subvert the type information, which is crucial to the security of the transaction set.

Although in most HSMs it is difficult to enter completely chosen keys (this usually leads straight to a severe security failure), obtaining a set of unknown keys with a chosen difference can be quite easy. Valuable keys (usually KEKs in the hierarchy diagram) are often transferred in multiple parts, combined using XOR to form the final key. After generation, the key parts would be given to separate couriers, and then passed on separate data entry staff, so that a dual control policy could be implemented: only collusion would reveal the value of the key. However, any key part holder could modify his part at will, so it is easy to choose a relationship between the actual value loaded, and the intended key value. The entry process could be repeated twice to obtain a pair of related keys. The Prism TSM200 architecture actually allowed a chosen value to be XORed with almost any key at any time.

7.2.5 Poor Key-half Binding

The adoption of 3DES as a replacement for DES has led to some unfortunate API design decisions. As two-key 3DES key length is effectively 128 bits (112 bits of key material, plus 16 parity bits), cryptographic keys do not fit within the 64-bit DES block size. Manufacturers have thus come up with various different approaches to storing these longer keys in encrypted form. However, when the association between the halves of keys is not kept, the security of keys is crippled. A number of APIs allow the attacker to manipulate the actual keys simply by manipulating their encrypted versions in the desired manner. Known or chosen key halves can be substituted into unknown keys, immediately halving the keyspace. The same unknown half could be substituted into many different keys, creating a related key set, the dangers of which are described in section 7.2.4.

3DES has an interesting deliberate feature that makes absence of key-half binding even more dangerous. A 3DES encryption consists of a DES encryption using one key, a decryption using a second key, and another encryption with the first key. If both halves of the key are the same, the key behaves as a single length key. ($E_{K1}(D_{K2}(E_{K1}(data))) = E_K(data)$ when $K = K1 = K2$). Pure manipulation of unknown key halves can yield a 3DES key which operates exactly as a single DES key. Some 3DES keys are thus within range of a brute force cracking effort.

7.2.6 Differential Protocol Analysis

Transactions can be vulnerable even if they do not reveal a secret completely: partial information about the value of a key or secret is often good enough to undermine security. *Differential Protocol Analysis*, coined by Anderson and the author in [4], is the method of attacking APIs by methodically varying input parameters, and looking for differentials in output parameters which vary in a way dependent upon the target secret. An individual command can be treated like a black box for the purposes of differential analysis, though of course it may help the attacker to study the internal workings of the command to choose an appropriate differential.

The first instances of differential attacks on Security APIs were only published during early 2003; there are not enough examples available to reason about the general case. The examples that have been discovered tend to exploit a trivial differential – that between normal operation, and some error state. This is the case in Clulow's PAN modification attack [15]. However, the same weaknesses that leak information through crude error responses, or via a single bit yes/no output from a verification command for example, are at work in the outputs of other API commands.

The principle of differential protocol analysis is best illustrated through the example of the decimalisation table attack on PIN generation in a financial security HSM. In this section, the attack is described in a simplified way – full details are in section 7.3.10.

PIN numbers are often stored in encrypted form even at machines which have the necessary key to derive the PIN from the account number of a customer. These encrypted PINs are useful when giving PIN verification capability to a financial institution who can be trusted with verifying PINs in general, but may not be trusted enough to take a copy of the PIN derivation key itself, or when sending correct PINs to a mailer printing site. The API thus has a command of the following form:

```
User -> HSM : PAN , { PMK }KM , dectab
HSM -> User : { PIN1 }LP
```

where `PAN` is the Primary Account Number of the customer, `PMK` is the PIN derivation key (encrypted with the master key) and `dectab` is the decimalisation table used in the derivation process. The PIN is returned encrypted with key `LP` – a key for local PIN storage[1].

Section 7.3.10 describes an attack on a PIN verification command, which calculates a customer PIN from a PAN using a specified decimalisation table, and then gives a yes/no answer as to whether a guess matches it.This PIN verification command can be used to learn information about the secret PIN. With an average of 5000 guesses, the verification command will respond with a *yes*, and the PIN is discovered. However, the attack improves on this. It works by changing a digit in the decimalisation table, and observing whether or not the PIN generation process interacts with the specific digit in the table changed. If there is an interaction, the PIN verification will fail instead of succeed, and from this fact, one of the digits composing the PIN can be deduced. The attack can be thought of as improving the rate of information leakage from the command from 1 combination to about 1000 combinations per guess. However, when the attacker does not have access to enter chosen PINs, this variant of the attack seems to be fixed. This is not the case, because the information leakage can still manifest itself as a differential. Look now at the PIN generation procedure in figure 7.6: only some of the digits in the decimalisation table will affect the specific PIN generated for a particular account, as the result of encrypting the PAN can only contain up to four different hexadecimal digits in the first four characters.

Thus if the decimalisation table is modified from its correct value, as in transaction B, if the modification affects the PIN generated, a differential will appear between the values of the first and second encrypted PINs . This is a true differential attack because neither run of the protocol reveals any information about the PIN in its own right. This is unlike the decimalisation table attack upon the verification phase, where both the normal and attack runs of the command each leak information.

[1]For a description of an attack on a command of this form which does not rely upon the manipulation of the decimalisation table, see the forthcoming paper "Encrypted? Randomised? Compromised? (When cryptographically secured data is not secure)"

	(Transaction A)	(Transaction B)
PAN	4556 2385 7753 2239	4556 2385 7753 2239
Raw PIN	3F7C 2201 00CA 8AB3	3F7C 2201 00CA 8AB3
Truncated PIN	3F7C	3F7C
	0123456789ABCDEF 0123456789012345	0123456789ABCDEF 0120456789012345
Decimalised PIN	3572	0572
PIN Block	4F1A 32A0 174D EA68	C3AA 02D6 7A8F DE21

Figure 7.6: A differential in outputs of the Encrypted PIN Generate Command

The verification attack described in section 7.3.10 could be considered an instance of a differential attack, but with a trivial differential. Many of the attacks discovered on financial APIs by the author and Clulow exploit such trivial differentials, as these seem to be comparatively easy to spot. However, when fixing the APIs to prevent the attacks, all types of differential must considered, not just the trivial cases. Furthermore, even when protocol output differentials are secured, there are always the threats of timing, power or electromagnetic differential attacks on the security module.

7.2.7 Timing Attacks

Timing attacks are an important element in the toolkit of a software only attacker, as unlike power analysis or electromagnetic emissions attacks, they do not necessarily require physical tampering with the host or access to the exterior of the HSM. With sabotaged device drivers, it should be possible in many cases to perform timing attacks from the host computer, and if the clock speed of the host processor is substantially higher than that of the HSM, counting iterations of a tight loop of code on the host should suffice for timing. Embedded processors inside HSMs run at only several hundred megahertz at the most, so a host machine with a clock rate of 1GHz or above should have no difficulty at all in measuring timing differences between executions of HSM commands to instruction granularity. Many comparison operations between long character strings are implemented using `memcpy`, or in a loop that drops out as soon as the first discrepancy is discovered. This sort of timing attack was used long ago to find partial matches against passwords in multi-user

operating systems. Naively implemented RSA operations also have data-dependent timing characteristics.

7.2.8 Check Value Attacks

Nearly all HSM designs use some sort of check value to allow unique identification of keys, establish that the correct one is in use, or that a key manually loaded has been entered correctly. Older HSM designs revolved around symmetric cryptography, and a natural way chosen to create a check value was to encrypt a known constant with the key, and return part or all of the ciphertext as a check value. Any module supporting this sort of check value comes immediately under attack if the known plaintext encrypted under a key can be passed off as genuine data encrypted under that key.

A variety of lengths of check value are in common use, the differences dictated by the different understandings of the primary threat by the designers. Full-length check values on single DES keys were rapidly seen as targets for brute force attacks, and furthermore as at risk from dangerously interacting with the set of encrypted values under that key. The Visa Security Module and its clones have shortened check values to six hex digits, in particular to avoid potential usage of the encrypted check value as a known key, but some other designs do not bother to shorten the value. Unfortunately some general-purpose APIs found themselves needing to support calculation of a range of different types of check value, in order to perform checks on the secure exchange of keys between that device and an older one. These APIs thus had configurability of their calculation method. This is the worst case of all, as even if a system only uses short check values, it may still be possible to calculate long ones.

Check values do not just open up the risk of pure brute force attacks, they can also enable the meet-in-the-middle attack on suitably sized key spaces, by providing a test for a correct match; they can also undermine key binding – for example in the Prism TSM200 where each half of the 3DES master key had a separate check value.

Modern APIs such as nCipher's nCore API identify a key uniquely by hashing it along with its type information. It is computationally infeasible to create a SHA1 collision, so these identifiers are generally speaking safe to use. However, in order to be useful as a check value function, it must function as a test for equality between keys, and there may be a few peculiar circumstances where this is dangerous, and check values should not be used at all. Take for example a situation where a large random key space is populated with only a comparatively small number of keys (for example if 3DES keys were derived from encryption of a PAN with a known key). As there is not a vast number of PANs, all PANs could be tried, and their check values compared with those of the encrypted derived keys. In these circumstances, supporting calculation of a check value on the keys would not be prudent.

7.3 An Abundance of Attacks

This section describes the concrete attacks known on Security APIs, which are comprised of one or more of the building blocks described in the attacker's toolkit. All these attacks are concrete in the sense that the API on which they operate was sufficiently well defined to be certain that it was vulnerable. In most cases the attacks themselves have been implemented on real modules too, which lends further credibility (though the level of correspondence between specification and implementation must always be taken into account).

In addition to the attacks in this section, there are a few more general attacks, which (for example) rely upon external factors such as poor design of procedural controls; these are described *in situ* in discussions in other chapters of the thesis. The last section (7.3.12) includes a list of attacks described elsewhere in the thesis, and also briefly describes attacks newly developed by the author; these appear to be very significant but cannot be adequately discussed until they are better understood.

7.3.1 VSM Compatibles – XOR to Null Key Attack

Anderson, 2000, [1]

The 'XOR to Null Key' attack was discovered by Anderson, and affected the Visa Security Module and many compatible clones. Modern implementations of the VSM API have been fixed.

The VSM's primary method for importing top level keys is from clear components, written by hand on pieces of paper, or printed into PIN mailer stationery. This methodology was used for establishing Zone Control Master Keys (ZCMKs) and also Terminal Master Keys (TMKs). The TMK establishment procedure consisted of two commands – a privileged console command for generating a TMK component, and an unrestricted command '`IG`', which was used to combine key components. The procedure for loading a key would thus be as follows:

```
(Key Switch Turned On)

HSM -> Printer : K1                  (Generate Component)
HSM -> Host    : { K1 }TMK
HSM -> Printer : K2                  (Generate Component)
HSM -> Host    : { K2 }TMK

(Key Switch Turned Off)

U -> HSM : { K1 }TMK , { K2 }TMK     (Combine Key Parts)
HSM -> U : { K1 ⊕ K2 }TMK
```

The supervisor key switch is turned, which enables sensitive commands to be run from the console – in particular, the generation of key components. Each security officer then retrieves her key component mailer from the printer. Once the components have been entered, supervisor state is cancelled, and a program is run on the host which calls the `IG` command to combine the components and form the final TMK. This procedure was followed for years before the retrospectively simple attack was spotted: the key components are combined using XOR, so if the same component is combined with itself, a key of all zeroes will necessarily result.

```
U -> HSM : { K1 }TMK , { K1 }TMK    (Combine Key Parts)
HSM -> U : { 0000000000000000 }TMK
```

Once there is a known TMK in the system, other transactions allow the encryption of other TMKs, or even PIN master keys under this key. A complete compromise results.

The VSM and its successors were 'fixed' by making the `IG` command a privileged transaction.

7.3.2 VSM Compatibles – A Key Separation Attack

Bond, 2000, [8]

The amalgamation of the `TMK` and `PIN` types in the VSM design is a weakness that can be exploited many ways. One possible attack is to enter an account number as a `TC` key, and then translate this to encryption under a `PIN` key. The command responsible is designed to allow `TC` keys to be encrypted with a `TMK` for transfer to an ATM, but because `TMK`s and `PIN` keys share the same type, the `TC` key can also be encrypted under a `PIN` key in the same way. This attack is very simple and effective, but is perhaps difficult to spot because the result of encryption with a PIN key is a sensitive value, and it is counterintuitive to imagine an encrypted value as sensitive when performing an analysis. Choosing a target account number `PAN`, the attack can be followed on the type transition diagram in figure 7.7, moving from `(CLEAR)` to `TC` (1), and finally to `TMK_I` (2).

```
(1)  User -> HSM : PAN
     HSM -> User : { PAN }TC

(2)  User -> HSM : { PAN }TC , { PMK1 }TMK
     HSM -> User : { PAN }PMK1
```

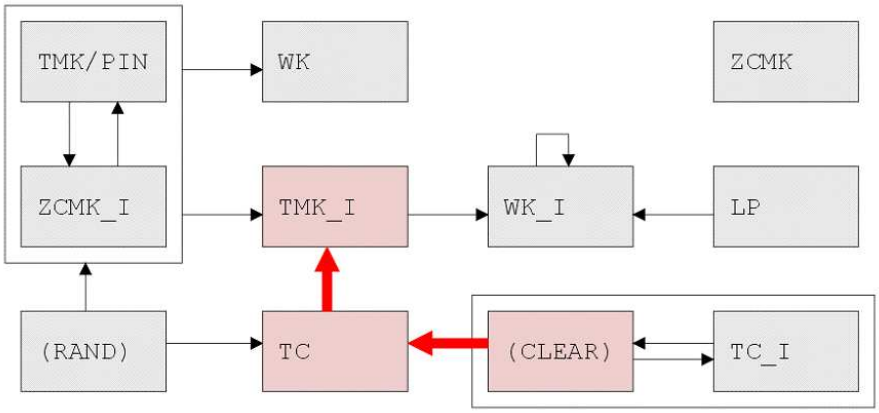


Figure 7.7: Type diagram for VSM with attack path highlighted

7.3.3 VSM Compatibles – Meet-in-the-Middle Attack

Bond, 2000, [8]

The meet-in-the-middle attack can be used to compromise eight out of the nine types used by the VSM. As is typical of monotonic APIs, the VSM does not impose limits or special authorisation requirements for key generation, so it is easy to populate all the types with large numbers of keys. Furthermore, it *cannot* properly impose restrictions on key generation because of the 'key conjuring' attack (section 7.2.3) which works with many HSMs which store keys externally.

The target type should be populated with at least 2^{16} keys, and a test vector encrypted under each. In the VSM, the dedicated 'encrypt test vector' command narrowly escapes compromising all type because the default test vector (which is `0123456789ABCDEF`) does not have the correct parity to be accepted as a key. Instead, the facility to input a chosen terminal key (`CLEAR` $\longrightarrow$ `TC` in figure 7.7) can be used to create the test vectors. The final step of the attack is to perform the 2^{40} brute force search offline.

The obvious types to attack are the `PIN/TMK` and `WK` types. Once a single `PIN/TMK` key has been discovered, all the rest can be translated to type `TMK_I`, encrypted under the compromised TMK. The attacker then decrypts these keys offline (e.g using a home PC). Compromise of a single Working Key (`WK`) allows all trial PINs entered by customers to be decrypted by translating them from encryption under their original `WK` to encryption under the compromised one (this command is shown by the looping arrow on `WK_I` in figure 7.7).

7.3.4 4758 CCA – Key Import Attack

Bond, 2000, [8]

One of the simplest attacks on the 4758 is to perform an unauthorised type cast using IBM's *'pre-exclusive-or'* type casting method [16]. A typical case would be to import a PIN derivation key as a data key, so standard data ciphering commands could be used to calculate PIN numbers, or to import a KEK as a `DATA` key, to allow eavesdropping on future transmissions. The `Key_Import` command requires a KEK with permission to import (an `IMPORTER`), and the encrypted key to import. The attacker must have the necessary authorisation in his access control list to import to the destination type, but the original key can have any type. Nevertheless, with this attack, all information shared by another HSM is open to abuse. More subtle type changes are possible, such as re-typing the right half of a 3DES key as a left half.

A related key set must first be generated (1). The `Key_Part_Import` command acts to XOR together a chosen value with an encrypted key. If a dual control policy prevents the attacker from access to an initial key part, one can always be conjured (section 7.2.3). The chosen difference between keys is set to the difference between the existing and desired control vectors. Normal use of the `Key_Import` command would import `KEY` as having the `old_CV` control vector. However, the identity `(KEK1`$\oplus$`old_CV) = (KEK2`$\oplus$`new_CV)` means that claiming that `KEY` was protected with `KEK2`, and having type `new_CV` will cause the HSM to retrieve `KEY` correctly (3), but bind in the new type `new_CV`.

Related Key Set	(1)	`KEK1 = KORIG` `KEK2 = KORIG` $\oplus$ `(old_CV` $\oplus$ `new_CV)`
Received Key	(2)	$E_{\texttt{KEK1}\oplus\texttt{old\_CV}}(\texttt{ KEY })$, `old_CV`
Import Process	(3)	$D_{\texttt{KEK2}\oplus\texttt{new\_CV}}(\ E_{\texttt{KEK1}\oplus\texttt{old\_CV}}(\texttt{PKEY})\ ) = \texttt{PKEY}$

Of course, a successful implementation requires circumvention of the bank's procedural controls, and the attacker's ability to tamper with his own key part. IBM's advice is to take measures to prevent an attacker obtaining the necessary related keys. Optimal configuration of the access control system can indeed avoid the attack, but the onus is on banks to have tight procedural controls over key part assembly, with no detail in the manual as to what these controls should be. The manual will be fixed [23], but continuing to use XOR will make creating related key sets very easy. A long-term solution is to change the control vector binding method to have a one-way property, such that the required key difference to change between types cannot be calculated – keys and their type information cannot be unbound.

7.3.5 4758 CCA – Import/Export Loop Attack

Bond, 2000, [8]

The limitation of the key import attack described in 7.3.4 is that it only applies to keys sent from other HSMs, because they are the only ones that can be imported. The Import/Export Loop attack builds upon the Key Import attack by demonstrating how to export keys from the HSM, so their types can be converted as they are re-imported.

The simplest Import/Export loop would have the same key present as both an importer and an exporter. However, in order to achieve the type conversion, there must be a difference of `(old_CV⊕new_CV)` between the two keys. Generate a related key set (1), starting from a conjured key part if necessary. Now conjure a new key part `KEKP`, by repeated trial of key imports using `IMPORTER1`, and claiming type `importer_CV`, resulting in (2). Now import with `IMPORTER2`, claiming type `exporter_CV`, the type changes on import as before (3).

(1) $$\begin{aligned} \texttt{IMPORTER1} &= \texttt{RAND} \\ \texttt{IMPORTER2} &= \texttt{RAND} \oplus (\texttt{importer\_CV} \oplus \texttt{exporter\_CV}) \end{aligned}$$

(2) $$E_{\texttt{IMPORTER1}\oplus\texttt{importer\_CV}}(\texttt{KEKP})$$

(3) $$D_{\texttt{IMPORTER2}\oplus\texttt{exporter\_CV}}(E_{\texttt{IMPORTER1}\oplus\texttt{importer\_CV}}(\texttt{KEKP})) = \texttt{KEKP}$$

(4) $$\texttt{EXPORT\_CONVERT} = \texttt{KEKP}$$

(5) $$\begin{aligned} \texttt{IMPORT\_CONVERT1} &= \texttt{KEKP} \oplus (\texttt{source1\_CV} \oplus \texttt{dest1\_CV}) \\ &\cdots \\ \texttt{IMPORT\_CONVERTn} &= \texttt{KEKP} \oplus (\texttt{source1\_CV} \oplus \texttt{destn\_CV}) \end{aligned}$$

Now use `Key_Part_Import` to generate a related key set (5) which has chosen differences required for all type conversions you need to make. Any key with export permissions can now be exported with the exporter from the set (4), and re-imported as a new type using the appropriate importer key from the related key set (5). IBM recommends audit for same key used as both importer and exporter [16], but this attack employs a relationship between keys known only to the attacker, so it is difficult to see how such an audit could succeed.

7.3.6 4758 CCA – 3DES Key Binding Attack

Bond, 2000, [8]

The 4758 CCA does not properly bind together the halves of its 3DES keys. Each half has a type associated, distinguishing between left halves, right halves, and single DES keys. However, for a given 3DES key, the type system does not specifically associate the left and right halves as members of that instance. The 'meet-in-the-middle' technique can thus be successively applied to discover the halves of a 3DES key one at a time. This attack allows *all keys* to be extracted, including ones which do not have export permissions, so long as a known test vector can be encrypted.

4758 key generation gives the option to generate *replicate 3DES keys*. These are 3DES keys with both halves having the same value. The attacker generates a large number of replicate keys sharing the same type as the target key. A meet-in-the-middle attack is then used to discover the value of two of the replicate keys (a 2^{41} search). The halves of the two replicate keys can then be exchanged to make two 3DES keys with differing halves. Strangely, the 4758 type system permits distinction between true 3DES keys and replicate 3DES keys, but the manual states that this feature is not implemented, and all share the generic 3DES key type. Now that a known 3DES key has been acquired, the conclusion of the attack is simple; let the key be an exporter key, and export all keys using it.

