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. The time-memory trade-off has of course been described several decades ago, for example 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 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.

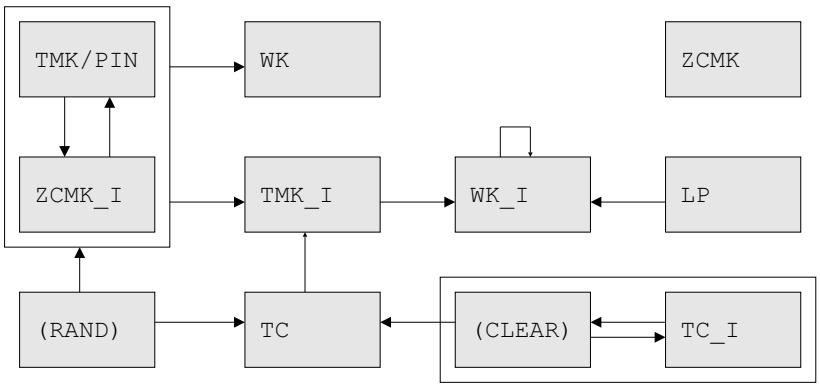

Figure 7.3: Example type system from the VSM

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.

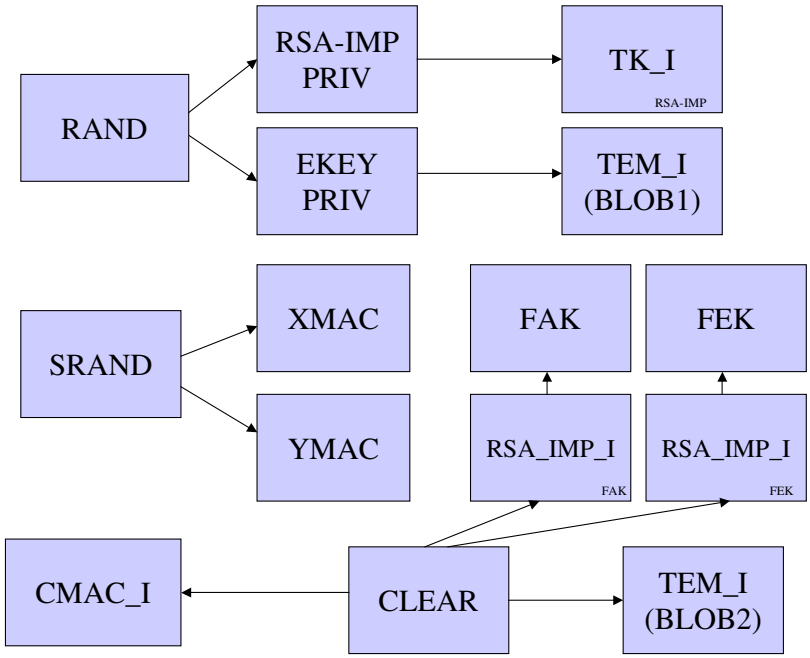

Figure 7.4: Example type system from real world application

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.

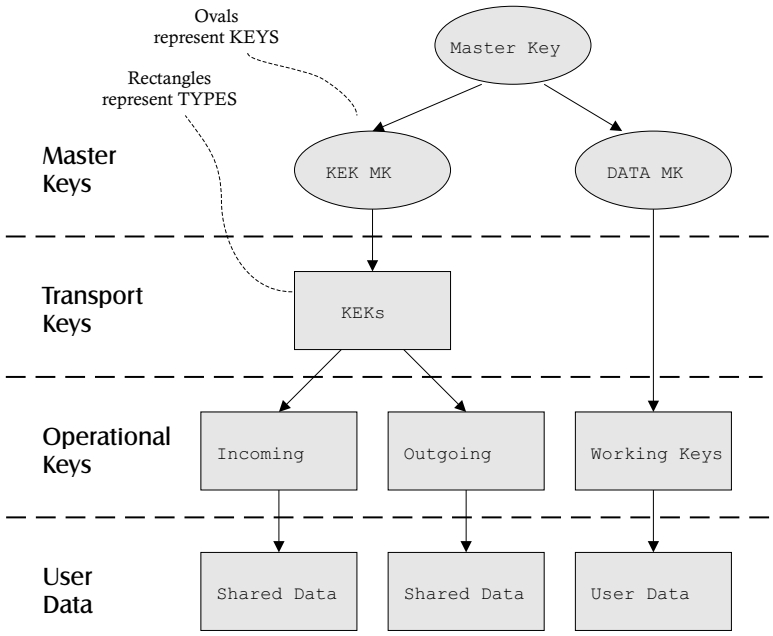

Figure 7.5: An example key hierarchy

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.

