Attacks on Cryptoprocessor Transaction Sets

Mike Bond

Computer Laboratory, University of Cambridge,
Pembroke Street, Cambridge, CB2 3QG, UK
Mike.Bond@cl.cam.ac.uk

Abstract. Attacks are presented on the IBM 4758 CCA and the Visa Security Module. Two new attack principles are demonstrated. Related key attacks use known or chosen differences between two cryptographic keys. Data protected with one key can then be abused by manipulation using the other key. Meet in the middle attacks work by generating a large number of unknown keys of the same type, thus reducing the key space that must be searched to discover the value of one of the keys in the type. Design heuristics are presented to avoid these attacks and other common errors.

1 Introduction

A cryptoprocessor is a tamper-resistant processor designed to manage cryptographic keys and data in high-risk situations. The concept of a cryptoprocessor arose because conventional operating systems are too bug-ridden and computers too physically insecure to be trusted with information of high value. A normal microprocessor is enclosed within a tamper-resistant environment, so that sensitive information can only be altered or released through a tightly defined software interface – a *transaction* set. In combination with *access control*, the transaction set should prevent abuse of the sensitive information. However, as the functionality and flexibility of transaction sets have been pushed up by manufacturers and clients, this extra complexity has made bugs in transaction sets inevitable.

Sections 2 and 3 of this paper give an overview of cryptoprocessors in the context of four important architectural principles, and then describe the new vulnerabilities in a generalised way. Sections 4 and 5 introduce attacks on two widely fielded cryptoprocessors – the IBM 4758, and the Visa Security Module. Finally, some straightforward design heuristics are suggested that, whilst not guaranteeing the security of a transaction set, will at least stop the same mistakes being made over again.

2 Tour of a Cryptoprocessor

A cryptoprocessor's interface to the world is its *transaction set* – a group of commands supported by the processor to manipulate and manage sensitive information, usually cryptographic keys. Users are limited to the subset of the

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 220–234, 2001.

transaction set which reflects their needs using an *access control* system. The intended inputs and outputs of commands in a transaction set are described in terms of a *type system*, which describes the content of each type, and then assigns a type to each input and output of the commands. Keys tend to be stored in a *hierarchical structure* so that large amounts of information can be shared by securely sharing only a single piece of information at the base of a branch in the hierarchy.

2.1 Transaction Set Fundamentals

- *User commands* are the bulk of the cryptoprocessor's workload. The commands allow data to be processed (e.g. encrypted, decrypted, MACs generated/verified) using keys whose values are retained within the tamper-proof environment, remaining unknown to the user. The user is thus restricted to performing actions with these keys online, where procedural controls can be enforced. Application-specific commands may also exist, which manipulate encrypted inputs and return an encrypted output or maybe a simple return code (e.g. a yes/no answer to whether an entered PIN matched the correct value for an account number, without revealing either value).
- *Key Management commands* give users the ability to rearrange the key structure. Import and export commands will allow extraction of keys from the structure for sharing with other processors or environments, and commands to build up keys from multiple parts may be available to support dual control policies.
- *Administration commands* are highly dependent on implementation details, but would generally include commands for management of particularly sensitive high-level keys, modification of the access rights for other users, and output of clear PIN numbers in financial systems.

2.2 Access Control

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

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

More versatile access control systems will maintain a record of which transactions each user can access, or a role-based approach to permit easier restructuring as the job of an individual real-world user changes, either in the long term or

through the course of a working day. In circumstances where there are multiple levels of authorisation, the existence of a *'trusted path'* to users issuing special commands becomes important. Without using a secured session or physical access port separation, it would be easy for an unauthorised person to insert commands of their own into this session to extract sensitive information under the very nose of the authorised user.

2.3 Key Hierarchies

Storage of large numbers of keys becomes necessary when enforcing protection between multiple users, and serves to limit damage if one is compromised. The common storage method is 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. Alternate roles include inferring the *type* of a key from its position in the hierarchy, or increasing the storage capacity of the cryptoprocessor by keeping only the top-level keys within the tamper-proofed environment, and storing the remainder externally, with each lower level encrypted using the appropriate key from the level above.