If the attacker does not have the CCA role-based access control (RBAC) permissions to generate replicate keys, he must generate single length DES keys, and change their left half control vector to *'left half of a 3DES key'*. This type casting can be achieved using the *Key Import attack* (section 7.3.4). If the value of the imported key cannot be found beforehand, 2^{16} keys should be imported as *'single DES data keys'*, used to encrypt a test vector, and an offline 2^{41} search should find one. Re-import the unknown key as a *'left half of a 3DES key'*. Generate 2^{16} 3DES keys, and swap in the known left half with all of them. A 2^{40} search should yield one of them, thus giving you a known 3DES key.

If the attacker cannot easily encrypt a known test pattern under the target key type (as is usually the case for KEKs), he must bootstrap upwards by first discovering a 3DES key of a type under which he has permissions to encrypt a known test vector. This can then be used as the test vector for the higher level key, using a `Key_Export` to perform the encryption.

A given non-exportable key can also be extracted by making two new versions of it, one with the left half swapped for a known key, and likewise for the right half. A 2^{56} search would yield the key (looking for both versions in the same pass through the key space). A distributed effort or special hardware would be required to get results within a few days, but such a key would be a valuable long term key, justifying the expense. A brute force effort in software would be capable of searching for all non-exportable keys in the same pass, further justifying the expense.

7.3.7 4758 CCA – Key_Part_Import Descrack Attack

Clayton & Bond, 2001, [13]

A number of attack instances in this section show techniques from the attack toolkit applied to the 4758 CCA revealing a vulnerability. However, whilst existence of the vulnerabilities is difficult to deny, it is debatable whether the particular configurations of the CCA RBAC system typically used will prevent a full and complete extraction of key material. This attack's goal is to extract a 3DES key with export permissions in the clear, using as few access permissions as possible – with the aim of staying with a realistic threat model. The explanation here is primarily taken from "Experience Using a Low-Cost FPGA Design to Crack DES Keys" [13], but focusses on the attack methodology, rather than the DES cracker design.

Performing the Attack on the HSM

A normal attack on the CCA using the meet-in-the-middle tool (section 7.2.2) and the related key tool (section 7.2.4) is fairly straightforward to derive, and consists of three stages, shown in figure 7.8 and described below:

(1) Test Pattern Generation: Discover a normal data encryption key to use as a test pattern for attacking an exporter key. This is necessary because exporter keys are only permitted to encrypt other keys, not chosen values. The method is to encrypt a test pattern of binary zeroes using a set of randomly generated data keys, and then to use the meet-in-the-middle attack to discover the value of one of these data keys.

(2) Exporter Key Harvesting: Use the known data key from stage (1) as a test pattern to generate a second set of test vectors for a meet-in-the-middle attack that reveals two *double-length replicate exporter keys* (replicate keys have both halves the same, thus acting like single DES keys). Once this stage is complete, the values of two of the keys in the set will be known.

(3) Valuable Data Export: Retrieve the valuable key material (e.g. PIN derivation keys). This requires a known double-length exporter key, as the CCA will not export a 3DES key encrypted under a single DES exporter key, for obvious security reasons. Here, the key-binding flaw in the CCA software is used to swap the halves of two known replicate keys from stage (2) in order to make a double-length key with unique halves. This full 3DES key can then be used for the export process.

However, the above approach is far from ideal because it requires multiple phases of key cracking and illicit access to the HSM. In order to perform the attack in a single access session, the second set of test vectors has to be generated immediately after the first. However, it is not possible to know in advance which data key from the set will be discovered by the search, in order to use it as a test pattern. Generating a second set of test vectors for every possible data key would work in principle, but

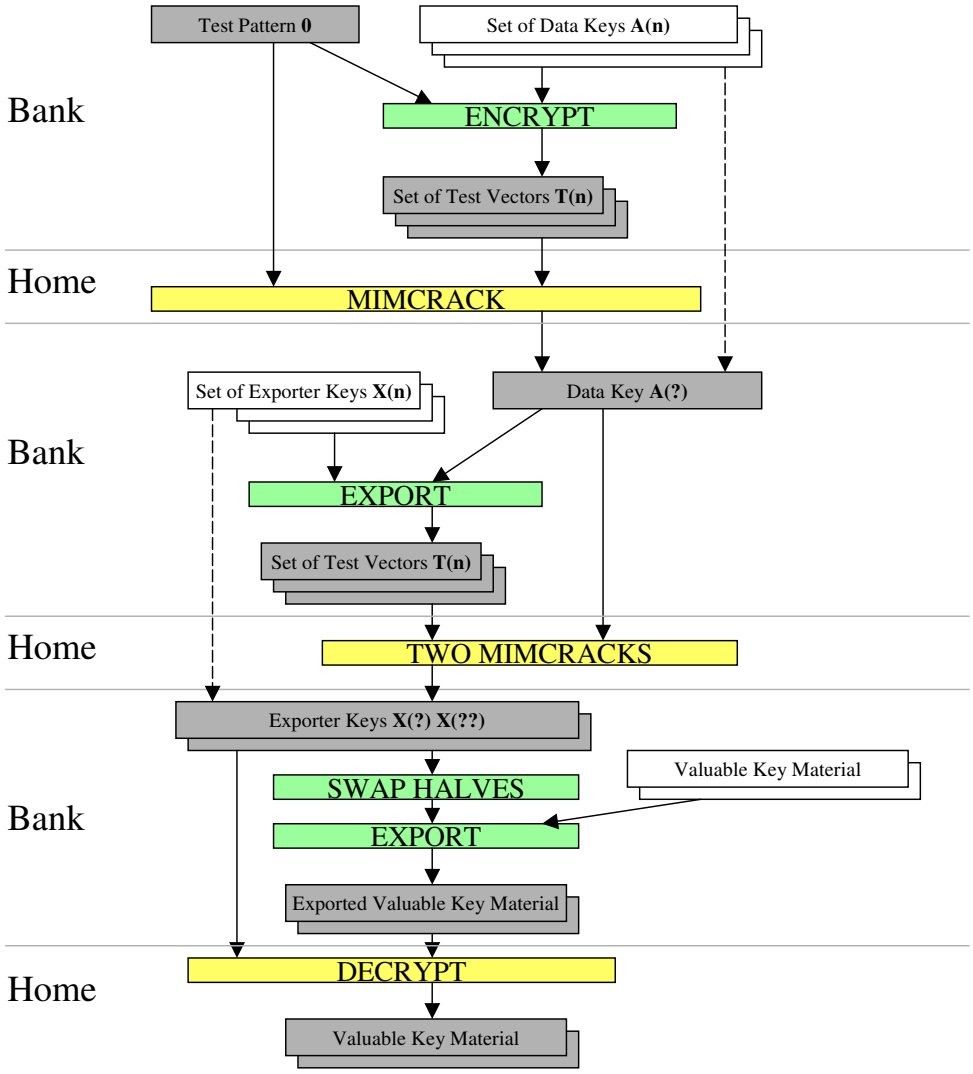


Figure 7.8: Standard implementation of attack on 4758 CCA

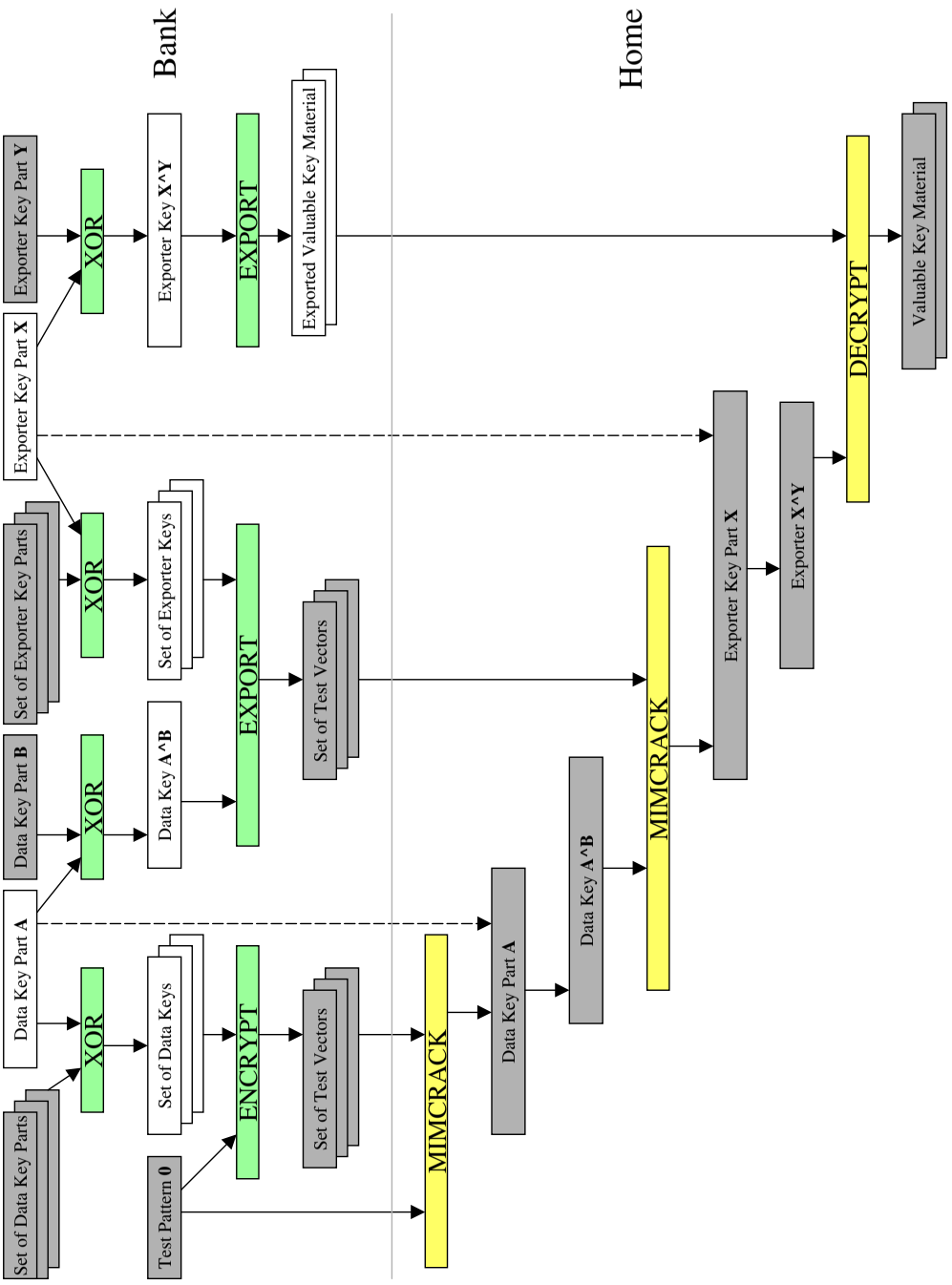


Figure 7.9: Optimised implementation of attack on 4758 CCA

the number of operations the HSM would have to perform would be exponentially increased, and at the maximum transaction rate for a 4758 (roughly 300 per second), collecting this data set would take ten days of unauthorised access.

So the first stage of the online attack had to yield the value of a particular data key that was chosen in advance, which could then be used as the test pattern for the second stage. The solution is shown in figure 7.9. It is first necessary to create a related key set using the `Key_Part_Import` command. From the discovery of any single key, the values of all of the rest can be calculated. This related key set is made by generating an unknown data key part and XORing it with 2^{14} different known values (for instance, the integers $0 \ldots 16383$). Any one of the keys can then immediately be used for the second stage of the attack, even though its actual value will only be discovered later on.

The second stage is to export this single data key under a set of double-length replicate exporter keys and to use a meet-in-the-middle attack on the results. Two keys need to be discovered so that their halves can be swapped to create a non-replicate exporter key. Once again the same problem arises – it is impossible to tell in advance which two keys will be discovered, and so the valuable key material cannot be exported until after the cracking was complete. Generating a set of related exporter keys again solves the problem. Discovering just one replicate key now gives access to the *entire* set. Thus a double-length exporter with unique halves can be produced prior to the cracking activity by swapping the halves of any two of the related keys.

Implementation of this second stage of the process reveals an interesting and well-hidden flaw in the `Key_Part_Import` command. The concept of binding flaws has already been identified in the encrypted key tokens (see section 7.3.6), but it is also present in `Key_Part_Import`: it is possible to subvert the creation of a double-length replicate key so as to create a uniquely halved double-length key by the simple action of XORing in a new part with differing halves. This second instance of the flaw was discovered during the process of trying to implement the naive three stage attack for real.

Finally, the new double-length exporter key made from the unknown replicate key part from stage two can be used to export the valuable key material, as is visible in figure 7.9. The attack retains the three conceptual stages, but there is no dependency on the values of cracked keys during the period of access to the HSM. This allows the data collection for all three stages to be run in a single session and the cracking effort to be carried out in retrospect.

Cracking the DES Keys

A home PC can be used for the DES key cracking, and this might be typical of the resources immediately available to a real-world attacker. However, experiments

performed when the attack was discovered showed that cracking a single key from a set of 2^{16} would take a typical 800 MHz machine about 20 days. It may not be possible to increase the number of test vectors collected, as 2^{16} is roughly the maximum number of encrypted results that can be harvested during a "lunch-break-long" period of access to the CCA software. "No questions asked" access to multiple PCs in parallel is also a substantial risk, so a faster method is preferable to allow the attack to complete before a bank's audit procedures might spot the unauthorised access to their HSM.

Given the benefits of implementing DES in hardware, and the flexibility and ease of implementation associated with FPGAs, Altera's "Excalibur" NIOS evaluation board [40] was a promising candidate platform for implementing a DES cracker. The NIOS evaluation board is an off-the-shelf, ready-to-run, no-soldering-required system, and comes complete with all the tools necessary to develop complex systems. Altera generously donated a board for free; in 2001 its retail price was US$995.

The basic idea of a brute force "DES cracker" is to try all possible keys in turn and stop when one is found that will correctly decrypt a given value into its plaintext; this is the sort of machine that was built by the EFF in 1998 [21]. To crack key material with known test vectors, the cracker works the other way round; it takes an initial plaintext value and encrypts it under incrementing key values until the encrypted output matches one of the values being sought. The design implemented runs at 33.33 MHz, testing one key per clock cycle. This is rather slow for cracking DES keys – and it would take, with average luck, 34.6 years to crack a single key. However, the attack method allows many keys to be attacked in parallel and because they are all related it does not matter which one is discovered first.

The design was made capable of cracking up to 2^{14} keys in parallel (i.e. it simultaneously checked against the results of encrypting the plaintext with 2^{14} different DES keys). The particular Excalibur board being used imposed the 16384 limitation; if more memory had been available then the attack could have proceeded more quickly. The actual comparison was done in parallel by creating a simple hash of the encrypted values (by XORing together groups of 4 or 5 bits of the value) and then looking in that memory location to determine if an exact match had occurred. Clearly, this gives rise to the possibility that some of the encrypted values obtained from the 4758 would need to be stored in identical memory locations. We just discarded these clashes and collected rather more than 2^{14} values to ensure that the comparison memory would be reasonably full.

As already indicated, the attack requires two cracking runs, so one would hope to complete it in just over 2 days. In practice, the various keys we searched for were found in runs taking between 5 and 37 hours, which is well in accordance with prediction.

Implementation Overview

The DES cracker was implemented on the Altera Excalibur NIOS Development board [40], as seen in figure 7.10. This board contains an APEX EP20K200EFC484-2X FPGA chip which contains 8320 Lookup Tables (LUTs) – equivalent to approximately 200000 logic gates. The FPGA was programmed with a DES cracking design written in Verilog alongside of which, within the FPGA, was placed a 16-bit NIOS processor, which is an Altera developed RISC design which is easy to integrate with custom circuitry. The NIOS processor runs a simple program (written in GNU C and loaded into some local RAM on the FPGA) which looks after a serial link. The test vectors for the DES crack are loaded into the comparison memory via the serial link, and when cracking results are obtained they are returned over the same link. Although the NIOS could have been replaced by a purely hardware design, there was a considerable saving in complexity and development time by being able to use the pre-constructed building blocks of a processor, a UART and some interfacing PIOs.

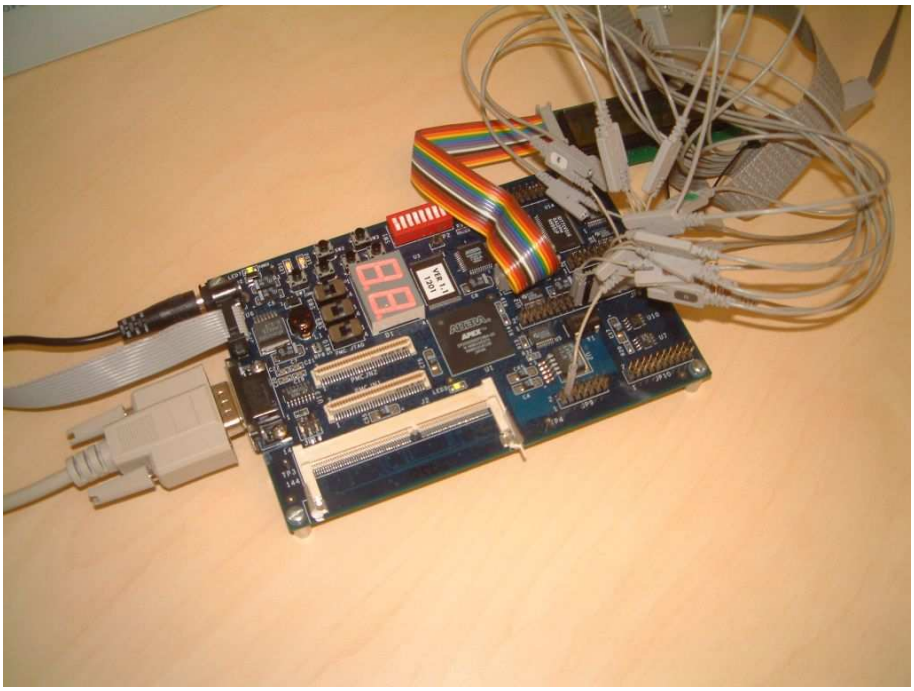


Figure 7.10: The NIOS Evaluation board running the DES cracker

The cracker can be seen in action on the logic analyser pictured in Fig. 7.11 below. The regular signal on the third trace is the clock. The second signal down shows a 32-bit match is occurring. This causes a STOP of the pipeline (top signal) and access to an odd numbered address value (bottom signal). The other signals are some of the data and address lines.

The full attack described in this paper was run on two occasions in 2001 at the full rate of 33.33 MHz (approx. 2^{25} keys/second). In both cases the expected running

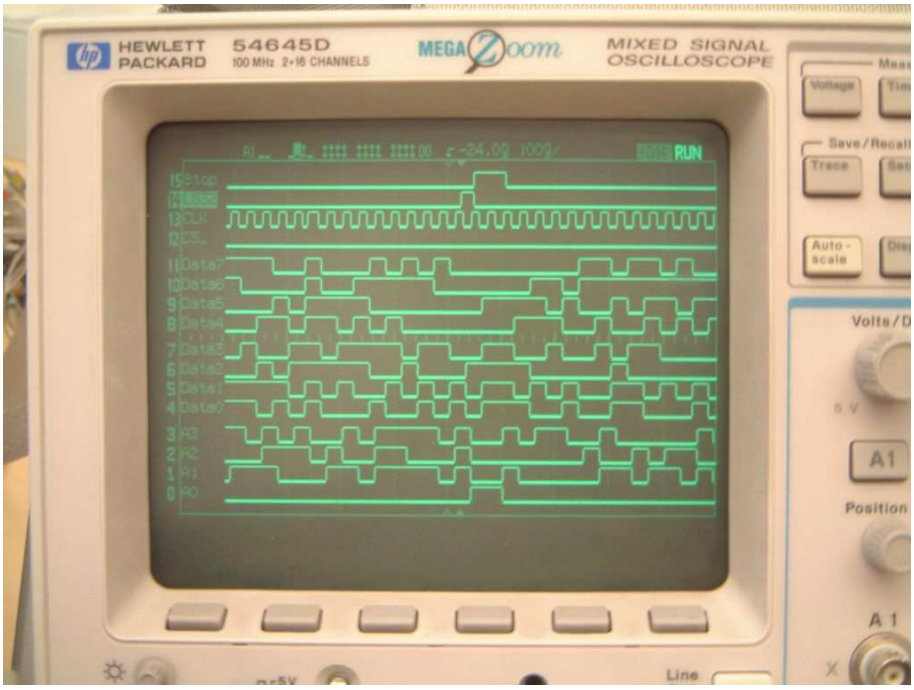


Figure 7.11: The DES cracker actually running

time of 50 hours (based on average luck in locating a key) was comfortably beaten and so it would have been possible to start using the PIN derivation keys well before unauthorised access to the 4758 could have been detected.