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.

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

	(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

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
```

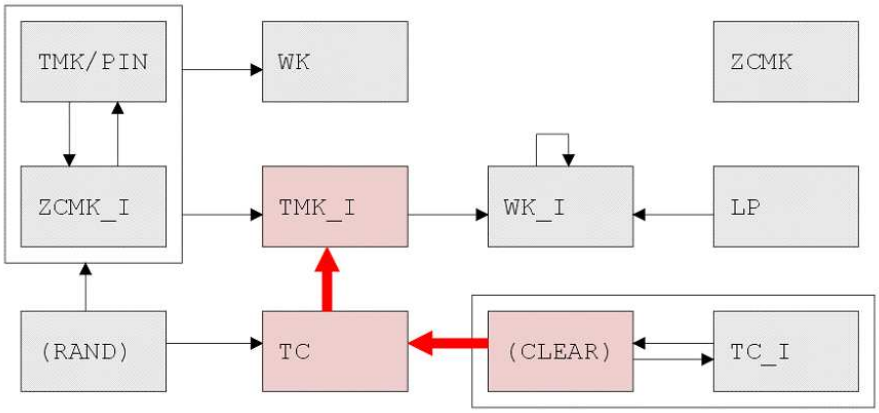

Figure 7.7: Type diagram for VSM with attack path highlighted

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

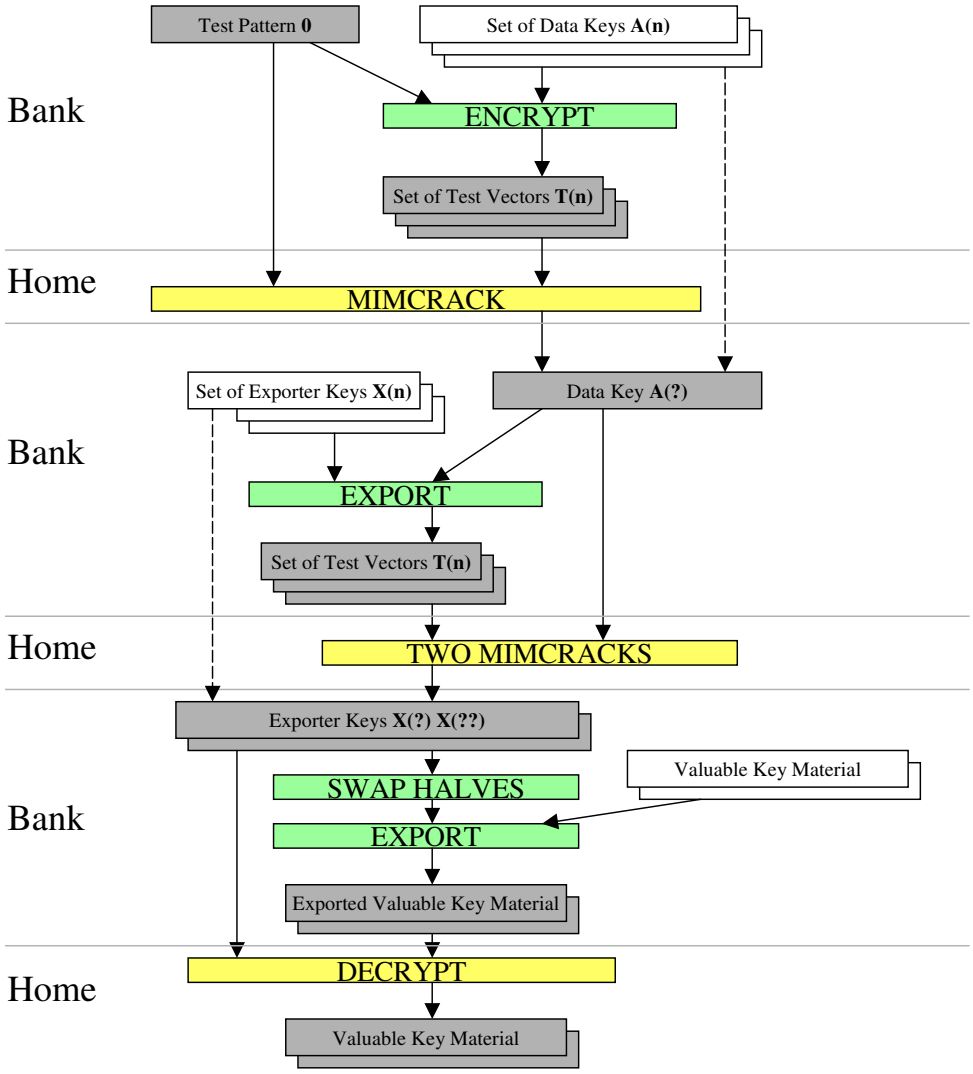

Figure 7.8: Standard implementation of attack on 4758 CCA

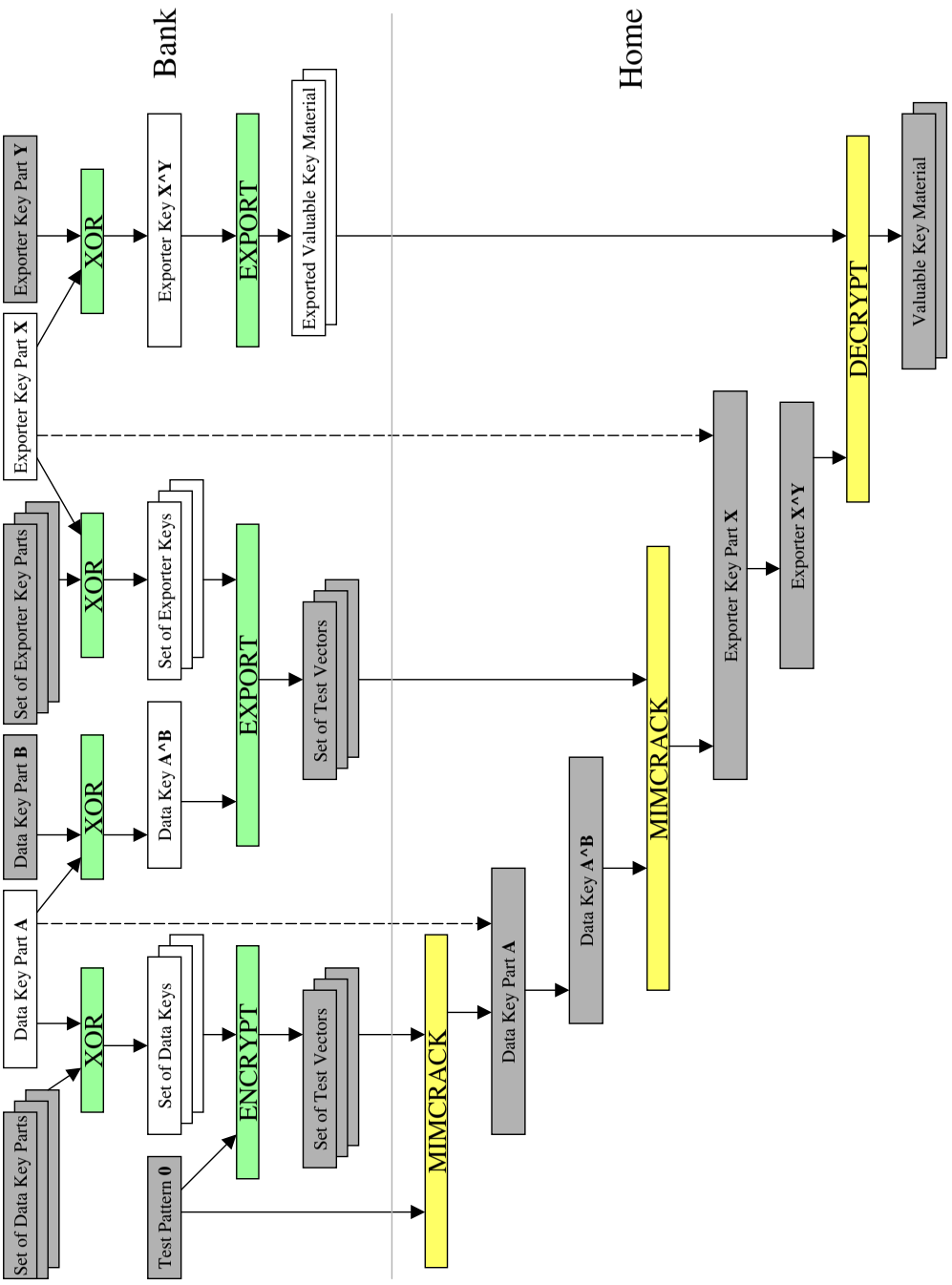

Figure 7.9: Optimised implementation of attack on 4758 CCA