Figure 1 shows a common model with three layers of keys:

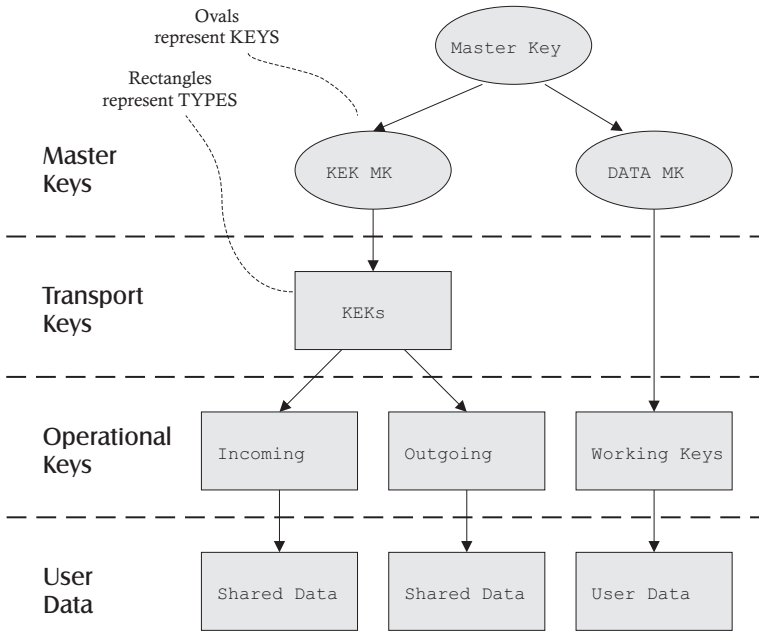

Fig. 1. An example key hierarchy

The top layer contains *'master keys'* which are never revealed outside the cryptoprocessor, 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 cryptoprocessors extends to an even lower layer, containing data encrypted with the operational keys.

2.4 Key Typing Systems

Assigning type information to keys is necessary for fine grain access control to the transaction set. This is because many transactions have the same core functionality, and without key typing an attacker could achieve the equivalent of execution of a transaction he doesn't have permission for by using an equivalent permitted transaction (e.g. calculating 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.

An important example is the type distinction between communications data keys and PIN processing keys in financial systems. Customer PIN numbers are calculated by encrypting the account number with a PIN derivation key, thus commands using these keys are carefully controlled. However, if PIN keys and data keys were indistinguishable in type, any user with access to data manipulation transactions could calculate the PIN numbers from accounts: both employ the same DES or triple-DES (3DES) encryption algorithm to achieve their purpose.

IBM's financial products use the Common Cryptographic Architecture (CCA) – a standardised transaction set. The CCA name for the type information of a key is a control vector. Control vectors are bound to encrypted keys by XORing the control vector with the key used to encrypt, and including an unprotected copy for reference (1). The control vector is simply a bitpattern chosen to denote a particular type. If a naive attacker changes the clear copy of the control vector (i.e. the claimed key type), when the key is used, the cryptoprocessor's decryption operation should simply produce garbage (2). The implementation details are in 'Key Handling with Control Vectors' [2], and 'A Key Management Scheme Based on Control Vectors' [3].

$$(1) \qquad E_{Km \oplus CV}(KEY) \ , \ CV$$

$$(2) \qquad D_{Km \oplus CV_M OD}(E_{Km \oplus CV}(KEY)) \neq KEY$$

3 The Attacker's Toolkit

The attacks in sections 4 and 5 are presented as combinations of attack *'building blocks'*. This section describes new building blocks, some intuitively dangerous in their own right, and others which only reap maximum damage in combination. The full set includes reapplications of existing techniques from other fields, and is augmented by the usual tools and methods available to an attacker (e.g. brute force search, cryptanalysis).

3.1 The Meet in the Middle Attack

Users can normally select which key is used to protect the output of a command, provided it is of the correct type. The flexibility gained from specification using the type system is at the price of risking catastrophic failure if the value of even just one key within a type is discovered – select the cracked key, and the command output will be decipherable. The meet in the middle attack is just common sense statistics: if you only need to crack a single key within a type to be successful, the more keys that you attack in parallel, the shorter the average time it takes to discover one of them using a brute force search.

The attacker first generates a large number of keys. 2^{16} (65,536) is a sensible target: somewhere between a minute and an hour's work for the cryptoprocessors 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 will now take slightly longer, but there will be many less to check. The observation at the heart of the attack is that 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.

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 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 cryptoprocessor attack the key space from opposite sides, and the effort expended by each meets somewhere in the middle.

3.2 Related Key Attacks

Allowing related keys to exist within a cryptoprocessor is dangerous, because it causes 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 3.1).

Keys with a *chosen* relationship can be even more dangerous because some architectures 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 cryptoprocessors 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. At generation, the key parts would be given to separate couriers and 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. Some architectures allow a chosen value to be XORed with any key at any time.