Date	Start	Finish	Duration	Key value found
Aug 31	19:35	17:47	22 h 12 min	`#3E0C7010C60C9EE8`
Sep 1	18:11	23:08	4 h 57 min	`#5E6696F6B4F28A3A`

Oct 9	17:01	11:13	19 h 12 min	`#3EEA4C4CC78A460E`
Oct 10	18:17	06:54	12 h 37 min	`#B357466EDF7C1C0B`

The results of this attack were communicated to IBM. In early November 2001 they issued a warning to CCA users [27] cautioning them against enabling various functionality that the attacks depended upon. In February 2002 they issued a new version of the CCA software [28] with some substantial amendments that addressed many issues raised by this attack.

Interestingly, the specification-level faults that were exploited in this attack have turned out to be just part of the story. Although much of the effort was devoted into reducing the effective strength of the CCA's 3DES implementation to that of single DES, IBM's analysis of the attack uncovered an implementation-level fault that made this whole stage unnecessary [29]. The CCA code was failing to prevent export of a double-length key under a double-length replicate key, despite the specifications stating that this would not be allowed.

7.3.8 4758 CCA – Weak Key Timing Attack

Bond & Clayton, 2001, Unpublished

The FIPS standards for DES advise the avoidance of using one of the 64 *weak DES keys*, and although IBM's CCA itself is not FIPS 140-1 validated, it observes precautions to avoid accidentally selecting one of these keys for a master key at random. The CCA master key is a three-key 3DES key, and it is checked by comparing each third with a table of weak keys stored in memory. The comparison is simply a call to the C `memcmp` command, and thus is a byte by byte comparison with the target data. The `memcmp` call will return when the first byte fails to match. There is thus an inherent timing characteristic created by the comparison, dependent upon the number of initial bytes of the master key which match weak key bytes.

The CCA firmware package that is loaded into the 4758 is only signed, not encrypted. It was disassembled and the comparison code located and confirmed to be susceptible. However, the task remained of performing an accurate timing measurement. The device driver DLL which governed interaction with the 4758 over the PCI bus was modified to sit in a tight loop waiting for a response over the bus, and count clock cycles. On a fast machine (the author used a 1.6GHz machine), this was easily enough accuracy to measure single instructions on the 100MHz 80486 within the 4758.

However, despite promising practical work, the attack remained theoretical, as the weak key testing was not done for normal key generation (only for master key generation which is a rare event, usually performed only when the 4758 is connected to a trusted host), and a large amount of noise was generated by the PCI bus buffering, which was never successfully modelled and compensated for. However, the principle of a timing attack on DES weak key checking remains, and it has recently become apparent that other FIPS approved devices are checking all DES keys generated against the weak key list in a similar manner. In particular, the Chrysalis Luna CA3 token seems to be vulnerable. It is hoped that this timing attack will be successfully implemented against a real device shortly. It will be ironic if it turns out that FIPS advice to avoid weak keys has (inadvertently) caused more severe attacks than the phenomenally rare pathological cases it protects against.

7.3.9 4758 CCA – Check Value Attack

Clulow, 2002, Unpublished

The 4758 CCA has a careless implementation fault in the `Key_Test` command. The command is a multi-purpose check value calculation command, which aims to be interoperable with equipment supporting all sorts of different check values of types and different lengths. It should be possible to calculate the check value of any key in the system – hence there are few restrictions on the possible control vectors supplied

to the command. It seems that the implementers recognised this, and decided that no control vector checking was necessary at all!

Whilst it is meaningful to calculate a check value for any type of key, it should not be possible to calculate check values for subcomponents of 3DES keys, nor present two key halves with completely different control vectors. One simple result is that the left half of a 3DES key can be supplied twice, and the check value retrieved as a test pattern on what is then effectively a single length DES key. The meet-in-the-middle attack can then be used to establish a known DES-key in the system. A normal run of the `Key_Test` command is shown followed by the attack in figure 7.12.

```
U -> C : { KL }Km/left , { KR }Km/right , left , right
C -> U : { 0000000000000000 }KL|KR

U -> C : { KL }Km/left , { KL }Km/left , left , left
C -> U : { 0000000000000000 }KL|KL
```

Figure 7.12: Normal and attack runs of Key Test

The resulting attack is thus a combination of an implementation level fault and a specification level fault (an API attack). Composite attacks of this nature are very hard to plan for and eliminate from designs.

7.3.10 VSM Compatibles – Decimalisation Table Attack

Bond & Zielinski, Clulow, 2002, [10], [15]

The decimalisation table attack affects financial Security APIs supporting IBM's PIN derivation method. It is a radical extension of a crude method of attack that was known about for some time, where a corrupt bank programmer writes a program that tries all possible PINs for a particular account. With average luck such an attack can discover a PIN with about 5000 transactions. A typical HSM can check maybe 60 trial PINs per second in addition to its normal load, thus a corrupt employee executing the program during a 30 minute lunch break could only make off with about 25 PINs.

The first ATMs to use decimalisation tables in their PIN generation method were IBM's 3624 series ATMs, introduced widely in the US in the late seventies. This method calculates the customer's original PIN by encrypting the account number printed on the front of the customer's card with a secret DES key called a *"PIN derivation key"*. The resulting ciphertext is converted into hexadecimal, and the first four digits taken. Each digit has a range of `'0'-'F'`. Hexadecimal PINs would have confused customers, as well as making keypads unnecessarily complex, so in order to convert this value into a decimal PIN , a "decimalisation table" is used,

which is a many-to-one mapping between hexadecimal digits and decimal digits. The left decimalisation table in figure 7.13 is typical.

0123456789ABCDEF	0123456789ABCDEF
0123456789012345	0000000100000000

Figure 7.13: Normal and attack decimalisation tables

This table is supplied unprotected as an input to PIN verification commands in many HSMs, so an arbitrary table can be provided along with the PAN and a trial PIN. But by manipulating the contents of the table it becomes possible to learn much more about the value of the PIN than simply excluding a single combination. For example, if the right hand table is used, a match with a trial pin of `0000` will confirm that the PIN does not contain the number `7`, thus eliminating over 10% of the possible combinations. This section first discusses methods of obtaining the necessary chosen encrypted PINs, then presents a simple scheme that can derive most PINs in around 24 guesses. Next it presents an adaptive scheme which maximises the amount of information learned from each guess, and takes an average of 15 guesses. Finally, a third scheme is presented which demonstrates that the attack is still viable even when the attacker cannot control the guess against which the PIN is matched.

Obtaining chosen encrypted trial PINs

Some financial APIs permit clear entry of trial PINs from the host software. For instance, this functionality may be required to input random PINs when generating PIN blocks for schemes that do not use decimalisation tables. The CCA has a command called `Clear_PIN_Encrypt`, which will prepare an `encrypted_PIN_block` from the chosen PIN. It should be noted that enabling this command carries other risks as well as permitting our attacks. If the PINs do not have randomised padding added before they are encrypted, an attacker could make a table of known trial encrypted PINs, compare each arriving encrypted PIN against this list, and thus easily determine its value. This is known as a *code book attack*. If it is still necessary to enable clear PIN entry in the absence of randomised padding, some systems can enforce that the clear PINs are only encrypted under a key for transit to another bank – in which case the attacker cannot use these guesses as inputs to the local verification command.

So, under the assumption that clear PIN entry is not available to the attacker, his second option is to enter the required PIN guesses at a genuine ATM, and intercept the `encrypted_PIN_block` corresponding to each guess as it arrives at the bank. Our adaptive decimalisation table attack only requires five different trial PINs – `0000` , `0001` ,`0010` , `0100` , `1000`. However the attacker might only be able to acquire encrypted PINs under a block format such as ISO-0, where the account

number is embedded within the block. This would require him to manually input the five trial PINs at an ATM for each account that could be attacked – a huge undertaking which totally defeats the strategy.

A third course of action for the attacker is to make use of the PIN offset capability to convert a single known PIN into the required guesses. This known PIN might be discovered by brute force guessing, or simply opening an account at that bank.

Despite all these options for obtaining encrypted trial PINs it might be argued that the decimalisation table attack is not exploitable unless it can be performed without a single known trial PIN. To address these concerns, a third algorithm was created, which is of equivalent speed to the others, and does not require any known or chosen trial PINs. This algorithm has no technical drawbacks – but it is slightly more complex to explain.

We now describe three implementations based upon this weakness. First, we present a 2-stage simple *static* scheme which needs only about 24 guesses on average. The shortcoming of this method is that it needs almost twice as many guesses in the worst case. We show how to overcome this difficulty by employing an adaptive approach and reduce the number of *necessary* guesses to 24. Finally, we present an algorithm which uses PIN offsets to deduce a PIN from a single correct encrypted guess, as is typically supplied by the customer from an ATM.

Initial Scheme

The initial scheme consists of two stages. The first stage determines which digits are present in the PIN. The second stage consists in trying all the possible PINs composed of those digits.

Let D_{orig} be the original decimalisation table. For a given digit i, consider a binary decimalisation table D_i with the following property. The table D_i has 1 at position x if and only if D_{orig} has the digit i at that position. In other words,

$$D_i[x] = \begin{array}{ll} 1 & \text{if } D_{\text{orig}}[x] = i, \\ 0 & \text{otherwise.} \end{array}$$

For example, for a standard table $D_{\text{orig}} = \texttt{0123 4567 8901 2345}$, the value of D_3 is $\texttt{0001 0000 0000 0100}$.

In the first phase, for each digit i, we check the original PIN against the decimalisation table D_i with a trial PIN of `0000`. It is easy to see that the test fails exactly when the original PIN contains the digit i. Thus, using only at most 10 guesses, we have determined all the digits that constitute the original PIN.

In the second stage we try every possible combination of those digits. The number of combinations depends on how many different digits the PIN contains. The table below gives the details:

Digits	Possibilities
A	AAAA(1)
AB	ABBB(4), AABB(6), AAAB(4)
ABC	AABC(12), ABBC(12), ABCC(12)
ABCD	ABCD(24)

The table shows that the second stage needs at most 36 guesses (when the original PIN contains 3 different digits), which gives 46 guesses in total. The expected number of guesses is about 23.5.

Adaptive Scheme

Given that the PIN verification command returns a single bit *yes/no* answer, it is logical to represent the process of cracking a PIN with a binary search tree. Each node v contains a guess, i.e., a decimalisation table D_v and a PIN p_v. We start at the root node and go down the tree along the path that is determined by the results of our guesses. Let p_{orig} be the original PIN. At each node, we check whether $D_v(p_{\text{orig}}) = p_v$. Then, we move to the right child if it is true and to the left child otherwise.

Each node v in the tree can be associated with a list $\mathcal{P}_v$ of original PINs such that $p \in \mathcal{P}_v$ if and only if v is reached in the process described in the previous paragraph if we take p as the original PIN. In particular, the list associated with the root node contains all possible pins and the list of each leaf contains only one element: an original PIN p_{orig}.

Consider the initial scheme described in the previous section as an example. To give a simplified example, imagine an original PIN consists of two *binary* digits and a correspondingly trivial decimalisation table, mapping $0 \to 0$ and $1 \to 1$. Figure 7.14 depicts the search tree for these settings.

The main drawback of the initial scheme is that the number of required guesses depends strongly on the original PIN p_{orig}. For example, the method needs only 9 guesses for $p_{\text{orig}} =$ 9999 (because after ascertaining that digit 0–8 do not occur in p_{orig} this is the only possibility), but there are cases where 46 guesses are required. As a result, the search tree is quite unbalanced and thus not optimal.

One method of producing a perfect search tree (i.e., the tree that requires the smallest possible number of guesses in the worst case) is to consider all possible search trees and choose the best one. This approach is, however, prohibitively inefficient because of its exponential time complexity with respect to the number of possible PINs and decimalisation tables.

It turns out that not much is lost when we replace the exhaustive search with a simple heuristics. We will choose the values of D_v and p_v for each node v in the following manner. Let $\mathcal{P}_v$ be the list associated with node v. Then, we look at all

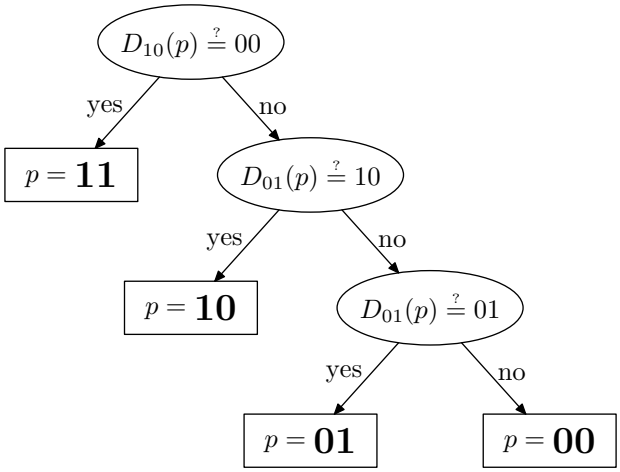


Figure 7.14: The search tree for the initial scheme. D_{xy} denotes the decimalisation table that maps $0 \to x$ and $1 \to y$.

possible pairs of D_v and p_v and pick the one for which the probability of $D_v(p) = p_v$ for $p \in \mathcal{P}_v$ is as close to $\frac{1}{2}$ as possible. This ensures that the left and right subtrees are approximately of the same size so the whole tree should be quite balanced.

This scheme can be further improved using the following observation. Recall that the original PIN p_{orig} is a 4-digit *hexadecimal* number. However, we do not need to determine it exactly; all we need is to learn the value of $p = D_{\text{orig}}(p_{\text{orig}})$. For example, we do not need to be able to distinguish between `012D` and `ABC3` because for both of them $p =$ `0123`. It can be easily shown that we can build the search tree that is based on the value of p instead of p_{orig} provided that the tables D_v do not distinguish between `0` and `A`, `1` and `B` and so on. In general, we require each D_v to satisfy the following property: for any pair of hexadecimal digits x, y: $D_{\text{orig}}[x] = D_{\text{orig}}[y]$ must imply $D_v[x] = D_v[y]$. This property is not difficult to satisfy and in reward we can reduce the number of possible PINs from $16^4 = 65\,536$ to $10^4 = 10\,000$. Figure 7.15 shows a sample run of the algorithm for the original PIN $p_{\text{orig}} =$ `3491`.

PIN Offset Adaptive Scheme

When the attacker does not know any encrypted trial PINs, and cannot encrypt his own guesses, he can still succeed by manipulating the offset parameter used to compensate for customer PIN change. The scheme has the same two stages as the initial scheme, so our first task is to determine the digits present in the PIN.

Assume that an encrypted PIN block containing the correct PIN for the account has been intercepted (the vast majority of arriving encrypted PIN blocks will satisfy this criterion), and for simplicity that the account holder has not changed his PIN so the correct offset is `0000`. Using the following set of decimalisation tables, the attacker can determine which digits are present in the correct PIN.

No	# Poss. pins	Decimalisation table D_v	Trial pin p_v	$D_v(p_{\text{orig}})$	$p_v \stackrel{?}{=} D_v(p_{\text{orig}})$
1	10000	1000 0010 0010 0000	0000	0000	yes
2	4096	0100 0000 0001 0000	0000	1000	no
3	1695	0111 1100 0001 1111	1111	1011	no
4	1326	0000 0001 0000 0000	0000	0000	yes
5	736	0000 0000 1000 0000	0000	0000	yes
6	302	0010 0000 0000 1000	0000	0000	yes
7	194	0001 0000 0000 0100	0000	0001	no
8	84	0000 1100 0000 0011	0000	0010	no
9	48	0000 1000 0000 0010	0000	0010	no
10	24	0100 0000 0001 0000	1000	1000	yes
11	6	0001 0000 0000 0100	0100	0001	no
12	4	0001 0000 0000 0100	0010	0001	no
13	2	0000 1000 0000 0010	0100	0010	no

Figure 7.15: Sample output from adaptive test program

Guess Offset	Guessed Decimalisation Table	Cust. Guess	Cust. Guess +Guess Offset	Decimalised Original PIN	Verify Result
0001	0123456799012345	1583	1584	1593	no
0010	0123456799012345	1583	1593	1593	yes
0100	0123456799012345	1583	1683	1593	no
1000	0123456799012345	1583	2583	1593	no

Figure 7.16: Example of using offsets to distinguish between digits

$$D_i[x] = \begin{array}{ll} D_{\text{orig}}[x] + 1 & \text{if } D_{\text{orig}}[x] = i, \\ D_{\text{orig}}[x] & \text{otherwise.} \end{array}$$

For example, for the table $D_{\text{orig}} =$ 0123 4567 8901 2345, the value of the table D_3 is 0124 4567 8901 2445. He supplies the correct encrypted PIN block and the correct offset each time.

As with the initial scheme, the second phase determines the positions of the digits present in the PIN, and is again dependent upon the number of repeated digits in the original PIN. Consider the common case where all the PIN digits are different, for example 1583. We can try to determine the position of the single 8 digit by applying an offset to different digits and checking for a match.

Each different guessed offset maps the customer's correct guess to a new PIN which may or may not match the original PIN after decimalisation with the modified table. This procedure is repeated until the position of all digits is known. Cases with all digits different will require at most 6 transactions to determine all the position data. Three different digits will need a maximum of 9 trials, two digits different 13 trials,

and if all the digits are the same no trials are required as there are no permutations. When the parts of the scheme are assembled, 16.5 guesses are required on average to determine a given PIN.

Results

We first tested the adaptive algorithm exhaustively on all possible PINs. The distribution in figure 7.17 was obtained. The worst case has been reduced from 45 guesses to 24 guesses, and the mode has fallen from 24 to 15 guesses. We then implemented the attacks on the CCA (version 2.41, for the IBM 4758), and successfully extracted PINs generated using the IBM 3624 method. The attack has also been checked against APIs for the Thales RG7000 and the HP-Atalla NSP.

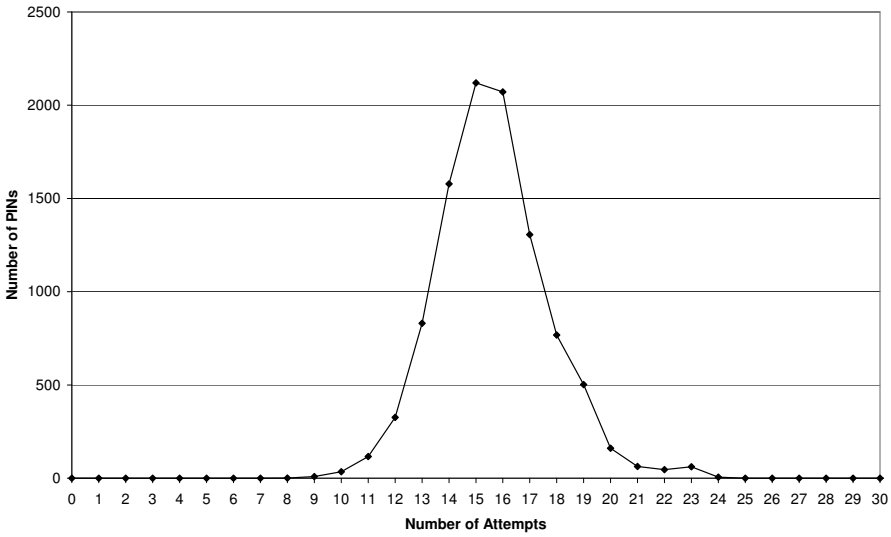


Figure 7.17: Distribution of guesses required using adaptive algorithm

Prevention

It is easy to perform a check upon the validity of the decimalisation table. Several PIN verification methods that use decimalisation tables require that the table be `0123456789012345` for the algorithm to function correctly, and in these cases enforcing this requirement will considerably improve security. However, the author has recently observed that skewed distribution of PINs will continue to cause a problem – see section 7.3.12 for details. Continuing to support proprietary decimalisation

tables in a generic way will be hard to do. A checking procedure that ensures a mapping of the input combinations to the maximum number of possible output combinations will protect against the naive implementation of the attack, but not against the variant which exploits PIN offsets and uses only minor modifications to the genuine decimalisation table. A better option is for the decimalisation table input to be cryptographically protected so that only authorised tables can be used.

The short-term alternative to the measures above is to use more advanced intrusion detection, and it seems that the long term message is clear: continuing to support decimalisation tables is not a robust approach to PIN verification. Unskewed random generation of PINs is really the only sensible approach.

7.3.11 Prism TSM200 – Master Key Attack

Bond, 2001, Unpublished

In the 1990s, South African HSM manufacturer Prism produced a version of its TSM200 with a transaction set specially customised for use in the prepayment electricity meter system developed by South African provider Eskom. Electricity meters included a tamper-resistant device which accepts "tokens" (simply 10 numeric digit strings) that increase its available credit. Tokens could be bought from a machine at a local newsagent: the function of Prism's module was to control the issue of tokens, and prevent the newsagent from cheating on the electricity supplier. If anyone could steal the keys for token manufacture, they could charge to dispense tokens themselves, or simply cause mischief by dispensing free electricity. The project is described more fully in [5].

A picture of the TSM200 is shown in section 6.3.6 of the Hardware Security Module chapter. The Prism API for the device is unusual compared with others, as it is non-monotonic (see section 7.1.4 for a discussion of monotonicity). The device uses only symmetric DES keys; these are stored within the device in 100 different registers available for the task. 3DES is supported by storing two keys in adjacent registers.