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.

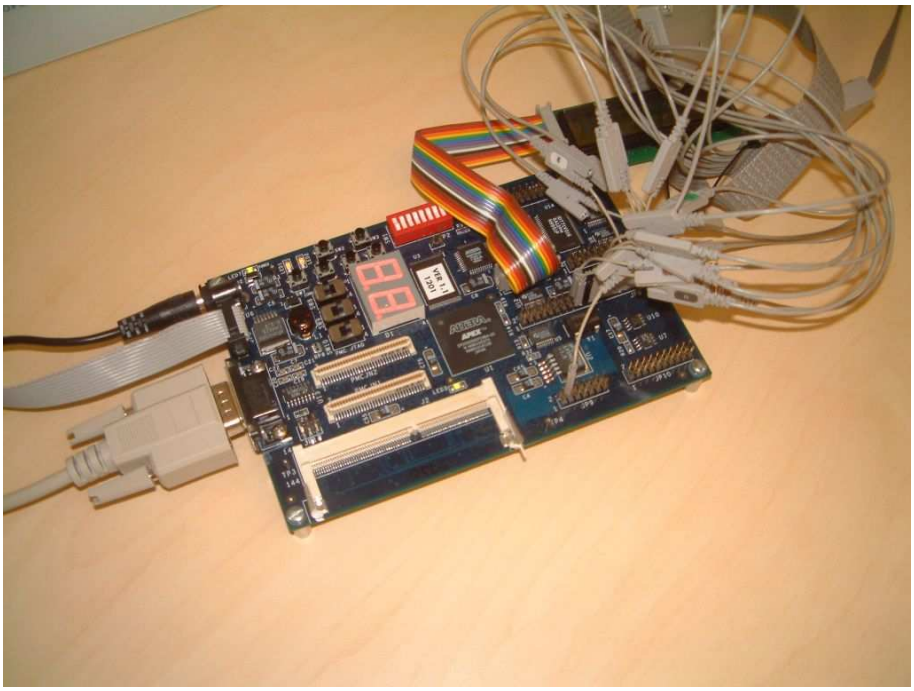

Figure 7.10: The NIOS Evaluation board running the DES cracker

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

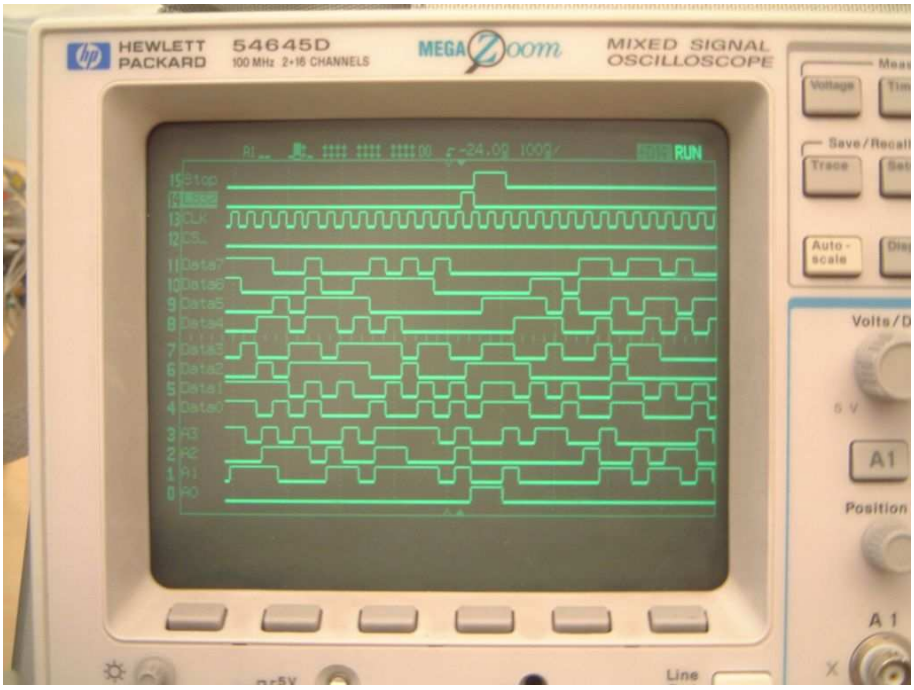

Figure 7.11: The DES cracker actually running

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

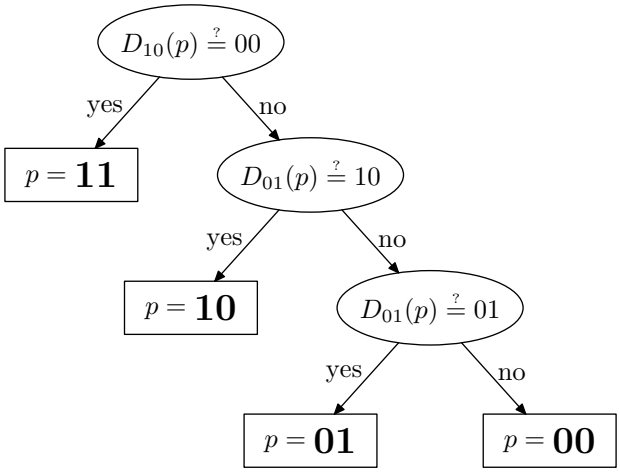

Figure 7.14: The search tree for the initial scheme. D_{xy} denotes the decimalisation table that maps $0 \to x$ and $1 \to y$.

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.