3.3 Unauthorised Type-Casting

The 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 KEK could be retyped as a data key, keys protected with it could be exported in the clear using a standard data decipherment transaction).

Certain type casts are only 'unauthorised' in so far as that the designers never intended them to be possible. In some architectures it may even be difficult to tell whether or not an opportunity to type cast is a bug or a feature! Indeed, IBM describes a method in the manual for their 4758 CCA [1] to convert between key types during import to allow interoperability 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.

Cryptoprocessors which do not maintain internal state about their key structure have difficulties deleting keys. Once an encrypted version of a key has left the cryptoprocessor it cannot prevent an attacker storing his own copy for later re-introduction 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.

3.4 Poor Key-Half Binding

Cryptographic keys get split into distinct parts, when the block length of the algorithm protecting them is shorter than the key length. 3DES is particularly common, and has a 112 bit key made up from two 56 bit single DES keys. When the association between the halves of keys is not kept, the security of the key is crippled. A number of cryptoprocessors allow the attacker to manipulate the actual keys simply by manipulating their encrypted versions in the desired manner. Known or chosen key halves could 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 3.2.

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.

3.5 Conjuring Keys from Nowhere

Cryptoprocessor designs which store encrypted keys outside the tamper-proof environment 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 cryptoprocessors used this technique to generate keys (keys with bad parity were automatically corrected). Most 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 3.4).

4 Attacks on the NSM (A Visa Security Module Clone)

The Visa Security Module (VSM) is a cryptoprocessor with a concise, focused transaction set, designed to protect PIN numbers transmitted over private bank ATM networks, and on the inter-bank link system supported by VISA. It was designed in the early eighties, and the NSM is a software compatible clone [5].

The VSM has two authorisation states (user and authorised) enabled using passwords. The NSM improves on this by splitting the authorised state in two – *supervisor* and *administrator*, selected by two key switches on the casing. The user state gives access to transactions to verify customers PINs in a number of ways, and to translate them between encryption keys to allow forwarding of requests to and from other banks in the network. The user state also contains transactions to permit key generation and update for session keys. The supervisor state is only enabled upon special procedural controls and enables transactions to allow extraction of PIN numbers to a printer connected to a dedicated port on the cryptoprocessor. Administrator authorisation allows generation of high-level master keys, and is rarely used. It recognises nine distinct types in total, shown by rectangles in figure 2. The ovals represent individual keys.

At the top of the key hierarchy are five 3DES master keys, stored in registers within the cryptoprocessor. These protect the five fundamental types, and all other types are likewise inferred implicitly from a key's position within the

hierarchy. Apart from the 3DES master keys, all other keys are Single DES, and so must be changed regularly. The PIN derivation keys are an exception to the regular changes, but are afforded extra protection by measures to ensure that known plaintext/ciphertext pairs are not available to an attacker.