Communications & Communications Client

The author was given access to a TRSM for testing purposes, which by chance came with a live master key. This gave the opportunity for a useful challenge – could an API attack be performed to extract this key with only a single module, and with no second attempt should the attack fail and somehow corrupt the module state?

The HSM communicates with the outside world via a 'virtual serial port' – a single address mapped in memory through which commands can be sent and responses received. The transactions and data are encoded in ASCII so, given enough patience, a human can communicate directly with the module. Commands consisted of two letters indicating the general type of command, a question mark symbol indicating

it was a query, and then two letters for the command name. For example, the commands to initialise register 86 and put in the clear value of a key is as follows:

```
Security Officer 1 -> HSM  : SM?IK 86 08F8E3973E3BDF26
HSM -> Security Officer 1  : SM!IK 00 91BA78B3F2901201
```

The module acknowledges that command by repeating the name with the question mark replaced by an exclamation mark: `SM` stands for 'Security Module', and `IK` for 'Initialise Key (Component)'. The two digits following are a response code '`00`', indicating success, followed by the check value of the new contents of the register. Some commands such as the one above affect the state of internal registers, and due to dependencies between registers specified by the API, modification of one register could trigger erasure of others. A communications client was thus designed that had active and passive modes. Passive mode would only permit commands to be sent which did not affect the internal state of the module, active would allow all commands.

In order to safely develop macros that repeatedly executed commands constituting an attack, the communications client logged all commands and responses, and had an offline mode where the sequence of automatically generated commands could be inspected before actually risking sending them. These features were instrumental in developing the implementation of the attack without damaging the HSM internal state.

The Attack

An attack on the API was rapidly spotted after experimenting with the API. It exploited three building blocks from the attacker's toolkit – meet-in-the-middle attack on DES, a key binding failure, and poor check values – as well as a further API design error specific to the TSM200.

In order to ascertain which registers contain which keys, the API provides a `SM?XX` command, which returns a check value. The check value is the complete result of encrypting a string of binary zeroes with the DES key in that register. In fact, every command that modified the state of a register automatically returned a check value on the new contents of the register.

This 64-bit check value constituted a perfect test vector for a meet-in-the-middle attack. However, there were some problems: keys generated at random were not returned in encrypted form, but stored internally in a register. So if a large set of keys were to be generated, each had to be exported under some higher level key which could not be attacked, and it would be possible to discover the value of a key using the meet-in-the-middle attack that was marked internally as unknown. This was a definite flaw, but did not in itself constitute an attack on the deployed

```
Security Officer 1 -> HSM  : SM?IK 86 08F8E3973E3BDF26
HSM -> Security Officer 1  : SM!IK 00 91BA78B3F2901201

Security Officer 1 -> HSM  : SM?IK 87 E92F67BFEADF91D9
HSM -> Security Officer 1  : SM!IK 00 0D7604EBA10AC7F3


Security Officer 2 -> HSM  : SM?AK 86 FD29DA10029726DC
HSM -> Security Officer 2  : SM!AK 00 EDB2812D704CDC34

Security Officer 2 -> HSM  : SM?AK 87 48CCA975F4B2C8A5
HSM -> Security Officer 2  : SM!AK 00 0B52ED2705DDF0E4
```

Figure 7.18: Normal initialisation of master key by security officers.

system: all keys stored in a hierarchy recorded their parent key, and could only be re-exported under that key. Thus if a new exporter key were generated and discovered using this attack, it could not be used for exporting any of the existing target keys in the device. This design feature was particularly elegant.

However, the key part loading procedure offered scope for attack as well as random generation of keys. Figure 7.18 shows the normal use of the `SM?IK` and `SM?AK` commands by two security officers, who consecutively enter their key parts.

There was no flag to mark a key built from components as completed – any user could continue to XOR in keyparts with existing keys *ad infinitum*. So to perform a meet-in-the-middle attack, a related key set could be used, based around an existing unknown key, rather than generating a totally random set of keys. There was just one key created from components under which the target keys were stored – the master key (by convention kept in registers 86 and 87). The key binding failure then came into play: the check values were returned on each half of the 3DES key independently, so this meant that it could be attacked with only twice the effort of a DES key (i.e. two meet-in-the-middle searches).

A final crucial hurdle remained – changing the master key caused a rippling erasure of all child keys, thereby destroying the token generation keys which were the ultimate target. Fortunately the answer was already there in the earlier reasoning about attack strategy – export all keys in the hierarchy before attacking the master key, and re-import them once it had been discovered.

Completing the Attack

The test vectors were harvested using a small loop operation, which XORed a new constant in with the master key half each time, and then recorded the check value returned. At the end of the loop the master key was restored to its original value.

```
For I= 0000000000000001 to 000000000001FFFF
    {
    SM?AK 87 (I xor (I-1))
    SM!AK 00 (result)
    store the pair (I, result)
    }
```

Finally, the key hierarchy exported before the master key was attacked was decrypted offline using a home PC. The author successfully implemented the attack as described in 2001; Prism was informed and they later modified their API to limit the number of components which could be combined into a register.

7.3.12 Other Attacks

Attacks described elsewhere

- Dual Security Officer Attack – see section 8.4.1
- M-of-N Security Officer Attack – see section 8.4.2

Recent Attacks not fully described

These new attacks have been discovered so recently that they cannot be fully incorporated in this thesis. Brief summaries suitable for those familiar with financial Security APIs have been included; academic publication is pending.

- **PVV Clash Attack** – VISA PVV values are calculated by encrypting the transaction security value, and then truncating and decimalising the result. There is a good probability that several different transaction security values will produce the same PVV as a result – thus there may be several PINs that could be entered at an ATM that will be authorised correctly. An insider could use PVV generation transactions to find the rarer accounts which may have ten or more correct PINS.
- **ISO-0 Collision Attack** – Some PIN generation commands return the generated PIN as an encrypted ISO-0 PIN block, in order to send off to mass PIN mailer printing sites. By using these generation commands and calculating all the PINs for the same account by stepping through the offsets, one can build up a full set of encrypted PIN blocks for a particular PAN. These blocks could alternatively be generated by either repeatedly calling a random PIN generate function (as with PVV) until by luck all values get observed. All the attacker can see are the encrypted PIN blocks, and he cannot see what order they are

in. Consider the example below, which uses 1 digit PINS and PANs, and 4 digit encrypted PIN blocks.

The attacker observes that encblock `AC42` from the left hand list does not occur in the right hand list, and likewise for encblock `9A91`. Therefore he knows that the PIN corresponding to `AC42` is either 8 or 9 (and that the PIN corresponding to `9A91` is either 8 or 9). The attack can be built up to reveal two digits of the PIN, as with Clulow's PAN modification attack [15].

PAN	PIN	PAN⊕PIN	encblock	PAN	PIN	PAN⊕PIN	encblock
7	0	7	`2F2C`	0	0	0	`21A0`
7	1	6	`345A`	0	1	1	`73D2`
7	2	5	`0321`	0	2	2	`536A`
7	3	4	`FF3A`	0	3	3	`FA2A`
7	4	3	`FA2A`	0	4	4	`FF3A`
7	5	2	`536A`	0	5	5	`0321`
7	6	1	`73D2`	0	6	6	`345A`
7	7	0	`21A0`	0	7	7	`2F2C`
7	8	F	`AC42`	0	8	8	`4D0D`
7	9	E	`9A91`	0	9	9	`21CC`

- **ISO-0 Dectab PIN Derivation Attack** – Imagine a financial HSM command `Encrypted_PIN_Generate`, which derives a PIN from a PAN, adds an initial offset, then stores it as an ISO-0 PIN block. It has a decimalisation table hardwired into the command that cannot be altered.

  1. By looping through the offset value you can discover all 10000 possible encrypted PIN blocks for that account, but you don't know which are which.
  2. The classic way to proceed is to make a genuine guess at an ATM, and try and catch the encrypted PIN block as it arrives for verification. This should give you a start point into the loop, which you can use to calculate the correct PIN. However, it is ugly – it requires one trip to a real ATM per account attacked.
  3. Instead, conjure many different PIN derivation keys, and use each to derive a 'correct' PIN from the PAN of the target account. Keep the offset fixed at `0000`. The derived PINs generated under different PIN derivation keys will be biased in accordance with the (fixed) decimalisation table.
  4. This creates a unique distribution of frequency of occurrence of encrypted PIN blocks outputed by the command. This distribution (combined with a loop through offsets under a single key) allows you to synchronise the loop of encrypted PIN block values with the loop of real PINs.

5. The estimated transaction cost is 10000 for the loop, and maybe 2,000–10,000 data samples to determine the distribution. With modern transaction rates this equates to about 30 seconds per PIN. The attack should work on any financial HSM where you can conjure keys (or that has unrestricted generation facilities).

7.4 Formal Analysis of Security APIs

7.4.1 Foundations of Formal Analysis

This thesis constitutes the first comprehensive *academic* study of Security APIs. Though they have existed for several decades, their design and development has been the preserve of industry. One might expect the formal methods community to have already embraced the study of Security APIs as a natural extension of protocol analysis. This has not been the case, due in part to restricted circulation of API specifications, but also due to the intrinsic nature of APIs themselves. Security API use sufficiently specialist cryptographic primitives central to functionality to put Security API design a distance away from O/S design (and the corresponding "program proving" formal methods camp), and much closer to cryptographic protocol analysis.

Unfortunately, the security protocols analysis camp seems reluctant to take on board and interest themselves in problems with any degree of functional complexity – that is, problems which cannot be expressed concisely. The only formal analysis previously made of a Security API is in the 1992 paper "An Automatic Search for Security Flaws in Key Management Schemes" [32], which describes the use of a search tool employing specially designed heuristics to try to find sequences of commands which will reveal keys intended to remain secret. The paper describes the search tool and heuristics in some detail, but shies away from describing the API itself, stating only that the work was done in cooperation with an unnamed industry partner.

In comparison, for example, with an authentication protocol, a Security API is several orders of magnitude more complex to understand, not in terms of subtleties, but in the multitude of commands each of which must be understood. It may take weeks, not days, of studying the documentation until a coherent mental picture of the API can be held in the analyst's head. In addition to the semantics of the transaction set, the purpose of the API must be understood – the policy which it is trying to enforce. For some examples such as PIN processing, the main elements of the policy are obvious, but for more sophisticated key management operations, it may require some thought to decide whether an the weakness is actually a breach of policy.

Indeed it now seems that many conventional real-world protocols are becoming less attractive targets for analysis, as they pick up further functional complexity, backwards compatibility issues, and suffer the inevitable bloat of committee design. The analysis of the Secure Electronic Transaction (SET) protocol made by Larry Paulson [7] gives an idea of the determination required to succeed simply in formalising the protocol.

There is thus little past work to build upon which comes directly under the heading of Security APIs. In light of this, the formal analysis in this thesis builds on that of security protocols, which is a well established area of work with hundreds of papers published. The APIs analysed are specifically concerned with financial PIN processing, due in no small part to its simple security policy – "the PIN corresponding to a customer's account must only be released to that customer".

So, under what circumstances can the literature and tools for protocol analysis be applied to Security APIs?

General-purpose tools from the program correctness camp such as theorem provers and model checkers which have been applied to security protocols with success might also be applied to Security APIs, as they were design to be general purpose in the first place. However, there is no guarantee that the heuristics and optimisations developed for these tools will be well-suited to Security API analysis.

There are obvious similarities between a Security API and a security protocol. The user and the HSM can be considered principals, and the primitives used for constructing messages – encryption, decryption, concatenation are very similar. In both, the messages consist of the same sorts of data: nonces, identifiers, timestamps, key material, and so on.

However, the differences are significant too. Firstly, an API is a dumb adversary. When a security protocol is run on behalf of a human – Alice or Bob – it is often assumed that deviations or inconsistencies in the execution of the protocol can be effectively reported and that the human can react when their protocol comes under attack. Todays APIs do not interact in this way with their owners, and will stand idly by whilst large quantities of malformed and malicious commands are sent to them. Secondly, APIs are qualitatively larger than security protocols. There are several orders of magnitude more messages than in an authentication protocol, and the messages themselves are larger, even though they are made from very similar building blocks.

APIs are simpler than security protocols in one area: there are usually only two principals – HSM and User. This eliminates the need for reasoning about multiple instances of protocols with multiple honest and dishonest parties, and the different interleavings of the protocol runs. Unfortunately, the effort put into reasoning about such things in the better-developed protocol analysis tools cannot be put to a useful purpose.

7.4.2 Tools Summary

There are over a dozen formal analysis tools available to the public which could be applied to Security APIs. Most are the product of academic research programmes and are available for free, while several are commercial products (for example, FDR [43]). In the context of gaining assurance about Security APIs, all formal tools do essentially the same thing – they search. There are three broad categories of tool, based on three different technologies: theorem provers, model checkers, and search tools themselves. Figure 7.19 lists some common tools.

- *Theorem Provers* search for a chain of logic which embodies all possible cases of a problem and demonstrates that a theory holds true for each case. In the best case they find an elegant mathematical abstraction which presents a convincing argument of the truth of the theory over all cases within a few lines of text. In the worst case, they enumerate each case, and demonstrate the truth of the theory for it.

  A special category of theorem provers exist – *resolution* theorem provers. *Resolution* is a method of generating a total order over all chains of logic that might constitute a proof, devised by Robinson in 1965 [36]. It permits a tool to proceed through the chains of logic in a methodical order that inexorably leads towards finding of the correct chain, or deducing that there is no correct chain of reasoning. Resolution initially enjoyed some success in finding proofs for theorems that had eluded other techniques, but this was largely due to the fact that the transformation of the proof space was difficult for humans to visualise, so it took a while to understand what problems resolution performed poorly at, and how to design classes of pathological cases. Eventually it became clear that the class of problems resolution performed well at was simply different from that of other provers, and not necessarily larger. It remains as an underlying mechanism for some modern theorem provers such as SPASS (see section 7.4.3) but is not nearly as popular as in its heyday in the 70s.

- *Model Checkers* also search – they explore the state space of the system which is specified as the problem, evaluating the truth of various conditions for each state. They continue to explore the state space hoping to exhaust it, or find a state where the conditions do not hold. Some model checkers use mathematical abstractions to reason about entire sets or branches of the state space simultaneously, or even apply small theorems to deduce that the conditions tested must hold over a certain portion of the space. In theory model checkers will examine the entire state space and can give the same assurance of correctness as a theorem prover, though in practice many set problems that the model checker cannot complete, or deliberately simplify their problem into one which can be fully examined by the model checker.

- *Search Tools* – such as PROLOG – most openly admit that at the heart of formal analysis is methodical search. These tools provide specification languages for expressing problems that make them amenable to breadth-first or depth first-search, and then search away, looking for a result which satisfies some end conditions. The searches are often not expected to complete.

Theorem Provers	Model Checkers	Search Tools
Isabelle	Spin	Prolog
SPASS	SMV	NRL Analyser
Otter	FDR	

Figure 7.19: Well known formal analysis tools

So at heart, all the tools *do* the same thing. For those simply searching for faults, the state-of-the-art tool that will perform best on their problem could lie in any of the three categories. However, for those concerned with assurance of correctness, there is an extra axis of comparison between the tools – **rigour**. Some formal tools are designed with an overriding goal that any answer that they produce is truly correct; that no special cases or peculiar conditions are missed by the tool (or any of its optimisations) that might affect the validity of the answer. The most visible affect of this design philosophy is in the preciseness and pedanticism of the specification language that the tool accepts. It is often this language – the API for the tool – which is the most important component of all.

7.4.3 Case Study: SPASS

SPASS [52] is a FOL (First Order Logic) theorem prover. It tries to automatically construct a proof of a theory by applying axioms presented in the user's problem specification, and inbuilt axioms of logic. The chain of reasoning produced will normally be much more detailed than that which would be necessary to convince a human of the truth of a theory, and will require a degree of translation to be human-readable.

SPASS was recommended as an appropriate and powerful tool which could reason about monotonically increasing knowledge, and solve reachability problems in a set of knowledge. In order to prove the existence of an attack on an API, SPASS had to demonstrate a sequence of commands which would reveal a piece of knowledge which was supposed to remain secret. The author represented commands in the API as axioms stating that if certain inputs were 'public' (i.e. known to the user of the device, and thus an attacker), then some manipulation of the inputs (i.e. the output) would be public also. Variables were used in the axioms to range over possible inputs. The simple example below shows a hypothetical command which takes an input `X`, and produces an output of `X` encrypted with key `km`.

```
formula( forall([X,Y,Z],
     implies( and(public(X),and(public(Y),public(Z))) ,
                  public(enc(enc(i(wk),Z),enc(i(enc(i(tmk),Y)),X))) )
       ),tran_tmki_to_wki).

formula( forall([X,Y,Z],
     implies( and(public(X),and(public(Y),public(Z))) ,
                  public(enc(enc(i(wk),Z),enc(i(enc(i(wk),Y)),X))) )
       ),tran_cc).
```

Figure 7.20: Sample SPASS code encoding several VSM commands

```
public( X ) -> public( enc(km,X) )
```

SPASS lacks infix notation, so the above command would be written in the problem specification as follows:

```
formula(forall([X],implies(public(X),public(enc(km,X))))).
```

Representations of API commands from models of the CCA and VSM APIs are shown in figures 7.21 and 7.20.

SPASS performed very well at reasoning about very simple API attacks – manipulating terms according to its inbuilt hunches, and could demonstrate for instance the 'XOR to Null Key' and 'Key Separation' attacks on the Visa Security Module (sections 7.3.1 and 7.3.2). Modelling of more complex multi-command attacks on the CCA was more problematic. In particular, one type-casting attack (see section 7.3.4) consisting of a sequence of three commands could not be proved to work even given several CPU days of execution time. If the the attack was artificially shortened by setting an easier goal – the output of the second of the three commands – SPASS would almost immediately be able to confirm that this goal was attainable. Likewise, by providing additional 'initial knowledge' equivalent to having correctly chosen the first command in the sequence of the attack, SPASS would conclude in less than a second that the final goal was attainable. The full sequence of three commands seemed to have defeated its reasoning, and there was no way to tell how or why.

The way in which SPASS reasoned, though highly developed and the topic of research for some years by the development team at MPI in Saarbruken, remained a mystery. The documentation provided with SPASS constitutes a brief HOWTO guide, a list of command line switches, and a elaborate document testifying to the rigour of the proof method used by SPASS [39]. None of this gave much illumination to the understanding of the circumstances in which SPASS would be likely to

```
\% these are the commands provided by the 4758 CCA

\% Encrypt Command
\% W is an encrypted token containing the key
\% X is the data
formula(forall([W,X],implies( and(public(W),public(X)) ,
        public(enc(enc(inv(xor(data,km)),W),X))
                )),Cmd_Encrypt).

\% Key Import Command
formula(forall([KEK,TYPE,KEY],implies(

and( public(TYPE) , and( public(enc(xor(KEK,TYPE),KEY)) ,
                         public(enc(xor(km,imp),KEK)) )) ,
                \%   ====>
                         public(enc(xor(km,TYPE),KEY))

        )),Cmd_Key_Import).

\% Key Part Import Command
\% W is input token
\% X is cv base
\% Y is clear xor value
formula(forall([W,X,Y],implies( and(public(kp(X)),
                                and(public(Y),
                                    public(enc(xor(km,kp(X)),W)))),
                \%    ======>
        public( enc( xor(km,X) , xor( W , Y ) ) ) )
                  ),Cmd_Key_Part_Import).
```

Figure 7.21: Sample SPASS code encoding several CCA commands

perform well, nor how to represent problems in a way most amenable to analysis in the extremely rich specification language of first order logic. The output from SPASS was similarly mysterious, a sample of which is shown in figure 7.22.

The biggest unanswered question remained unaddressed: how could the progress of the tool itself whilst be monitored whilst it was running? The release version simply sits there and thinks. It says 'PING' once every few seconds, to assure the user that it is still alive, but apart from that there is no user feedback as to how the search for a proof of the theory is proceeding.