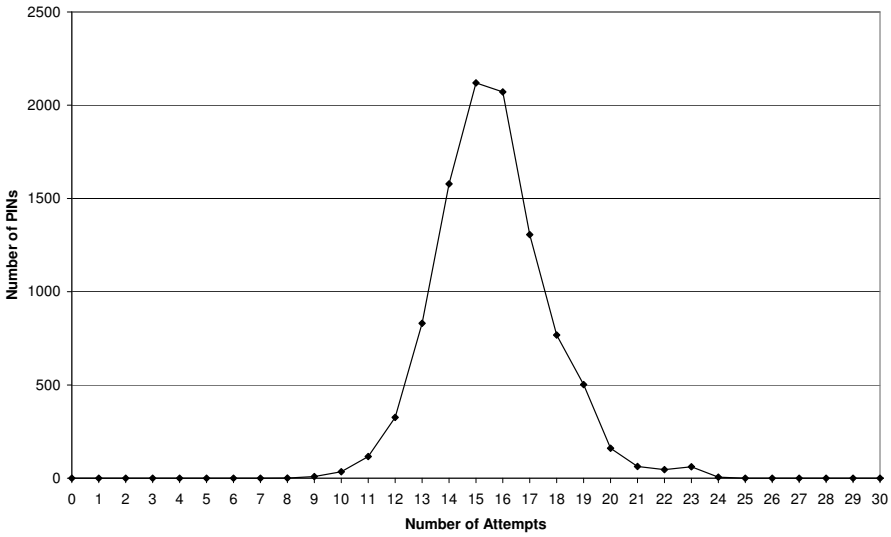

Figure 7.17: Distribution of guesses required using adaptive algorithm

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.

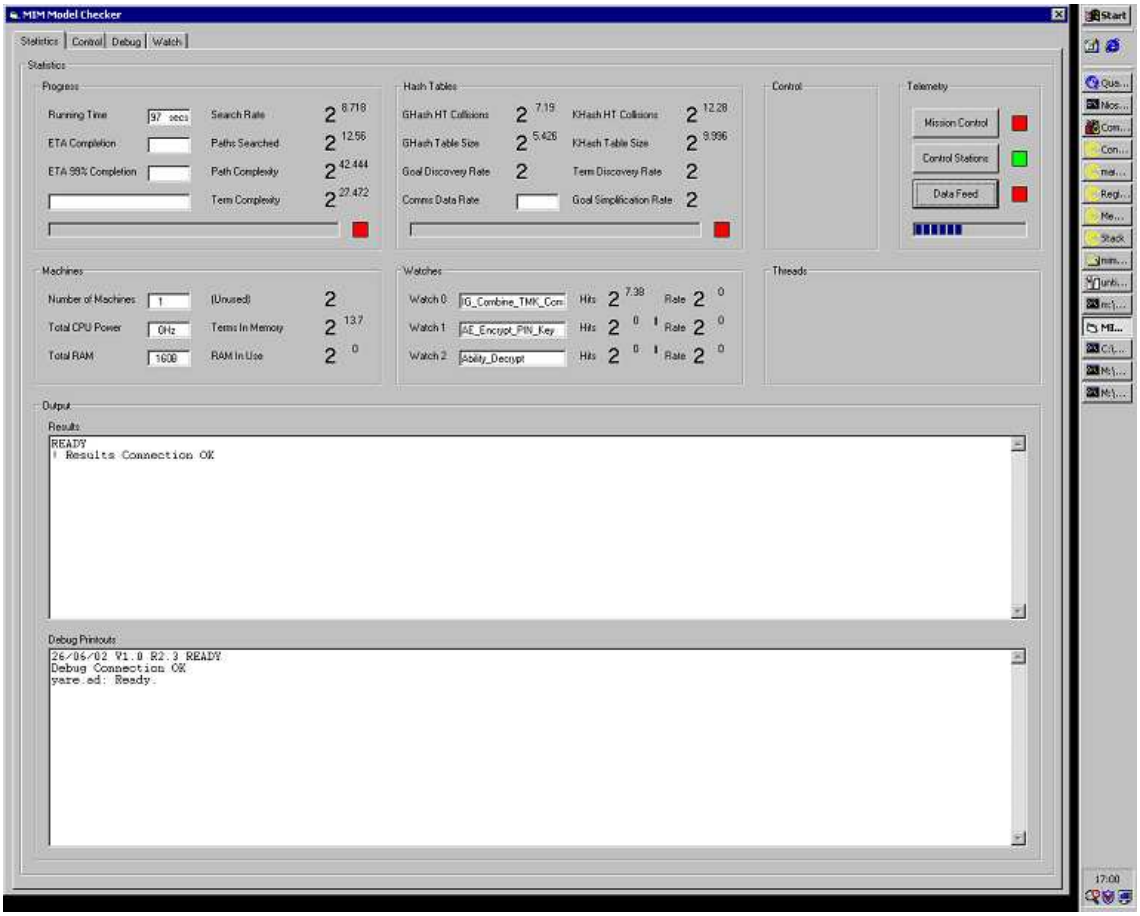