It became clear that whilst it was appealing to harness decades of research into search methods, optimisations, and data structures for representing state, doing this by picking an off-the-shelf tool would be a hit-and-miss affair, and would never be unlikely to yield interesting new results about Security APIs, unless given several years of research.

```
-------------------------SPASS-START-----------------------------
 PING PING PING PING PING PING PING PING
SPASS V 1.0.0
SPASS beiseite: Proof found.
Problem: d:\mkb23\proj\formal\4758-cmd3c.spass
SPASS derived 2987 clauses, backtracked 0 clauses and kept 622 clauses.
SPASS allocated 2168 KBytes.
Here is a proof with depth 6, length 38 :

2[0:Inp] ||  -> public(k3)*.
5[0:Inp] ||  -> public(data)*.
6[0:Inp] ||  -> public(imp)*.
7[0:Inp] ||  -> public(xor(pin,data))*.
8[0:Inp] ||  -> equal(xor(U,z),U)**.
9[0:Inp] ||  -> equal(xor(U,U),z)**.
10[0:Inp] ||  -> public(enc(xor(kek,pin),pp))*.
11[0:Inp]public(U) ||  -> public(kp(U))*.
12[0:Inp] ||  -> equal(xor(U,V),xor(V,U))*.
13[0:Inp] || public(enc(xor(km,data),pp))*+ -> .
15[0:Inp] ||  -> public(enc(xor(km,kp(imp)),xor(kek,k3)))*.
17[0:Inp]public(U) public(V) ||  -> public(xor(V,U))*.
18[0:Inp] ||  -> equal(xor(xor(U,V),W),xor(U,xor(V,W)))**.
21[0:Inp]public(U) || public(kp(V)) public(enc(xor(km,kp(V)),W))+
-> public(enc(xor(km,V),xor(W,U)))*.
22[0:Inp]public(U) || public(enc(xor(km,imp),V))*+
public(enc(xor(V,U),W))* -> public(enc(xor(km,U),W))*.
23[0:Rew:12.0,15.0] ||  -> public(enc(xor(km,kp(imp)),xor(k3,kek)))*.
24[0:Res:22.3,13.0]public(data) || public(enc(xor(km,imp),U))*
public(enc(xor(U,data),pp))* -> .
26[0:ClR:24.0,5.0] || public(enc(xor(km,imp),U))*+
public(enc(xor(U,data),pp))* -> .
31[0:SpR:12.0,8.0] ||  -> equal(xor(z,U),U)**.
78[0:SpR:18.0,12.0] ||  -> equal(xor(U,xor(V,W)),xor(W,xor(U,V)))*.
-------------------------SPASS-STOP------------------------------
```

Figure 7.22: Sample output from SPASS (edited to fit on one page)

7.4.4 MIMsearch

The MIMsearch tool is a distributed search tool developed by the author as part of this thesis, designed for exploring sequences of API commands to determine if they violate security assertions about an API. It was created as an experiment rather than as a potential rival to the other formal tools; its specific goals were as follows:

- The primary goal was to learn about the strengths and weaknesses of model checkers (and theorem provers) through comparison with a well understood example;
- A secondary goal was to improve the author's ability to use the existing tools, through better understanding of their internal working
- A third goal was to develop a tool which allowed reasonable estimates of the complexity of models of APIs to be made, to get an idea of the bounds on complexity of API attacks which are already known
- The final goal was functional: to try to create a tool which was powerful enough to reason about existing financial APIs, in particular those using XOR.

MIMsearch works by manipulating trees of terms representing functions and atoms. A command is executed by substituting the arguments into variables within a larger term, and then simplifying the term. It has native support for encryption, and crucially, for reasoning about the XOR function. It has a sophisticated suite of readouts to allow an observer to see the progress of a search for an attack, and compare this progress against predefined subgoals when searching for a known attack.

Heuristics

Most existing tools have an array of heuristics which are applied to control the direction of the search, and to try to produce an answer as quickly as possible. Whilst these are indeed useful when they solve a problem rapidly, they hinder attempts to measure problem difficulty by seeing how long a search tool takes to solve it. As one of the goals of the MIMsearch tool was to gain a greater understanding of problem complexity, as few heuristics as possible were used.

The only heuristic available in the current implementation is *'likely reduction filtering'*. This technique filters out a subset of possible terms that could be substituted in as an argument into one of the terms representing a transaction. The filters are provided by the human operator along with the problem specification, and are conventional terms with wildcards to specify ranges of structures. The reasoning behind the heuristic is that substituting in a phrase which does not enable execution of a command to perform any simplification after all the arguments have been substituted is not likely to correspond to a meaningful step of any attack. Whilst this heuristic sounds pleasing, there can be no proof that it is true in all cases.

Problem Specification

The API is specified to MIMsearch as a series of terms containing variables representing the arguments. The example below shows a specification for the CCA `Encrypt` command. The `Input` lines are terms describing likely-reduction filters. The `Output` line describes the function of the command; `ZERO` and `ONE` are variables where the first and second arguments are substituted in (in this example `KM` and `CV_DATA` are atoms specific to this API).

```
Cmd ‘‘Encrypt’’
Input ENC(XOR(KM,CV_DATA),ANY)
Input ANY
Output ENC(DEC(XOR(KM,CV_DATA),ZERO),ONE)
End_Cmd
```

After the API specification, the conditions for a successful attack are specified with a set of “initial knowledge” – terms which are publicly available and thus known to the attacker from the beginning. Finally, there are goals – terms which if proved to be made public by some sequence of API commands will constitute a breach of security. The classic initial knowledge includes one instance of every type of key normally available in the device, in particular, a live PIN derivation key encrypted under the relevant master key, and a PAN. The typical goal is to discover `{ PAN1 }PDK1` – a particular PAN encrypted under the live PIN derivation key.

Architecture

The idea at the heart of MIMsearch is “meet-in-the-middle” – searches are performed both forwards from the initial knowledge and backwards from the final goal, and the resulting terms and goals stored in hash tables. The tool constantly looks for collisions between entries of one hash table and the other, effectively square-rooting the complexity of the search in the optimum case.

The search proceeds in a depth-first manner, with separate but interleaved threads of the program searching forwards and backwards. The total search depth is the sum of the forward and backward depths. For each layer of the the search, the forward searching thread first selects a command from the API specification, then randomly selects initial knowledge to substitute in as arguments to that command. Each new term produced is added to the initial knowledge, hashed using collision-resistant hash function, and then used to set a single bit in a knowledge hash table to represent its discovery. The hash is also looked up in the goal hash table, and if the corresponding bit in the goal table is set, an attack has been found (provided that it is not a false collision). Once the maximum depth is reached and the final term has been hashed and added to the knowledge table, the initial knowledge is

reset to that of the problem specification – the only recording of the path searched are the entries in the knowledge and goal hash tables.

MIMsearch is unlike other tools that often continue to expand the knowledge set, storing each term in a usable manner. This prevents wasted effort repeatedly deriving the knowledge of the same term again, but does not actually make for a more balanced search, as it does not give any clue as to how to weight the probabilities for selection of these terms as inputs for the next search. The MIMsearch approach is a simple one: pick randomly and apply no heuristics to weight term selection. In order to tackle significant size problems using this approach, a lot of brute force is required.

Implementation

MIMsearch is written in C++ and Visual Basic, and comprises about 150 kilobytes of source code. As it is intended to operate as a distributed system on a tightly knit cluster of PCs, multiple programs are required. The main program can be activated in three roles – launcher, searcher, and central node. The task of the launcher is to receive the latest version of the searcher via a TCP stream from the central node, and then set it running when requested. The searcher actually performs the search, and communicates statistics and results back to the central node using TCP. The central node collates the statistics, and then feeds to a GUI written in Visual Basic which displays them in both graphical and textual forms.

Communication between searchers and mission control was implemented from scratch on top of the standard Winsock API, for reasons of customisability and in mind of future concerns about efficiency. There were a number of freely available distributed systems packages for managing communication between nodes, but all suffered from either unnecessary complexity in terms of simply providing a link for communicating statistics, or from potential poor efficiency and difficulty of customisation in the context of enabling communication between nodes for hash table sharing.

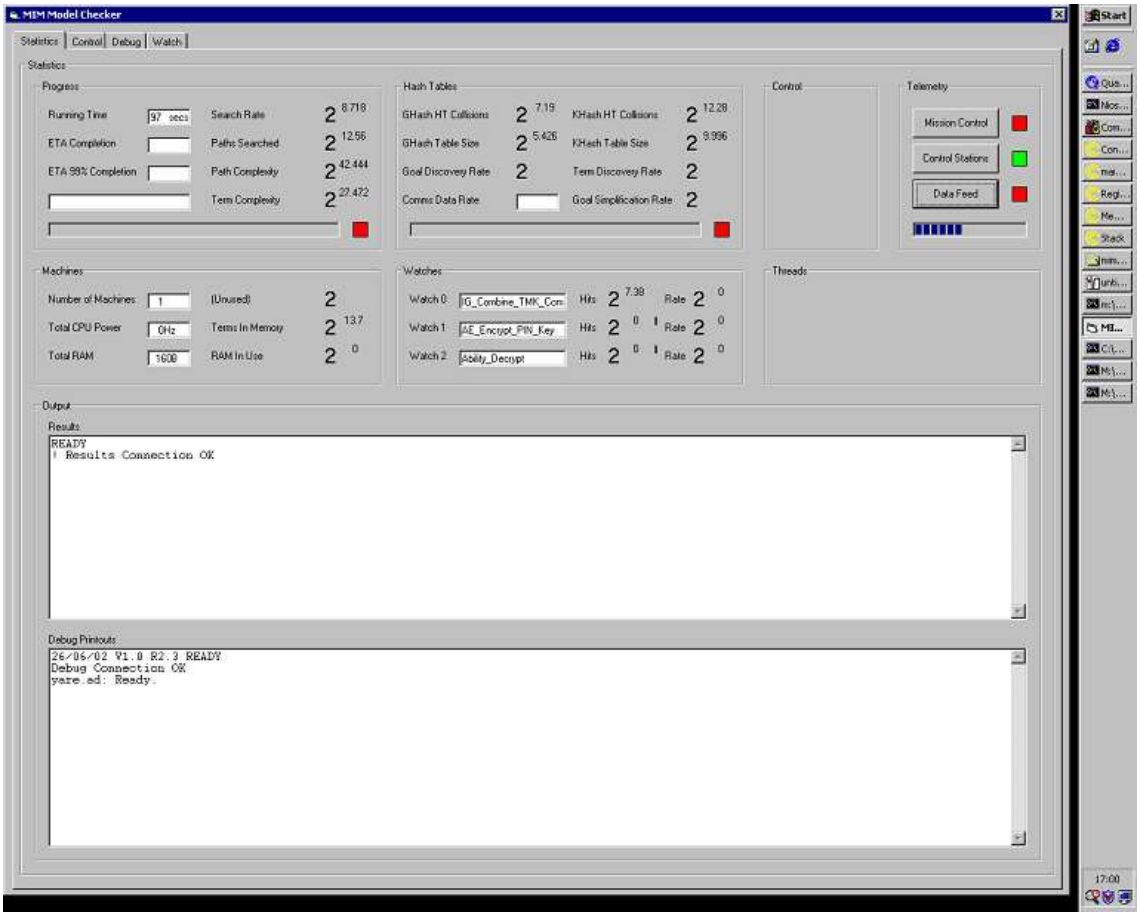


Figure 7.23: The MIMsearch statistics display

UI Decisions

Design of the user interface was in some senses the most important part of the project, as detailed feedback was found lacking from the other tools, and was the key to gaining greater understanding of both problem and tool. The GUI comprises of a statistics screen for the problem (figure 7.4.4), a control screen which monitors the status and correct operation of the search nodes (figure 7.4.4), a watch screen to monitor progress of the search against user defined goals (figure 7.4.4), and an interactive command line interface for specific queries.

The main statistics screen shows most figures in powers of two, displaying the rate of search and monitoring the filling of the knowledge and goal hash tables (figure 7.4.4). For searches lasting more than several hours, this data serves just to assure the user that the search is still in progress and that none of the search nodes has crashed. There are also output screens displaying possible results from the search (when using small hash tables this screen will display some false matches).

The control screen shows the status of the nodes, and gives a *"complexity report"* of the problem in question (figure 7.4.4). This report gives an upper bound upon the

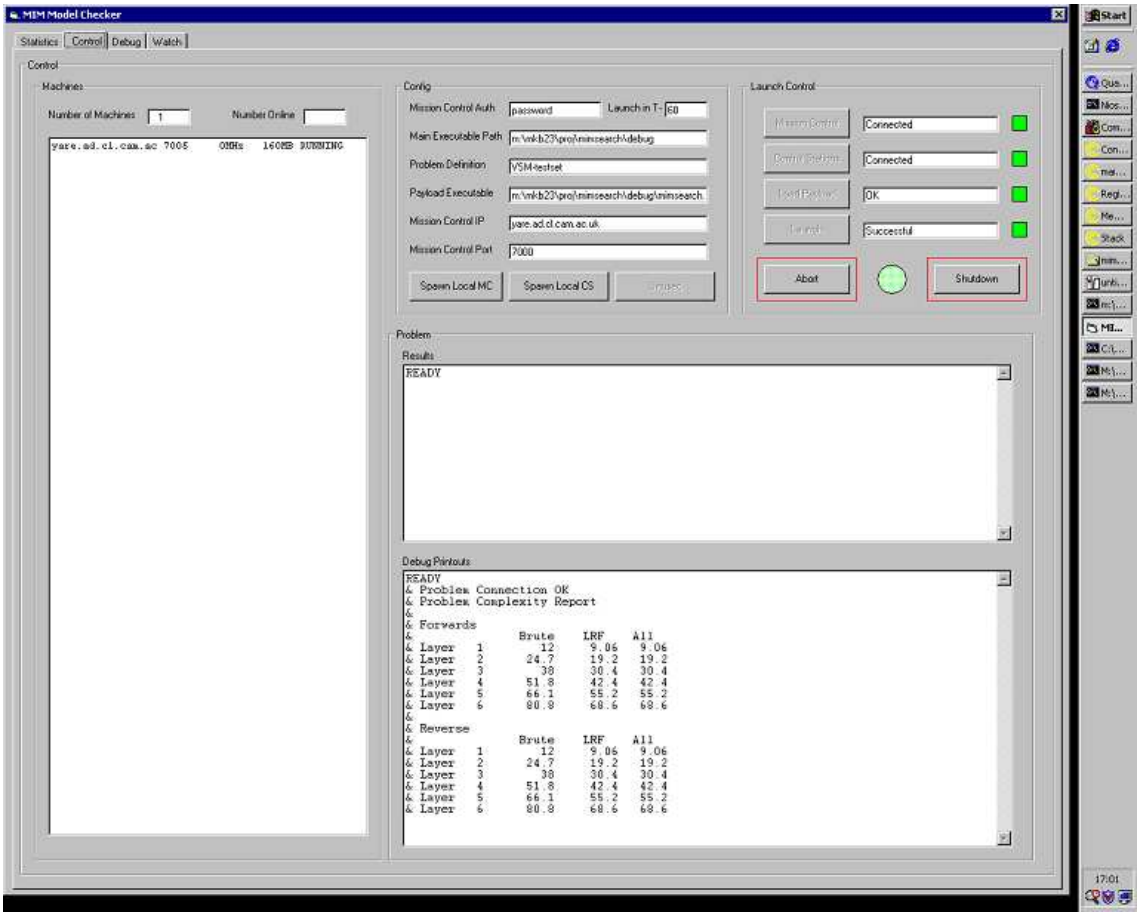


Figure 7.24: The MIMsearch control interface

size of search required to explore all sequences of commands up to a certain depth.

The most detailed statistics are shown on the watch display (figure 7.4.4). Each watch entry represents a command in a user-defined sequence that represents the attack the tool is searching to find. For each command, the display shows the number of times the correct term as been chosen for each of its input, and the number of times the correct output has been produced (i.e. when every input is chosen correctly simultaneously). The rates at which correct inputs are chosen per second is also shown. On the right hand side, two columns display the status of these terms with respect to the hash table. As output terms are produced correctly for the first time, they are entered into the hash table, and this is denoted by a '1'. This watch display makes it possible to observe whether or not all the individual preconditions for a command in the sequence of the attack are occurring, and observe their rates. It can then be easily seen whether it will just be a matter of time waiting for these conditions to coincide simultaneously by luck, or whether the tool is incapable for some reason of finding the attack sequence.

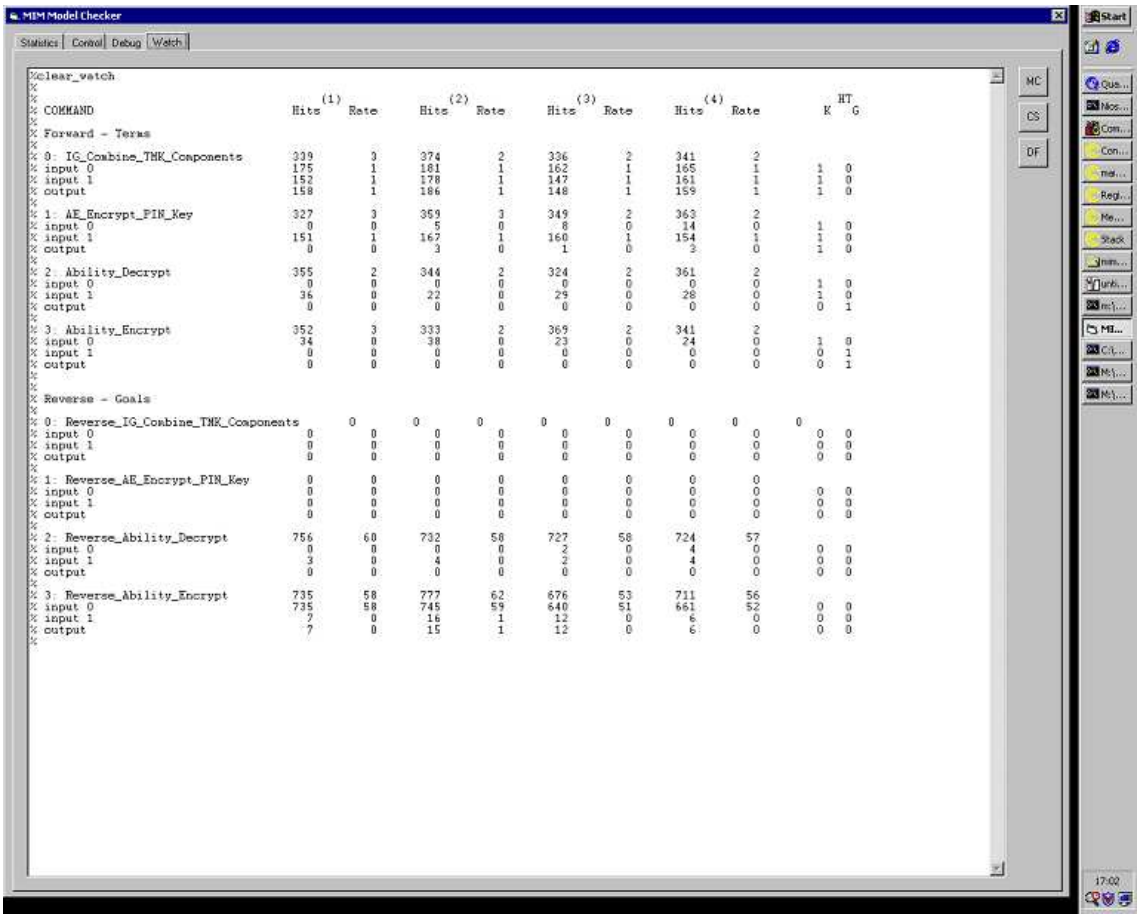


Figure 7.25: The MIMsearch ‘Watch’ display

Chapter 8

Designing Security APIs

This chapter discusses the design of Security APIs. It first discusses the circumstances under which a Security API might be useful, then discusses heuristics for good design. It goes on to discuss the issues of *authorisation* and *trusted path*, as well as *procedural control design* and *social engineering* issues.

8.1 Can Security APIs Solve Your Problem?

How do you decide whether or not you need a Security API, with or without an accompanying tamper-resistant HSM embodying it, to solve your problem? And how can you decide whether or not your problem *can be solved* by a properly designed Security API?

A good Security API design will do a mixture of two things:

1. Take a policy that you don't know how to economically enforce, and translate it into multiple well understood policies which are easy (or, at least, obvious) to enforce.

2. Take pieces of a policy that are rigid and easy to enforce, and deal with them

Suppose you have the problem that an employee has the potential to damage your company through abusive manipulation of an API that he operates every day. The Security API will be able to monitor and control this person, but when it translates the broad 'prevent abuse' policy into smaller policies, these policies must be ones that you can easily enforce. Consider three possible designs for the API:

1. If the Security API performs an audit function, watching the stream of commands from the user, and logging them for later examination, this will only solve your problem if you have attentive and capable auditors to watch the trail.

2. If the Security API prevents certain types of commands from being issued without supervisor approval, you must have well-trained and trustworthy supervisors to give the approval.

3. If the Security API recognises all the bad sequences of commands and can prevent them, you implement it, and you need do nothing more. However, the API designers must have fully understood the full range of abuse possible.

In the first two cases, the API serves a simple but very useful purpose – it pushes the problem elsewhere– to another service. For your problem to be solved, these destination services must either exist, or be obvious how to develop. In the final case, the API tries to actually solve the problem itself– preventing known abuses (or, in the limit, designing a system which cannot be abused). However, in many circumstances, deciding whether a not an action is an abuse of the API will rely on a lot of context, so the API will not have nearly enough environmental information, nor reasoning capability to decide for itself.

To understand whether or not you *need* a Security API you need to understand whether there is any other way to split the problem into pieces that you already understand, and whether or not a Security API solution proposed to you will actually absorb some of the problem and stop it dead in its tracks, or whether or not it will return the problem to you in a new form (which may or may not be easier to solve).

To understand whether or not you *can* solve your problem with a Security API, you need to understand whether you have the resources to enforce all the smaller policies that the Security API will throw back at you. If your policies are highly context sensitive, the API will most likely throw them back at you in a different way, and it will take careful design for these new policies to be amenable to you. If, on the other hand, they are static and well-defined, the Security API may be able to rigidly interpret them.

So Security APIs are good at enforcing policies to the letter, and good at devolving responsibility for decisions elsewhere. This rigidity of purpose makes them both strong and brittle. They are strong because a persuasive attacker cannot socially engineer them to change their policy. They are brittle because this policy gives them a blinkered understanding of the outside world and the entities within it, and how to reason about them. Thus if there is a slight environmental change in the outside world, for instance if it becomes acceptable for several people to share an authorisation token, many of the security properties might be lost.

8.2 Design Heuristics

How do you design a Security API to make sure it does its job, and nothing more? This section provides some advice and heuristics for good API design, categorised into five sections.

8.2.1 General Heuristics

- Understand your threat model. If a clear policy has been provided stating the necessary properties of the API, it shouldn't be to hard to imagine how different potential attackers might try to violate these properties, and then design in some resistance. If you have not been given a clear policy that defines which threats the API must resist, be conservative. Protect against threats which can be dealt with, but don't get bogged down with complexity trying to solve *hard problems*. For example, Certification Authority APIs must avoid the *authorisation problem* – "how do you decide when to give permission to use a signing key?". This problem is so subjective that trying to encode a partial solution into the API is probably a bad idea. Likewise, HSMs communicating with unknown other principals should not get lost in the *secure introduction problem* – "how do you know you are communicating securely to someone you have only just met for the first time?". The 4758 CCA's role-based access control system is arguably an example of an API trying to bite off too large a chunk of a hard problem. Although it is powerful and flexible, instead of increasing assurance of security, it increases uncertainty that the configuration and access permissions of the device are correct for the policy.

- Just as there can be no such thing as a general purpose security policy, thus there is no such thing as a general purpose *security* API. However, a general purpose API can be a useful thing – it has architectural features for data representation and commands and access control which need not be redesigned from scratch. It is only when a specific policy is added that it turns into a *Security API*. If you are designing a general purpose API, recognise that it is policy free, and that someone will have to add the policy. Make sure OEMs can shield its features from clients. Do this either with fine-grain access control (highly flexible transactions need highly flexible access control), or by allowing OEM code to enter the trusted perimeter and establish a second layer API.

- Avoid adopting an unnecessarily complex general purpose API to get the functionality you want – you'll get more functionality than you need, and you may not be able to easily switch off the bits you don't like. If you want to build on existing technology, a better approach is to commission an extension to an API. Section 7.3.9 describes problems with the CCA `Key_Test` command. This command is a general purpose check value calculation facility, and though

it can be enabled and disabled, the different input parameters cannot be properly locked down.

- Invoke public key technology only when necessary, when the benefits to the larger system are clear. Consider whether your product will only interact with its own kind, or whether it must be compatible with heterogeneous designs. Be aware that public key technology (especially PKCS standards for RSA) have some policy decisions embedded in the underlying constructs. This embedding can make robustness and diversity easier, as simply by being compatible with another node, you can take for granted certain aspects of their security policy. However, if you take too many standards on board, compliance may become a problem (Davis details some of the problems in [17]).

  There is also the more subtle problem of loss of explicitness in your own security policy. For example whilst PKCS#1 public key padding is an industry standard and considered vital for providing a useful API, implementing it within the HSM puts a whole set of implicit requirements on the module to be responsible for protection against the sorts of mathematical attacks on RSA that PKCS#1 was designed to resist. The split of responsibility between the PKI software and the supporting crypto hardware can get quite muddied. Identifying and resisting transfers of responsibility requires a contextual awareness outside the scope of the API, so if your policy is not well defined, understanding who amongst the suppliers, developers and integrators has taken responsibility for an aspect of your policy will be very difficult.

- Look at the environments in which a device implementing your API will operate, and find opportunities for trusted interaction. Exploit trustworthy environments to establish an identity behind your API, and to introduce it to trustworthy parties. The *'XOR to Null Key'* attack in section 7.3.1 was effectively fixed by exploiting the available trusted path to make all of the key loading operation secure.

- Limit the amount of state which can be accumulated in and around your device, if only to make analysis easier. Don't let it hold more keys than it needs to, and don't allow keys to be used for longer than they need to. The unnecessarily high capacity of monotonic APIs has made attack techniques such as the 'meet-in-the-middle' attack much easier (see section 7.2.2).

8.2.2 Access Control

- Give your API a view of the outside world with just enough knowledge for its policies to make sense. If the policies are "*everyone* must do something" or "*no-one* should be able to do something", then it may be sufficient to have one conceptual identity – "the outside world".

- Decide what abuse will be *prevented*, and what abuse will only be *detected*. Separate the API features that prevent from the features that detect, and do not muddle roles and identities. Assign roles to people to prevent abuse; collect identities of people to detect abuse. When supporting the detection of abuse, be whole hearted – identify real world individuals (not roles) to the device, and give secure audit of the actions they perform. The proposed attacks on Baltimore's implementation of Security Officer access control shows what can happen if the purpose of tokens given to humans gets muddled (section 8.4.1). When non-human entities issue commands, think carefully about who is responsible for this entity – is it the author of the entity code, or the shift supervisor? If it is a supervisor, make sure they are properly trained to understand this responsibility.

- Try not leave clients with complex decisions on how to configure their access control systems; it is better if you design only one way that your system can be configured to achieve a particular end, rather than multiple ways your system can be configured to accommodate a range of procedural controls. You need then only state clearly the assumptions you make about the client's procedural controls. Vendors may prefer to externalise these issues, and devolve responsibility, but interactivity will make for more secure systems. Interactivity between API designer and client can be encouraged by employing vendor-enabling codes to unlock API functionality (these are sometimes already present for feature-oriented marketing strategies). You can then maintain a stake in the configuration of the system, and provide advice as to the consequences of changing functionality.

- Be cautious about using dual control as its use can easily create the poor incentives for long-term security. Decide which parties in your dual control system should be active, and which should be passive. Two common situations are:

    1. Two parties both make an independent decision whether to authorise, and are not necessarily expected to concur. Here both parties have to be active.
    2. Both parties parties are aware of a procedure that is to be followed to the letter, and must ensure that together it is done exactly according the specifications. Here, one party can be active, and one passive.

  Be careful about mixing active and passive dual control. Split responsibility can quickly turn into diminished responsibility, so an easy mistake will be for one party to ignore his duty to monitor the activities of the other, thinking instead that the small active stage he has to perform is all that matters.

- When the host machine sends data that the HSM cannot check for integrity, try to make it difficult for the host to modify the data undetected. Exploit trusted paths such as smartcard readers and built-in displays, as these can be used to provide feedback upon the host's activity. Another option is to maintain an audit trail; the risk of detection is a powerful deterrent to possible subversion of the host.

- Do not make authorisation tokens multi-purpose. In particular, do not use the same token for authorisation and identification.

8.2.3 Transaction Design

- Treat the design of each transaction like the construction of a protocol and follow the *explicitness* and *simplicity* principles: be explicit about identifying the types, sources and destinations of data passing, but keep the design simple, and ensure that the additional data you have included can be verified in a meaningful way.

- Maintain orthogonality between different sets of API functions. Avoid re-using a command to achieve extra functionality. In particular, keep backup orthogonal to import/export.

- Protect the integrity of data. Protect the confidentiality of data where necessary. The correct tool to use to protect integrity is a MAC with a suitable size block cipher, a keyed cryptographic hash function, or a signed cryptographic hash of the data. Never use parity bits or CRCs for integrity checking.

- Be careful how you process secret information in a transaction. Be aware that each error message which is conditional upon secret data will leak some information about that secret data. Try to avoid entropy leakage all together, or ensure that small pieces of information cannot be aggregated. Partition your error return code space to make it easier to manage and contain sensitive errors. However, where possible, it is preferable for a transaction to return an error code that does not leak information than for it to produce 'garbage' (i.e. results which are undefined when the inputs are invalid) – this practice frustrates analysis. Sections 7.3.10 and 7.3.12 describe information leakage attacks.

- As well as watching for information leakage through the output of an API command, be aware of information leakage through side-channels, in particular timing attacks, as they can be performed purely in software. Section 7.3.8 describes a timing attack where DES keys generated can be identified by observing timing characteristics of partial matches against the table of DES weak keys.

- Put randomisation and redundancy in encrypted secret data. If the data contains weak secrets, or different data values have a non-uniform distribution, then encryption without randomisation will leak information. If you use randomisation, be sure that you have enough redundancy, or use a cryptographic integrity check, otherwise the randomness fields may make it easier to conjure encrypted data (see sections 7.2.3 and 7.3.5).

- Think carefully about structural similarities between transactions that perform the same sort of task, but for a different part of the API. Identify when a task is roughly the same and imagine you are going to re-use the code in your implementation. Put in measures to ensure there is an explicit incompatibility between inputs and outputs to the data flowing in and out of the shared functions. As a rule of thumb, if you identify similar transactions, redesign them to make them completely different, or exactly the same.

8.2.4 Type System Design

- Categorise key material and data by form and allowed usage, and create 'type information' describing each usage policy. Bind this type information to the data cryptographically. Be explicit when generating this type system, and make things as simple as possible (but no simpler).

- Create a graphical representation of your type system using boxes to represent types and arrows describing the major information flow through transactions. Section 7.1.2 describes the author's preferred type system notation, which can express the much of the semantics of an API in a single diagram.

- Analyse the type system for bi-directional flows of information, and try to eliminate the need for them, should they occur. Bi-directional flow creates equivalent types, which add unnecessary complexity.

- Most type systems are concerned with the flow of secret key material. Known or chosen keys should not be allowed into such type systems.

- Assemble your key material into a hierarchy. Try to keep the hierarchy as shallow as possible. Then map out the types on this diagram, and look out for and avoid types whose roles cross hierarchical boundaries – these are a very bad idea. The hierarchy diagram of the VSM [8] clearly showed an anomaly which led to an attack on the VSM (see section 7.3.2).

- If a single HSM is to be multi-purpose, virtualise the entire device and ensure that no data from one virtual device can be used in the other. Avoid cohabitation of unrelated data in the same hierarchy or type system. Restrict entry of key material into type systems and hierarchies using authorisation. Do this upon generation and import.

8.2.5 Legacy Issues

- Isolate your legacy support from the rest of the system. Where possible, use an orthogonal command set for the legacy mode. In effect, create a virtual legacy device and only pass data between your device and the legacy device when absolutely necessary, and even then only according to well analysed restricted rules.

- Make sure the legacy support can actually be switched off for all transactions. In a monotonic API design, the existence of old legacy tokens could compromise a modern installation, even though the generation facilities for new tokens of this form have been disabled.

- Avoid related key sets. Section 7.2.4 described the many problems they create. If related keys must be used for some protocol, try to generate them dynamically from a master key. Keeping the value of the relationship between keys secret is possibly sufficient.

- If you must use encryption with short key lengths, watch for parallel attacks on sets of keys (see section 7.2.2 for the attack technique, and sections 7.3.3, 7.3.6, 7.3.7, 7.3.9 and 7.3.11 to see the far reaching consequences of this attack). Possible preventative measures include limiting membership levels of types to avoid the meet-in-the-middle attack, or preventing test vector generation. To push the 'limited type membership' idea to the extreme – have a set of slots that can contain keys, and restrict the procedure to re-load keys into these slots. That way you can generate new keys without restriction, but if you crack one of them it won't help, because you can't go back and access it again.

- Ensure that keys are 'atomic': permitting manipulation of key parts is dangerous (see sections 7.2.5 and 7.3.6). When keys are derived using master secrets and device-specific information such as serial numbers, avoid adding structure to the key material. Don't provide generic mechanisms for supporting key material extraction operations, such as those used in SSL – PKCS#11 does this with it's `EXTRACT_KEY_FROM_KEY` mechanism, which is useless unless all the parameters can be authenticated generically. A better approach is to generate a special transaction which can only support this particular usage.

- If you do identify a weaknesses in your API assume the worst – that it is generic. Search carefully for all possible instances of attack exploiting this weakness, then seek an equally generic solution. Patching individual parts of the transaction set is unlikely to solve all of the problems. The author nearly got caught out in [10] suggesting a 'sanity check' countermeasure for the decimalisation table attack which was not generic, and could have been

defeated by trivially upgrading the attack to modify the decimalisation table in a more subtle way (see section 7.3.10).

8.3 Access Control and Trusted Paths

Access control is necessary to ensure that only authorised users have access to powerful transactions which could be used to extract sensitive information. These can be used to enforce *procedural controls* such as *dual control*, or *m-of-n sharing schemes*, to prevent abuse of the more powerful transactions.

The simplest access control systems grant special authority to whoever has first use of the HSM and then go into the default mode which affords no special privileges. An authorised person or group will load the sensitive information into the HSM at power-up; afterwards the transaction set does not permit extraction of this information, only manipulation of other data using it. The next step up in access control is including a special *authorised* mode which can be enabled at any time with one or more passwords, physical key switches, or smartcards.

More versatile access control systems will maintain a record of which transactions each user can access, or a role-based approach to permit easier restructuring as the job of an individual real-world user changes, either in the long term or through the course of a working day. In circumstances where there are multiple levels of authorisation, the existence of a *'trusted path'* to users issuing special commands becomes important. Without using a secured session or physical access port separation, it would be easy for an unauthorised person to insert commands of their own into this session to extract sensitive information under the very nose of the authorised user.

In many applications of HSMs, human authorisation is an important part of normal operation. There is a danger that the path by which this authorisation reaches the HSM will be seen as the weakest link in the chain, so this trusted path must be carefully designed.

The phrase *'trusted path'* is used to describe a channel by which sensitive data is communicated to and from a HSM. This could be a special port for attaching a keypad or smartcard reader, or could be an encrypted communications channel which passes through the host before reaching some piece of "trusted hardware", maybe a special keypad with a microcontroller. Some HSMs do not have long-term trusted paths for data – they instead use the same channel all the time, and have a ratchet function that ensures that the level of trust in this path can only go down. Resetting of the trust ratchet incurs a wipe of the HSM memory. For certain applications, this model is sufficient. The "resurrecting duckling" policy [38] and its terminology are useful in understanding this trust model.

A trusted path must always end with a trusted device, which should be tamper evident. A classic example of a mistake in key entry procedure is described in [3].

The Visa Security Module had a serial port to which a VT100 terminal was normally connected. However, one bank had a purchase policy that only IBM terminals should be bought, and these were not compatible with the VSM's trusted path. A bank programmer helpfully provided a compatible laptop, which could easily have included key logging software.

8.3.1 How much should we trust the host?

Security processors were originally created to avoid placing trust in the O/S and physical security of the host computer, but due to inadequate trusted I/O channels and 'defence in depth' policies, API designers tend to place some trust in the host. Once the host takes its own security precautions, the split of responsibility becomes a grey area, and in the worst case can lead to risk dumping.

Certainly banks running highly developed accounting and book-keeping systems that interface with a HSM believe they can exercise control over the host. For instance, IBM stated in response to software attacking banking security HSMs, that an insider would not be granted access to run programs of his choosing and copy information from the host [24]. This may have been the case when the host was a mainframe and was programmed in obscure languages, but PC hosts with modern software set a much lower bar for the attacker. Furthermore, while banking HSMs may have few human operators interacting directly with the host, humans regularly interact with PKI supporting HSMs in the course of their daily duties.

If the host is connected to some form of network – a VPN or the internet itself – it is theoretically vulnerable to many more than just its usual operators. But compromising a machine remotely and remaining undetected is far from straightforward; the unusual traffic may be spotted anywhere between entering the corporate VPN and reaching the host. An insider's physical access to the host machine and even its hard drive makes defeating O/S security almost trivial. Once an insider with physical access has compromised the O/S security, he may of course choose to do some of the sabotage work remotely, to avoid attracting attention by working too long at the machine.

Highly protected modules disconnected from networks by policy sport different weaknesses: Discouraging access to the these will make it likely that the O/S will remain the vanilla version of whatever was first installed, requiring a multitude of patches to even start blocking all the security holes. If procedural controls for initialising the host are tight, the O/S will likely be installed with minimum configuration changes and explicit instructions on what these settings should be. Thus disconnected machines tend to be in well documented insecure configurations. If there is a weakness, an attacker could write a script, and may be able to exploit it with only seconds of access to the HSM.

In summary, the insider threat is so great that it is unlikely that a host machine can resist determined attempts to run unauthorised software. However, subtle sabotage of the existing host software is harder, and is most feasible when the insider can continue their attack via the local network.

Despite this risk, it makes sense trust the host at least as much as its least privileged operators. As a rule of thumb, host compromises that result in low-level abuses detectable by audit can be little worse than abuse by an operator (though blame is harder to assign). It is only when HSM manufacturers require a full re-establishment of a trust in the host O/S integrity before performing sensitive operations that end-user compliance is unrealistic.

8.3.2 Communicating: Key Material

Key material must be exchanged whenever a link is established between two parties who have never communicated before. For symmetric keys, this key material had to be kept secret, so dedicated terminals or keypads were introduced especially for key entry. The VSM had a serial port which was typically connected to a VT100 terminal. VT100 terminals were not programmable devices – unobserved modification of the device to achieve eavesdropping capability would be difficult.

In the case of public key HSMs, during key-loading it is only the integrity of the key which needs to be assured. A new key entry approach developed, where the key material was entered through an untrusted path, and then a trusted path was used for the HSM to report what it had received (i.e. a hash of the public key entered), and for a final authorisation to enable the key. Public key HSMs can be considered to have two trusted paths, one for display and one for authorisation, instead of the original key material trusted path of the previous generation.

Because this sort of key material can be authorised in retrospect of its entry, PKI HSMs shift their concern away from secure entry of key material, and consider the authorisation of the material as a quite different act from its entry. Both FIPS 140-1 and the new 140-2 have not captured this shift of emphasis, and maintain that anything within the module is sensitive, and privileged. The PKI analogue of FIPS key generation/import is admission of the key into a hierarchy, which is done with a signing operation, producing a certificate. These signing operations are governed using a trusted authorisation path.

The IBM 4758 CCA recommends use of public key cryptography for securing communications, and provides dual control governing the acceptance of public keys. It has no trusted path for communication at all. Sabotaging the host to swap in a false public key during the dual control entry phase would be non-trivial. However, the legacy approach of entering key material in the clear is an even worse case. Here, setting a single environment variable `CSU_DUMP` to `TRUE` is enough to capture all the key material entering a 4758 in a debugging log file. The lesson here is that

Class I	Passive tokens, which present a shared secret upon request, and readily duplicated with appropriate hardware. includes passwords, PINs, and secret holding smartcards.
Class II	Active tokens, which are keyed to a specific device
Class III	Active tokens, which require secondary authorisation (e.g. entering of a PIN)
Class IV	Active tokens, with integral trusted I/O capability

Figure 8.1: Classification of authorisation tokens

communications paths, trusted or otherwise, cannot easily accommodate the entry of *confidential* key material – eavesdropping on unencrypted communications is too easy.

8.3.3 Communicating: Authorisation Information

The authorisation trusted path need only transmit a single bit of information – *yes* or *no*. However, designing trusted paths for authorisation has proved harder than one would think, because they must be viewed in conjunction with a display trusted path, which reveals to the authoriser the nature of the request.

Different types of authorisation token have advantages and disadvantages depending upon the nature of the request to be performed. A classification of authorisation tokens is shown in figure 8.1, based upon input output capabilities of the tokens.

Temporal authorisation is where the trusted path will respond with a *yes* for authorisation requests, for the duration in which the token is present. *One-shot authorisation*, is where a token will only respond with a *yes* when it is activated with respect to a specific request.

Passwords

It is common to use passwords to authorise access to both HSM functions, and data stored on other access tokens. It is sometimes impractical to provide the HSM with a dedicated keyboard for password entry, so the passwords are routed via the host.

This is the case with nCipher, and CCA RBAC passphrases. A subverted host could of course record these passwords as they are entered, but nevertheless this approach remains popular.

In the case of the 4758 RBAC, the passphrase is the only line of defence; it is entered on the keyboard of the host PC, and keypress logging software or hardware is easy to install. It is vital to regularly audit this sort of system to maintain assurance that users have not been compromised, but a simple hardware logger [41] may be all that is required to easily add and removing the logging capability without leaving a 'paper trail'. Passwords still are reasonably common in authorisation situations: they are rarely the only line of defence, and the risks involved in mounting a long term eavesdropping attack are not insignificant.

One smartcard reader vendor [42] connects their reader between keyboard and host, so that they can bypass the host when a password or PIN is to be submitted to the smartcard. This is a step in the right direction, but whilst enabling or disabling the bypass is optional, the challenge simply becomes for the host machine to fool the user into entering their password when the bypass is not activated.

The major weakness of passwords is that they can be duplicated by completely passive eavesdropping.

Special Keypads

PKI HSM vendors Baltimore and Chrysalis both support PIN entry as part of the authorisation process. Baltimore's modules in particular actually integrate the special keypad into the front of the security module housing. Special keypads are useful because they push up the difficulty of intercepting the trusted path, not because keypad entry is inherently more secure, but just because they are more likely to be proprietary and difficult to duplicate. Chrysalis has a special approved datakey reader with integrated keypad, shown in figure 8.2.