Figure 7.23: The MIMsearch statistics display

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

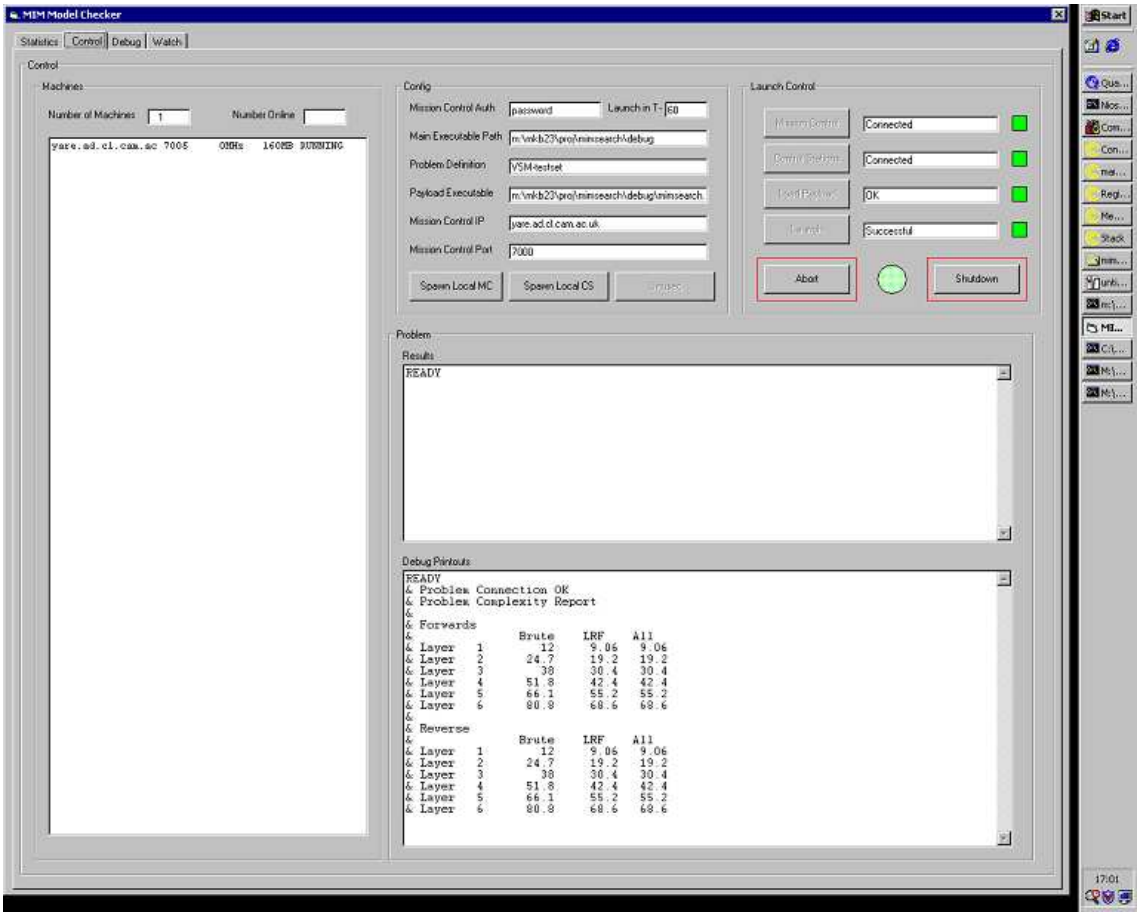

Figure 7.24: The MIMsearch control interface

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.