There remains a danger of hardware interception: the smartcard may be able to secure its trusted path to the host with a previously generated shared secret, but the same is not generally true of the keypad or GUI. In fact, in the case of the Luna PED, both the key material from the datakey and the user's PIN travel in the clear down the cable connecting the PED to the dock, which is a standard RJ45 ethernet cable.

To get maximum benefit from special keypads, they should be easy for users to identify as genuine, difficult for an attacker to sabotage or replace, securely connected to the HSMs, only used for communicating the most sensitive information.

Figure 8.2: Chrysalis Luna PED (PIN Entry Device)

Smartcards

Smartcards are good for *temporal authorisation*, but poor for *one-shot authorisation*.

The chip itself has no input/output capability aside from that reserved for communication with the host. It has no concept of its physical location, nor whether or not it is present in a reader. It can only note power being applied and removed from its power pins, and assume that it has been inserted.

A straightforward way to set up temporal authorisation is for the HSM to engage in a continual dialogue with the smartcard, e.g. repeating a challenge-response authentication protocol, with a repetition rate set as desired. If the smartcard is removed, the HSM will notice within at most one repetition period, and will terminate the authorisation. However, when operating in a hostile environment, the duration of authorisation should only be considered terminated once the smartcard has been returned to a sealed environment out of which it cannot communicate. Rigorous procedural controls should specify this as a safe with no room for wires, and electromagnetic shielding.

One shot authorisation is far more difficult to achieve with smartcards, because the internal circuitry of the smartcard has no way of telling what is happening to the environment outside it. The only way to achieve one-shot authorisation with smartcards is by placing trust in the smartcard reader. The smartcard reader needs to explicitly represent the act of single authorisation. One way to do this would be for the reader to leave the smartcard unpowered, and only power it up for a fixed duration (e.g. 5 seconds) upon a press of a button on the reader. Human auditors can thus observe the number of one-shot authorisations occurring, but the reader must

Figure 8.3: Smartcards and reader used with the nCipher nForce

be trusted to make sure that nothing can interrupt the power between the itself and the card. The electrical contacts on frequently used smartcards may deteriorate, and the resulting errors must be taken into account when designing procedures for authorisation. A set of administrator smartcards and their corresponding reader is shown in figure 8.3. Note that nCipher's 5 1/2" form factor nShields (see section 6.3.4) have an integrated smartcard reader, which is much more difficult to tamper with.

Keyswitches

Keyswitches have some nice properties, and they have been used successfully in military environments. Specially chosen key types are very difficult to copy without specialist equipment, which is expensive, and difficult to obtain.

The disadvantage of keyswitches is that they are not really compatible with high quality tamper-resistant enclosure. They must either be tamper resistant themselves, or sit on the boundary of tamper resistance for the main module. If the key switch is to be tamper-resistant, it must have the computing power to set up a secure connection between itself and the HSM. Constructing a tamper resistant package with all this functionality is just a reinstantiation of the latter problem – including a non-tamper-resistant keyswitch on the boundary of the HSM itself. This problem is hard too.

The real undoing of keyswitches is their inflexibility. Protecting against key loss, and implementing configurable threshold sharing schemes are difficult when the manufacturer must be consulted each time. This approach simply moves the weakest

link to the authorisation process for requesting an extra key from the manufacturer.

Other Access Tokens

Electronic key shaped physical tokens have very similar properties to smartcards, though the physical form factor is arguably preferable as it psychologically reminds the holders to take the same care of the token as they would with a physical key. In addition, the form factor of datakeys (seen in figure 8.4) has the connector inset into the plastic of the key, and this makes stealthy establishment of electrical connect rather more difficult than with a smartcard.

Figure 8.4: Datakeys used with the Chrysalis Luna PED

Biometrics (and any devices with significant error rates) are inappropriate for one-shot authorisation.

8.3.4 Providing Feedback: Display Information

Trusted paths can be useful for providing feedback. Though several manufacturers now have HSMs with multi-line LCD displays, none are yet utilising this display path for certification information. This means that it is impossible to tell exactly what is getting signed. There has been little support so far for implementing such displays, because designers are uncertain what information should be displayed to the user, and whether it is possible for a human to make an absolute judgement on this information before authorising the signature operation. This one of the reasons why procedural controls must still be designed to prevent host compromise.

The *"authorisation problem"* – writing a policy which describes when and by whom a key can be used is a hard problem, and is not a purely technical one. One of the important building blocks to helping pass this problem back to the outside world is the trusted display. Companies such as nCipher are only tentatively starting to integrate the policy processing into HSMs themselves, and once this happens, we may see the displays taking a more active role than just reporting status.

Figure 8.5: nCipher's netHSM 'trusted display'

8.3.5 Recommendations

There can be no trusted path between HSM and operator without trusted initialisation of the module. It is true that even the initialisation process is susceptible to attack, but procedures have already been designed to permit the manufacturer to deliver a module which can be verified as un-tampered, so from there these procedures may be extensible to monitoring the activity of installing parties.

System designers should accept the need to trust the installers of the module, and capitalise on this channel of trust to bootstrap as much future trust as possible. The HSM should ideally run autonomously from this stage without ever requiring administrator intervention.

If authorisation decisions are outside the scope of the HSM, then a trusted path for both display and authorisation should be used. Designing an application so that the details of the action to be authorised are intelligible by humans is difficult, and somewhat outside the scope of a HSM manufacturer, who can only provide the trusted general purpose display. For authorisation, all the currently available electronic access tokens seem to have similar weaknesses.

To minimise the risk of breach, each should possess a serial number to make it individually identifiable. It should be hard to forge a token; not just from base materials, but also hard to pass off one genuine token as another. To this end, affixing multiple unique serial numbers which are difficult to forge (e.g. holographic stickers) may be part of the solution.

The future of authorisation tokens might head towards form factors with their own I/O channels, such as a mobile phone. Chrysalis' trusted keypad goes part of the way, but the crucial point is that this form factor allows each user to be responsible for the integrity of their own trusted path. Each token should either be locked away in a tempest shielded safe (so that no radio communication can occur with the token), or remain in the possession of its assigned owner. Swapping of tokens means that the single attacker could compromise multiple tokens.

Secondary token protection methods such as PINs or passphrases reduce the perceived value of the token and encourage carelessness. If PINs or passphrases are used, they should be bound to individuals rather than tokens.

8.4 Personnel and Social Engineering Issues

Whilst different approaches to trusted path between the security module and the user possess different pros and cons, all have to contend with attacks that work at a higher level, regardless of the mechanism used – those that exploit weaknesses in the personnel operating the machines, or in the human designers perception of how the machines should be operated.

Trust must be placed in the human operators, but deciding where to put it is not easy. There are no obvious tests to determine whether an employee is good or evil, but gauging their intellectual ability is much easier. Just like the principle of least privilege, there can be a principle of least ability – an employee should be capable enough to do his job and not any more capable. This could be considered the ultimate protection against exploits which have yet to be found. However, it opens up the possibility for easy deception by colleagues, superiors, or even inferiors in the organisation. The procedural exploits described here could well be thwarted by observant and intelligent operators.

A better principle is that of the proverb "Idle hands do the devil's work". An intelligent employee with no motive to commit a crime is preferable to a motivated but incompetent employee.

Once the employees are chosen, they need to be briefed with a set of instructions on how the HSM should be operated – the procedural controls. It is important to explain to those performing the controls the consequences of failure (in terms of the system, not just their jobs). Without offering this explanation, the actual controls

enforced may fall short in some subtle way, or migrate to insecure controls seemingly equivalent, particularly in organisations with a rapid turnover of staff. Let us now consider some specific examples where poorly designed or explained procedural controls can leave room for social engineering. For simplicity the discussion will talk only in terms of smartcards, but the principles apply equally to other access tokens of this class.

8.4.1 Dual Security Officer Attack

Baltimore's Sureware Keyper module [49] requires two security officers to perform privileged functions such as changing of master keys, and cloning of the module. However, only a single officer is required to perform less privileged operations such as changing the communications configuration, or re-enabling the module after a power down.

A corrupt operator could replace a genuine module with a false module, including smartcard reader and keypad. The operator then claims that the power has failed (maybe after staging a gratuitous fire alarm to cause confusion), and asks one of the security officers to re-enable it. As the security officer inserts his card and types his PIN, this data is routed to a genuine module, which then becomes halfway authorised towards privileged operations. The operator need only repeat the request to the other security officer, and the module is fully authorised.

Changing the module so that both cards are required simultaneously is not necessarily the correct solution. Although both security officers would now need to use their cards at the same time, a single operator may still be capable of deceiving both. Each could be asked to re-enable a different fake module, thus meeting the conditions for a full authorisation of one real module.

The operation of the module and properties of the access token need to be fully considered and harmonised with the procedural controls. If the cards are normally stored in the corporate safe, a suitable rule would be *"only one of the SO smartcards may be withdrawn from the safe by each SO officer. If the requests are separate they should be dealt with one at a time and in sequence, with at least an 5 minute gap between return of the first card and withdrawal of the second"*.

If the cards are permanently entrusted to security officers or are stored separately, then they should ideally make clear to the authoriser the exact nature of the action which is to be performed. As the I/O capabilities of the commonly used class I-III tokens (see figure 8.1) are very poor, they are not suitable as multi-role tokens. This is the primary weakness of the Sureware Keyper design, in legitimising the use of security officer cards for less sensitive configuration options. During this time the security officer may take less care of his card because he assumes it is useless on its own. But with or without this architectural feature, a mistake in the procedural controls still leaves the module vulnerable.

When no privileged operations are required, the best practice is to keep the access tokens locked in a well administered safe. The procedural controls should make it clear exactly which combinations of access control tokens are dangerous, and the HSM designers should adjust their functionality to keep the combinatorics simple.

When using a sensitive access token, a security officer must always check that the module he is authorising has no evidence of tamper, and that it is the correct module. This task will always remain time-consuming when the outermost housing is not the tamper-evident one, and does not provide identification.

The two security officer paradigm makes deception attacks too feasible, because it requires only a single deception if one security officer is corrupt, and may still leave simultaneous automated deception feasible. Where possible, procedural controls should be designed so that more than one simultaneous human deception is required to gain worthwhile access. The next example illustrates the risks of automated deception.

8.4.2 M-of-N Security Officer Attack

The poor performance of smartcard based authorisation tokens for one-shot authorisation has already been touched upon. The nCipher nForce module has neither a trusted path for ensuring one-shot authorisation, nor for verifying what action is about to be authorised. So when an operator inserts his smartcard to authorise a module to sign a certificate confirming someone as a new CA, how can the operator be sure that no other certificates have been signed at the same time?

One possible solution for this is to use an "action counter" in the HSM which limits the number of actions which can be performed per insertion of a valid card sequence. The operator still cannot be sure of what was signed, but if the number of signing operations per insertion is restricted to one, then if the operator observes his certificate to be correctly signed, he can deduce that nothing else has been signed.

But here is the catch: the host need only convince the operator that there was an error during the insertion to trick him into re-inserting his card. When the operator puts his card in, the host displays an error dialogue *"error reading smartcard, please try again"* or *"incorrect PIN, please try again"* and the operator will dutifully obey.

This trusted path issue is not easy to fix, because giving the operator inherent assurance that the correct certificate has been signed is a difficult task – particularly because certificates are not normally human readable.

This attack may seem trivial and obvious, but trying to fix it in a generic way reveals that lack of a trusted display path has considerable implications. Addressing anything other than this root problem will lead to errors. Suppose that the procedural controls are adjusted so that two cards are required. It appears that deceiving an operator into re-inserting a card can no longer produce the correct sequence, but this is not the case.

Imagine a Certification Authority where each signing operation which must be authorised by the insertion of two smartcards, each held by a different operator. Normally, each operator inserts his card into the reader in turn, and then a single signing operation is authorised. However, if the host computer reports an error upon insertion of the second smartcard, it can deceive that operator into performing a second insertion. The host performs the requested signing after the first insertion, then the second insertion is used to go half-way through a second run of the authorisation process. The module is left half authorised, and the corrupt operator can return later and complete this authorisation to gain a second signing. The actions of normal operation and the attack can be annotated as follows:

```
1.  AB*              (Normal Operation)

1.  AB*B             (Single SO Attack)
2.  B'A*
```

`X` stands for authorisation by principal X, `*` stands for a completed authorisation and signing operation, and `X'` stands for an authorisation carried over from a previous attempt. A worse scenario is where neither operator is corrupt, and the sabotaged host software steals a third signing after two completed authorisations.

```
1.  AB*B           (Corrupt Host Attack)
2.  B'*AB*
```

The same technique can scale up to *m-of-n* smartcard schemes, and one signing can be stolen per m valid signings. An example follows for $m = 4$:

```
1. ABCD*D          (Corrupt Host Attack)
2. D'ABC*CD
3. C'D'AB*BCD
4. B'C'D'A*ABCD*
```

As number of cardholders goes up, the probability of a workable order of insertion (given that the cardholders use the reader in a random order) goes down, but for $m \leq 3$ it is quite likely that the host can steal a signing. Demanding a reinsertion of the smartcard is not an unusual event; when a smartcard becomes damaged due to dirty contacts, insertions commonly fail.

8.4.3 How Many Security Officers are Best?

Even in the absence of ingenious attacks that can be perpetrated by a single security officer, the optimal number of security officers must be decided. Too few security officers makes collusion or deceit feasible, whilst involving too many can result in diminished responsibility.

If a company can be confident that abuse will be quickly spotted, the optimal number is one: blame is obvious, and the security officer will be aware of this. But companies protecting critical data and top level keys may not be happy entrusting their most valuable assets to a single person. Two security officers seems to have become the de facto standard, as the division of trust is still clear, and rising to three seems to add extra expense without a quantifiable increase in security. But three security officers does tighten security: a corrupt officer will be outnumbered, and deceiving two people in different locations simultaneously is next to impossible. The politics of negotiating a three-way collusion is also much harder: the two bad officers will have to agree on their perceptions of the third before approaching him. Forging agreement on character judgements when the stakes are high is very difficult. So while it may be unrealistic to have three people sitting in on a long-haul reconfiguration of the system, where the officers duties are short and clearly defined, three keyholders provides that extra protection.

8.4.4 Recommendations

After considering the issues, there are a plethora of decisions to be made when designing procedural controls for the handling of tokens. Should each person own a particular token, should it be signed in and out, or carried at all times? An insertion procedure must be developed, and rules for duration of possession, holidays and illness must be put in place. Some conditions are best enforced by functional separation (multiple personnel with differing tasks), and some by dual control (multiple personnel with the same task). However, the single overriding principle is to give the HSM maximum possible knowledge about its human environment. Separation of duty into "doers" and "auditors" encourages both to perform the task correctly, but care must be taken that pure auditors do not become lazy. Where possible the procedures should require conflicting sets of knowledge to circumvent e.g. involving personnel from different departments makes it unlikely that a single person will know both well enough to socially engineer them. These principles have been crystallised into some suggested procedures below:

- *Electronic Tokens.* Each token should always correspond directly to a single individual; thus the HSM is aware of how many distinct humans have been involved in each authorisation operation. When in the office, it should be carried on a chain around the neck, which is a well visible location and discourages it from being accidentally left behind.

- *Regular Authorisation (occurs during the normal operation of the company).* It is not economical to invoke dual control for all regular authorisations, thus the opposite approach of catching abusers at audit is employed instead. Audit cannot however catch abuses which are not yet known. To this end, there should be a supervisor who randomly inspects the usage of tokens. For example, whenever an operation is requested from the HSM, it could with a small probability demand further authorisation from a supervisor. If this authorisation is not given the HSM would raise an alarm. Thus while a token is abused for in a way that does not set off existing audit alarms, the supervisor is likely to become aware. A single token holder is the only viable model for regular authorisation, but it is important to have a physically present supervisor as well as a retrospective audit.

- *Key Entry.* There should be three keyholders chosen from different departments. All keyholders should remain in each others presence whilst in possession of the keys. The keys should be not be available singly – either they are all signed out, or none are. The storage media for the key should not be multi-role.

- *Irregular Authorisation.* For rarely occurring authorisation events there should be one management-level token holder and two trained security officers. Once the module has been highly authorised, one of the security officers should perform the actions while the other audits. Involving a management-level token holder at the beginning of the authorisation causes extra hassle which protects against security officers over-using their tokens, and encourages the security officers to assess the validity of each others requests. The tokens should be signed in and out simultaneously.

Chapter 9

The Future of Security APIs

Security APIs design has already gone through substantial change since the first APIs were incorporated into HSMs. They have advanced past monotonic designs with encrypted argument passing, and current designs do not shy away from internal state – they use object handles, usage counters, and storage slots. Meanwhile, HSM input/output facilities have developed too. Access control has moved on from the humble key-switch and has heartily embraced secret sharing schemes, smartcards and even role-based access control (RBAC). True Security APIs are now less and less designed by the hardware manufacturers, but instead by OEMs who take a general purpose crypto API and add in the security policy. The languages used for policy description have also become frighteningly complex.

As the Security API industry expands and begins to mature, will we be faced on all fronts with the "inevitable evolution of the Swiss army knife"? What will APIs look like in two decades time, who will use them, and will they be secure?

9.1 Designing APIs Right

There must be hope for the future because secure designs for simple APIs are already in our reach. Credit-dispensing APIs implement straightforward policies which do not have too ambitious goals, and on the whole, they work. After Prism's TSM200 API has its master key loading weakness fixed (section 7.3.11), we are left with a relatively simple API that takes a secret key, holds a value counter, and dispenses electronic credit tokens until the value counter runs out. This API is secure, as far as we can tell.

APIs like this work because the security policy does not take on the whole real-world problem that the security architect is faced with, instead considering a smaller problem and completely solving it. All the API promises is that the device will not dispense more than a certain amount of credit; it does not worry about to whom and where and when dispensing can happen. This simple building block is a valuable

component of a real-life infrastructure because it prevents a catastrophic failure mode of the system. A more ambitious API may promise even greater potential security, but fall short of its greater goals and have a glitch that lets an even worse failure mode occur.

In the future we are likely to be faced with larger and more complex real-world problems, that will tax our ability to identify the neatly soluble subcomponents. The danger is that the API designers of the future will be given very high-level problems, and will in effect have to perform security architecture design. Digital Rights Management systems, for instance, are designed to enforce qualitatively more complex marketing models than those that are used today – the high-level policy is in essence that everyone should pay exactly what they can afford.

If in the future API design policies remain as simple as those of protocol design, we will have all the tools we need to design a good API. We have modern crypto algorithms which are comfortably strong enough; we have much less limitation on storage and processing power; we have all sorts of useful peripherals and trusted I/O devices for controlling and interacting with the API. But if the larger component of Security API design is security architecture, we need new design technology as well. We will need tools to manage complexity, a better understanding of policy and procedural design, and good methods for splitting policies into appropriate well-understood chunks.

So a crucial aspect to designing APIs right in the future will be to spot the hard problems that lurk within high-level security policies, and not to take them on – leave them for the security architect to see and deal with. Let us suppose that future API designers are given well-formed and simple policies to implement: what will the resulting APIs look like?

9.2 Future API Architectures

One possibility is that new Security APIs will all be specialisations built upon the general-purpose crypto APIs provided by HSM manufacturers and O/S vendors. There would be a few dominant vendors (or maybe only one) of general purpose crypto APIs. New APIs will be built from configuration information that specialises the key and data type systems, and maybe a couple of specialised transactions written in a scripting language that glues existing transactions together. In this vision of the future, APIs will become more and more heavyweight – each encrypted data token could be several kilobytes in size, maybe requiring a public key operation, symmetric decryption and multiple hash function invocations just to check. If APIs are designed by building upon a rich feature set, there will be an even worse danger of API bloat. Even though designers have a clear policy to implement, they may have difficulty understanding the general purpose API's full feature set. They may

be unable to assure themselves that their configuration choices for the general-purpose API do actually implement their policy. We are already beginning to see this problem in today's APIs. Some OEMs are finding that only a few of their programmers fully understand all the features of a complex API such as the nCipher nCore API. They end up creating their own intermediary APIs to lock down certain features and make it easier to review code written by less trusted or less skilled programmers. In the future this problem could get much worse.

An alternate vision for the future does not entail API bloat in quite the same way. As wireless computers become ubiquitous in the future, communications security will be pushed forward. The constant requirement for interaction between low-end embedded systems and more sophisticated devices or HSMs acting as 'concentrators' may keep API design rooted in low-level bit-twiddling primitives. These primitives will be poorly suited to implementation in the less powerful and restrictive scripting languages which can only grow in complexity to the limit of what can be sandboxed inside an HSM. So instead we may see APIs continuing to be designed afresh, where the HSM manufacturer provides only the tamper-resistant shell and some hardware crypto support, and the API message specifications are hand grown by the same person who is trying to implement the security policy. In this future, there is a good chance that such APIs can be designed right, so long as the security policy is manageable and not too ambitious. Ironically, the wisdom accumulated in this thesis about good design of the older, less powerful HSMs could continue to be relevant for some time, if Security APIs are pushed out into less powerful platforms.