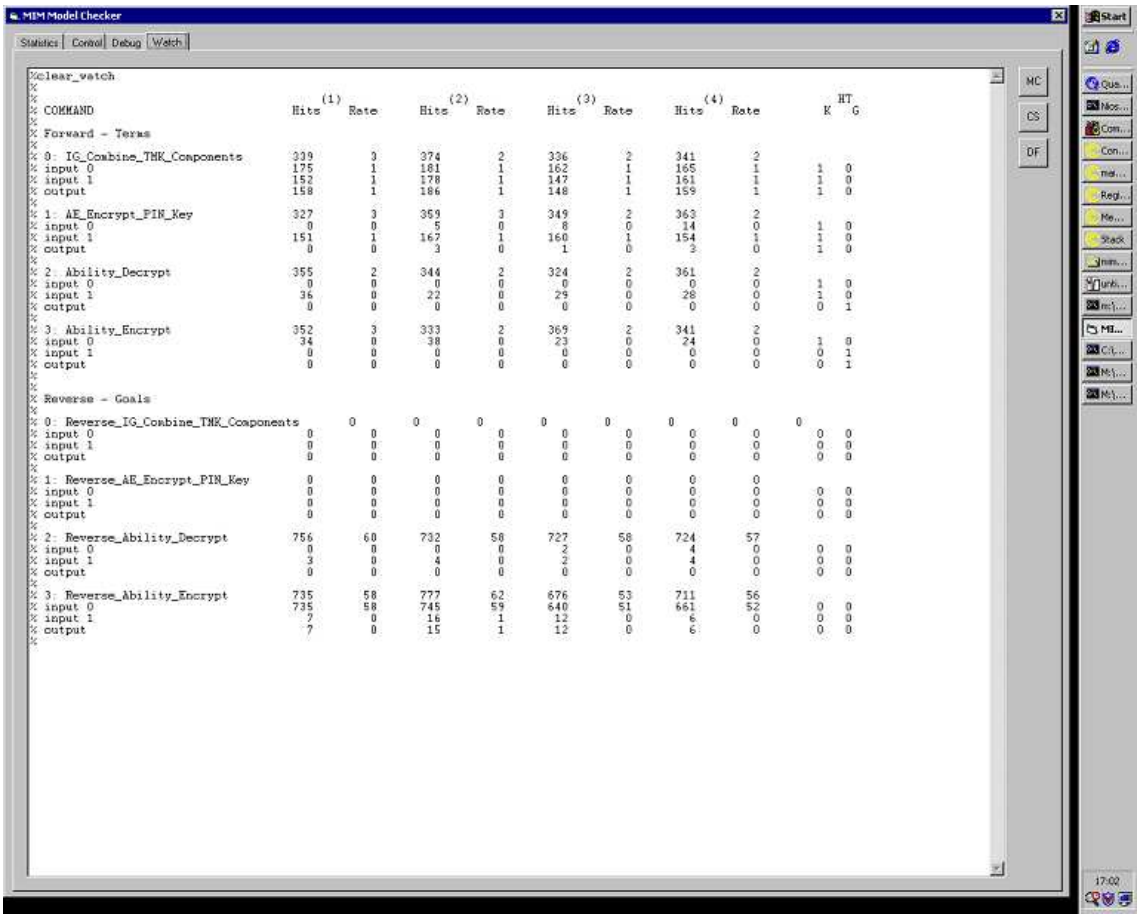

Figure 7.25: The MIMsearch ‘Watch’ display

Chapter 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.