In the very long term, *ease of analysis* may be a new player shaping the design of future APIs. If particular tools and techniques that are popular and familiar to the protocol design community are adopted, then APIs may well develop to be amenable to these forms of semi-automated analysis. These tools will probably reward explicitness, and concise transactions, rather than parameterised ones. Such tools are currently poor at dealing with the 'state space explosion'; it seems probable that the tools will encourage designs where all the long-term state is accumulated in once conceptual place, rather than being spread across internal registers, encrypted inputs and authorisation tokens. It may even happen that future designers will find a way to limit the accumulation of state, for instance by resetting to a default state after a certain number of command executions.

Finally, we must always be wary of function creep: today's APIs will certainly expand to take on new purposes tomorrow, whether through expansion of an HSM manufacturer's underlying general-purpose API, or through development of the security policy. In addition to the risks of inadvertent addition of dangerous features, we have to watch for economic incentives punishing security – developers may knowingly add strange and risky new facilities to an API, if the value of opening new markets is seen to be more important than encouraging a high standard of security.

9.3 Trusted Computing

How much work will there be for Security API designers in the future? The relaxation of US export restrictions on cryptographic hardware and the bursting of the dot com bubble have not been good for the size and diversity of the Security API industry. However, Trusted Computing could change everything; it is the biggest single factor in future prospects for Security APIs. Trusted Computing has become a catch-all term describing efforts by multiple large hardware and software vendors to allow security policies to be enforced effectively on their platforms. It is already surrounded by controversy (see section 5.3.11 and Anderson's TC FAQ [55]) .

If Trusted Computing arrives on the PC platform, myriads of new Security APIs will follow. Microsoft's Next Generation Secure Computing Base (NGSCB, previously known as 'Palladium') aims to allow any and all application developers to create trusted components of their applications, which will perform sensitive processing in a memory space curtained and protected from both other processes and a potentially sabotaged operating system. At the boundary to every trusted component there will be a policy on access to the sensitive data and thus a Security API of sorts.

There is lots of speculation upon possible and intended applications of this technology. Media vendors will use it to create Digital Rights Management systems, software vendors will look to achieve lock-in to their product lines, and service providers will use it to support new purchase models. It seems that even the smallest shareware vendors will have the opportunity to exploit this technology to enforce whatever registration requirements they wish. Trusted Computing Security APIs will probably look more and more like interfaces between object-oriented classes, and it is likely that this mixing of API design with hands-on programming will fragment the policy definition of the API, and commands will be poorly specified, if at all. In the rush to exploit the new centre of trust, there will be plenty of hard decisions to be made; they will likely be glossed over, and only got right on the third release. One trend may be to place as much functionality in the trusted component as possible. For instance, consider how much code is required to render an HTML email. It is also likely that these poorly designed trusted components will be broken in the same ways that operating systems are today: by having buffer and integer overflows, and through exploitation of the lesser used parts of the API which were never properly tested for security. In fact it is not entirely clear that the quality of code inside HSMs is that good at the moment – it may just be that not enough people are searching for the classic faults.

If Trusted Computing does arrive, it will be more important than ever not just to understand how to design Security APIs right, but also to communicate this understanding and build it into the skill set that the average programmer picks up.

9.4 The Bottom Line

Whilst it is exciting for industry and researchers to lick their lips and imagine a world with HSMs in every PC, and Security APIs everywhere, we have to accept that none of this new technology – Security APIs, Hardware Security Modules, Trusted Computing, Whatever – is going to be a silver bullet. There is no silver bullet. The hope for the future of Security APIs is not that they will solve hard problems for good, but that they will become important bricks, even keystones in security architectures, and that we can apply the engineering know-how we have to get these APIs correct and secure.

Chapter 10

Conclusions

This thesis has brought the design of Security APIs out into the open. Chapter 6 reveals pictures of HSMs that are a long way from being consumer devices in the public eye (until the late 90s they were classified as munitions in the US). Chapter 7 explores the API abstractions, designs and architectures and shows what has gone wrong with existing APIs. Under the harsh light of day we see that *every* HSM manufacturer whose Security API has been analysed has had a vulnerability identified, most of which have been detailed in this thesis. Some APIs have suffered catastrophic failures – a master key compromise on the Prism TSM200 (section 7.3.11), and nearly every financial HSM broken by the decimalisation table attack (section 7.3.10). We see practical implementations of theoretical attacks (section 7.3.7) that reveal aspects of both the system attacked and the attack method itself, that are difficult to spot in any other way.

The harsh light of day also shows us a more unpleasant truth: we are still largely ignorant about the causes of these failures. How did the designers fail to notice the vulnerabilities, and what new wisdom can they be given to enable them to get it right next time? Chapter 8 discusses heuristics for API design, drawing together established wisdom from other areas of security, in particular highlighting the ever-applicable robustness principles of explicitness and simplicity. Yet there is little in these heuristics that is fundamentally new and has been until now unavailable to designers.

We could resign ourselves to ignorance, or continue to search blindly for good heuristics. On the other hand, maybe the truth is that little *new* wisdom is actually needed for Security API design – it is just a matter of assembling the knowledge we have, giving it a name, and building it into the set of skills we impart to the next generation of programmers. For this approach to work, we have to get the roles and responsibilities right.

10.1 Roles and Responsibilities

Security APIs aim to enforce policies on the manipulation of sensitive data. When an API attack is performed, it is the policy in the specification document given to the Security API that is violated. The trouble is that in real life this API-level policy document may not exist, and there is probably not an API designer to read it anyway. Instead, it seems that APIs are designed by someone examining the top-level policy: what the entire system – people, computers, bits, bytes and all – is supposed to do, and trying to conceive a computer component that bites off as large a chunk of the problem as possible.

It is this continuing practice that could keep Security API design a hard problem, where mistake after mistake is made. While this lack of definition in the API security policy makes it hard to build good APIs, it also has the side-effect of creating an identity crisis for API research.

The vulnerabilities discovered and catalogued in chapter 7 draw on a bewildering range of principles. All of them are clearly failures of the system as a whole, but it is hard to pick one out and declare it to be a typical API attack. A forced decision might conclude that a type confusion attack (such as that on the VSM in section 7.3.2) is typical. Restricting our scope of attacks to those similar to this, we find firstly that we have only a few attacks, and secondly that they all exploit classic failures well known and documented in security protocols literature, such as key binding and separation of usage. This definition reduces Security API analysis to a backwater of protocol analysis.

On the other hand, if we embrace the many and varied attacks, and declare them all to be Security API attacks, we can only conclude that the API designer must be the security architect – the man with the big picture in his head.

The separation of roles between security architect and Security API designer is identified in chapter 9 as crucial in the shaping of the future of Security APIs and our ability to tackle harder real-world problems in the future. Without the role separation, Security API research will be stuck in a state of disarray: a messy smorgasbord of knowledge, techniques and wisdom plucked from other fields of security research. It is up to the security architect to try to develop an *understanding of Security APIs*, create a role for the API designer, and resolve this identity crisis. Armed with a broad-ranging familiarity of Security API architectures, hardware security modules and procedural controls, a security architect should become able to perceive a potential conceptual boundary at the HSM component of their design. With encouragement he might put some of his security policy there, and there will emerge a 'Security API designer' role, in which it is possible to get a design 100% right. The policy assigned to this role must be one that benefits from the rigorous and unforgiving execution of a computer, otherwise the HSM will be a champion of mediocrity, and require as much human attention and supervision as another untrustworthy human would.

10.2 The Security API Designer

The Security API designer may now have a clear policy to implement, but will not necessarily have an easy job. She will need to appreciate the target device's view of the world – its input and output capabilities, its trusted initialisation, authorisation, identification and feedback channels, its cryptographic primitives and storage architecture. She will then need to diligently balance simplicity and explicitness in the design of the transaction set, obey principles of binding, key separation, and carefully monitor information leakage of confidential data processed.

The Security API designer will have to choose whether to build her design on top of general purpose crypto services, or alternately to grow a new API from scratch each time. She must also be rigorous in avoiding absolutely all implementation faults, as it is extremely hard to incorporate robustness in a design against attacks combining both specification and implementation level faults. If the policy given to her is clear enough, she may possibly benefit from formal methods at design time: identifying complexity bloat, and spotting and preventing classic binding, key separation and type confusion attacks before they happen. New formal methods may even be developed to help quantify and set bounds on information leakage through APIs.

The Security API she designs will be part of a whole system, and whole systems inevitably remain in a state of flux. The security architect will have chosen how much flux to pass down. He has the option of creating a point of stability where 'designing Security APIs right' becomes a solved problem. Alternatively he may pass down so much flux that uncertainty is guaranteed, and Security API designers must resign themselves to the traditional arms race between attack and defence that so many software products have to fight.

10.3 Closing Remark

This thesis does not have many of the answers to good API design, but it does constitute a starting point for understanding Security APIs. Once this understanding is absorbed we will really have a chance to build secure APIs and use them to change the way we do computing, be it for better, or for worse.

Chapter 11

Glossary

4753 IBM Cryptographic Adaptor packing 4755 HSM for connection to IBM mainframe. A stripped down PC with an 4755 or 4758 and an IBM channel interface card. May provide additional physical security too.

4754 IBM Security Interface Unit. Provides security of access to PS/2s and PC. Comprises a reader for your IBM Personal Security Card, 12-key keypad, and gives access to optional signature verification feature on the 4755. Can also be used to set or change 4753 functions. Part of TSS

4755 IBM Hardware Security Module, built around Intel type chips. Provides DES support and has tamper-resistant packaging. Part of TSS

4758 IBM's PCI Cryptographic Coprocessor. Programmable tamper-resistant PCI bus card supporting DES, RSA and more.

API Application Programmer Interface

Atalla An HSM manufacturer

ATM Automated Teller Machine. A synonym for 'cash machine'.

CA Certification Authority

CCA Common Cryptographic Architecture – IBM's financial security API

CLEF Commercial Licenced Evaluation Facility

Conformance Profile PKCS#11 document describing what features of PKCS#11 it implements, and what additional semantics it adds

Cryptoprocessor Synonym for HSM

CVV Card Verification Value. A secret field written on the magstripe calculated from the PAN, to make it harder to forge magstripe cards from the account number alone.

Decimalisation Table A table used for converting PINs produced in hexadecimal (i.e. containing digits A-F) down to containing only 0-9. The typical

	table maps `0-F` to `0123456789012345`. An example decimalisation is `3BA2` becoming `3102`.
DRM	Digital Rights Management. The restriction of flow of digital information, in particular, entertainment media.
EFT	Electronic Funds Transfer
EMV	Europay, Mastercard, VISA. EMV is the name for the new chip and PIN electronic payments scheme that is currently replacing magstripes on credit and debit cards.
Eracom	Australian HSM manufacturer
FIPS	Federal Information Processing Standard – an American IT standard produced by NIST
HSM	Hardware Security Module
IRM	Information Rights Management. A subcategory of Digital Rights Management.
KEK	Key Encrypting Key. A transport key normally established by manual transfer of key components, which other keys are encrypted under (IBM terminology).
MAC	Message Authentication Code. A *MAC Key* is a key held by an HSM permissions set such that it can only be used for calculating or verifying MACs on messages.
MIM	Meet-In-the-Middle – An attack exploiting collisions between items in a data set, as in the birthday paradox
Monotonic	A monotonic API is one where a transaction performed with certain inputs can be repeated at any time by providing the same inputs, and will result in the same output. (During operation of the API, the set of valid outputs increases in size monotonically)
nCipher	An HSM manufacturer
nCore	nCipher's main API, on top of which other applications are built
NIST	National Institute of Standards and Technology
NGSCB	Next Generation Secure Computing Base. New name for Microsoft's 'Palladium' initiative
OEM	Other Equipment Manufacturer. In a Security API context, a company that takes a custom programmable HSM and builds an API for it.
PAL	Permissive Action Link. A Nuclear weapon authorisation system, also referred to as a Prescribed Action Link.
Palladium	Microsoft's Trusted Computing initiative, designed to allow application developers to shield code and data from each other, and malicious parties.
PAN	Primary Account Number
PIN	Personal Identification Number
PIN Mailer	Special tamper-evident stationery for sending PINs through the post to customers.

PKCS#11	Public Key Cryptography Standard #11 – Standardised HSM API developed by RSA.
PKI	Public Key Infrastructure
PMK	PIN Master Key (aka PIN Derivation Key, PIN Generation Key) – key used to generate a customer's PIN from her account number
POS	Point of Sale (ie. a retail outlet)
Prism	An HSM manufacturer
Racal SPS	Racal Secure Payments Systems – one of the most successful financial HSM manufacturers
RBAC	Role Based Access Control. In this thesis, probably referring to IBM's 4758 CCA RBAC system.
RG7000	Highly popular banking HSM produced by Racal (now owned by Thales)
SSL	Secure Sockets Layer – An encryption protocol at TCP/IP level, commonly used to secure HTTP
Thales	An HSM manufacturer (originally called Racal)
TLA	Three Letter Acronym
TMK	Terminal Master Key – the key at the top of the key hierarchy inside an ATM machine. Also found lower down the hierarchy in HSMs at bank operations centres.
Transaction	A command which forms part of the Security API. This naming is intended to highlight the atomicity of the command – manipulation of the HSM state cannot occur in smaller steps than one transaction.
Transaction Set	The set of commands available to the user of a Security API
TRSM	Tamper-Resistant Security Module – synonym for HSM (Prism terminology).
TSS	Transaction Security Services (IBM mainframe financial transaction set)
UDX	User Defined eXtension. IBM's toolkit for adding custom transactions to the CCA.
Velocity Checking	Intrusion detection for possible fraud against a bank account, done by measuring the rate of withdrawal of cash over time.
VISA	A large payments service provider.
XOR	Bitwise Exclusive-OR
Zaxus	Racal SPS's new brand name, that lasted for only a year or so before they were bought by Thales.
ZCMK	Zone Control Master Key. VSM speak for a communications key held between to member banks of an ATM network. Identical to a ZMK.
ZMK	Zone Master Key. Racal speak for a ZCMK.

Bibliography

[1] R. Anderson, "The Correctness of Crypto Transaction Sets", 8th International Workshop on Security Protocols, Cambridge, UK, April 2000

[2] R. Anderson, "Security Engineering – a Guide to Building Dependable Distributed Systems", Wiley (2001) ISBN 0-471-38922-6

[3] R. Anderson, "Why Cryptosystems Fail" in Communications of the ACM vol 37 no 11 (November 1994) pp 32-40; earlier version at `http://www.cl.cam.ac.uk/users/rja14/wcf.html`

[4] R. Anderson, M. Bond, "Protocol Analysis, Composability and Computation", Computer Systems: Papers for Roger Needham, Jan 2003

[5] R. Anderson, S. Bezuidenhoudt, "On the Reliability of Electronic Payment Systems", in IEEE Transactions on Software Engineering vol 22 no 5 (May 1996) pp 294-301; `http://www.cl.cam.ac.uk/ftp/users/rja14/meters.ps.gz`

[6] RJ Anderson, MG Kuhn, "Low Cost Attacks on Tamper Resistant Devices", in Security Protocols (Proceedings of the 5th International Workshop (1997) Springer LNCS vol 1361 pp 125–136

[7] G. Bella, F.Massacci, L.Paulson, "An overview of the verification of SET", International Journal of Information Security

[8] M. Bond, "Attacks on Cryptoprocessor Transaction Sets", CHES 2001, Springer LNCS 2162, pp. 220-234

[9] M. Bond, R. Anderson, "API-Level Attacks on Embedded Systems", IEEE Computer, Oct 2001, Vol 34 No. 10, pp. 67-75

[10] M. Bond, P. Zielinski, "Decimalisation Table Attacks for PIN Cracking", University of Cambridge Computer Laboratory Technical Report no. 560

[11] M. Burrows, M.Abadi, R. Needham, "A Logic of Authentication", ACM Transactions on Computer Systems, 1990, pp. 18-36

[12] G. Campi, "Thermal Simulations Applied to Embedded Cryptographic Coprocessor Devices" `http://www.flotherm.com/technical_papers/t278.pdf`

[13] R. Clayton, M. Bond, "Experience Using a Low-Cost FPGA Design to Crack DES Keys", CHES Workshop 2002, San Francisco, Springer LNCS 2523, pp. 579-592

[14] J. Clulow, "On the Security of PKCS#11", CHES Workshop 2003, Cologne, Germany, LNCS 2779 pp. 411-425

[15] J. Clulow, "The Design and Analysis of Cryptographic APIs for Security Devices", MSc Thesis, University of Natal, SA

[16] IBM 4758 PCI Cryptographic Coprocessor, CCA Basic Services Reference And Guide, Release 1.31 for the IBM 4758-001

[17] D Davis, "Compliance Defects in Public-Key Cryptography", Sixth Usenix Security Symposium Proceedings, July 1996, pp. 171-178

[18] Y. Desmedt, "An Exhaustive Key Search Machine Breaking One Million DES Keys", Eurocrypt, 1987

[19] W. Diffie and M. Hellman, "Exhaustive cryptanalysis of the NBS Data Encryption Standard" , Computer vol.10 no.6 (June 1977) pp. 74-84.

[20] Diners Club SA Pty. versus A. Singh and V.Singh, March 2000-May 2003, High Court of South Africa, Durban, SA

[21] Electronic Frontier Foundation, "Cracking DES : Secrets of Encryption Research, Wiretap Politics & Chip Design", O'Reilly. (May 1998)

[22] "Security Requirements for Cryptographic Modules" Federal Information Processing Standards 140-1

[23] IBM Comment on 'A Chosen Key Difference Attack on Control Vectors', Jan 2000, available at `http://www.cl.cam.ac.uk/~mkb23/research/CVDif-Response.pdf`

[24] IBM "Update on CCA DES Key-Management" `http://www-3.ibm.com/security/cryptocards/html/ccaupdate.shtml`

[25] IBM 3614 Consumer Transaction Facility Implementation Planning Guide, IBM document ZZ20-3789-1, Second edition, December 1977

[26] IBM, 'IBM 4758 PCI Cryptographic Coprocessor – CCA Basic Services Reference and Guide, Release 1.31 for the IBM 4758-001', available through `http://www.ibm.com/security/cryptocards/`

[27] IBM Inc.: Update on CCA DES Key-Management. (Nov 2001) `http://www-3.ibm.com/security/cryptocards/html/ccaupdate.shtml`

[28] IBM Inc.: CCA Version 2.41. (5 Feb 2002) `http://www-3.ibm.com/security/cryptocards/html/release241.shtml`

[29] IBM Inc.: Version history of CCA Version 2.41, IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide for the IBM 4758-002. IBM, pg xv (Feb 2002)

[30] "IBM Enhanced Media Management System", `http://www-306.ibm.com/software/data/emms/`

[31] M. Kuhn, "Probability Theory for Pickpockets – ec-PIN Guessing", available from http://www.cl.cam.ac.uk/~mgk25/

[32] D. Longley, S. Rigby, "An Automatic Search for Security Flaws in Key Management", Computers & Security, March 1992, vol 11, pp. 75-89

[33] S.M. Matyas, "Key Handling with Control Vectors", IBM Systems Journal v. 30 n. 2, 1991, p. 151-174

[34] S.M. Matyas, A.V. Le, D.G. Abraham, "A Key Management Scheme Based on Control Vectors", IBM Systems Journal v. 30 n. 2, 1991, pp. 175-191

[35] J.F Molinari, "Finite Element Simulation of Shaped Charges" http://pegasus.me.jhu.edu/~molinari/Projects/Shape/SLIDE-1.html

[36] Robinson, J.A. "A machine-oriented logic based on the resolution principle", 1965, Journal of the ACM, 12(1): 23-41.

[37] S. Skorobogotov, "Low temperature data remanence in static RAM", University of Cambridge Computer Laboratory Technical Report TR-536 http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-536.pdf

[38] F.Stajano, R. Anderson, "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks", 1999, 7th International Workshop on Security Protocols, Cambridge, UK

[39] C. Weidenbach, "The Theory of SPASS Version 2.0"

[40] Altera Inc.: Excalibur Development Kit, featuring NIOS. http://www.altera.com/products/devkits/altera/kit-nios.html

[41] http://www.keyghost.com

[42] "Algorithmic Research" PrivateSafe Card Reader http://www.arx.com/products/privatesafe.html

[43] Formal Systems, the manufacturers of FDR http://fsel.com

[44] http://www.research.att.com/~smb/nsam-160/pal.html

[45] http://www.celestica.com

[46] http://www.nextpage.com

[47] http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/default.mspx

[48] http://www.infraworks.com

[49] http://www.baltimore.com

[50] http://www-3.ibm.com/security/cryptocards/html/overcustom.shtml

[51] http://www.cl.cam.ac.uk/~rnc1/descrack/

[52] http://spass.mpi-sb.mpg.de/

[53] http://www.cl.cam.ac.uk/~rnc1/descrack/sums.html

[54] http://www.nsa.gov/selinux

[55] http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html