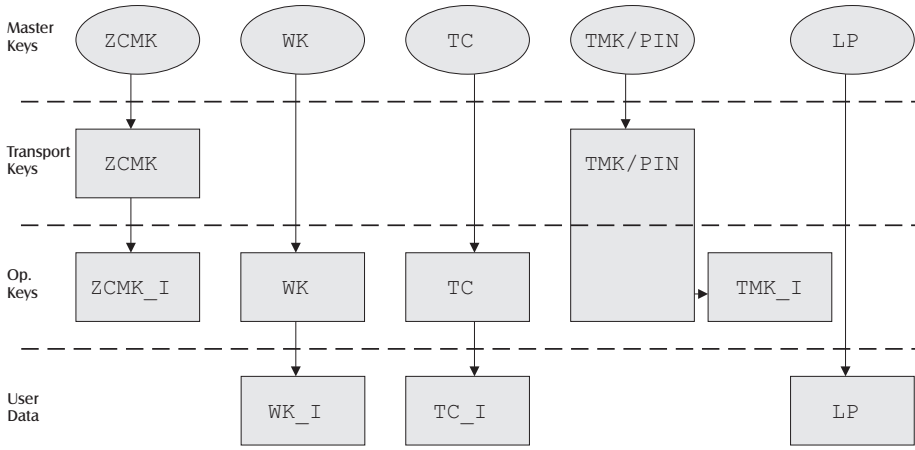

Fig. 2. The VSM key hierarchy

Terminal Master Keys (`TMK`s) are copies of those used in ATMs, available to the VSM so that it can prepare keysets allowing the ATMs to verify PINs themselves. `PIN` keys are used to convert account numbers into PIN numbers. The 4 digit PINs entered by customers are calculated from the result of encrypting the account number with the PIN key, using a publicly available algorithm. `TMK`s and `PIN` keys occupy the same type in the VSM, even though they are conceptually different. *Zone Control Master Keys* (`ZCMK`s) are keys to be shared with other banking networks, used to protect the exchange of working keys. *Working Keys* (`WK`s) are used to protect trial PINs that customers have entered, whilst they travel through the network on the way to the correct bank for verification, and are not used for intra-bank communications. *Terminal Communications keys* (`TC`s) are for protecting control information going to and from ATMs. Note that all *keys* sent to an ATM are protected with a `TMK`. Figure 3 shows the commands available to the normal user as lines between types. Two extra 'types' are shown: `(RAND)` and `(CLEAR)`. The `(RAND)` type can be thought of as a source of unknown random numbers, so lines emanating from it represent key generation transactions. `(CLEAR)` is a source of user chosen values. The notation `TYPE_I` is used to stand for information encrypted with a key of type `TYPE`.

4.1 VSM Compatibles – A Poor Type System Attack

The amalgamation of the `TMK` and `PIN` types is responsible for a number of weaknesses in the VSM. One possible attack is to enter an account number as

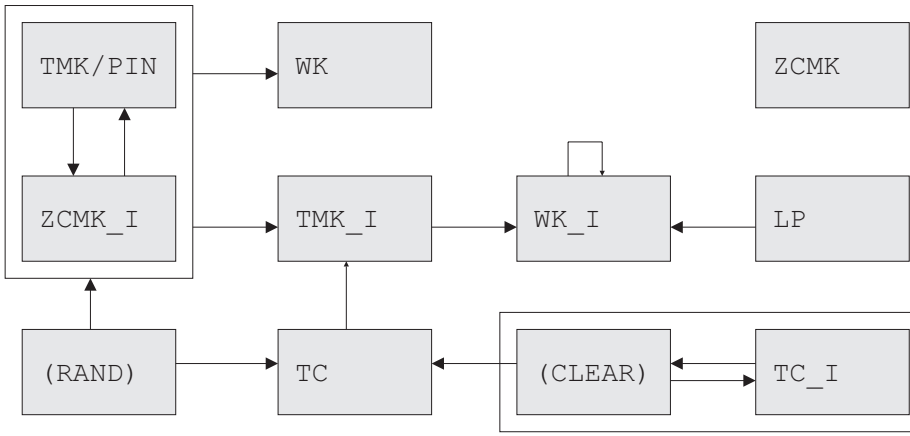

Fig. 3. The VSM type system

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` 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 `ACCNO`, the attack can be followed on the type transition diagram in figure 3, moving from `(CLEAR)` to `TC` (1), and finally to `TMK_I` (2).

$$(1) \qquad ACCNO \longrightarrow \{ACCNO\}TC \qquad (ACCNO \in CLEAR)$$
$$(2) \qquad \{ACCNO\}TC \longrightarrow \{ACCNO\}TMK\_I \qquad (TMK\_I = A\ PIN\ key)$$

Although the attack does not directly exploit any of the methods from section 3, it demonstrates the fragility of transaction sets, and is a good example of the characteristics of a broken transaction set when analysed in the context of *key hierarchies* and *type systems*.

4.2 VSM Compatibles – Meet in the Middle Attack

The meet in the middle attack can be used to compromise eight out of the nine types used by the VSM. 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. Indeed, it *cannot* properly impose restrictions on key generation because of the 'key conjuring' attack (section 3.5) which works with many cryptoprocessors which store keys externally.

The target type should be populated with at least 2^{16} keys, and a test vector encrypted under each. The dedicated 'encrypt test vector' command narrowly escapes compromising all type because the default test vector 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 3) 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 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 3).

5 Attacks on the IBM 4758 CCA

The Common Cryptographic Architecture (CCA) is a standardised transaction set which is implemented by the majority of IBM's financial security products. The 4758 is a PC-compatible cryptographic coprocessor which implements the CCA. Control over the transaction set is quite flexible: role-based access control is available, and the users communicate via trusted paths protected with 3DES session keys. The transaction set itself is large and complex, with all the typical transactions described in section 2.1, as well as many specialised commands to support financial PIN processing. The CCA stores nearly all keys in encrypted form outside the cryptoprocessor, with a single 168-bit master key *KM* at the root of its key hierarchy:

The CCA holds type information on keys using *control vectors*. A control vector is synonymous with a type, and is bound to encrypted keys by XORing

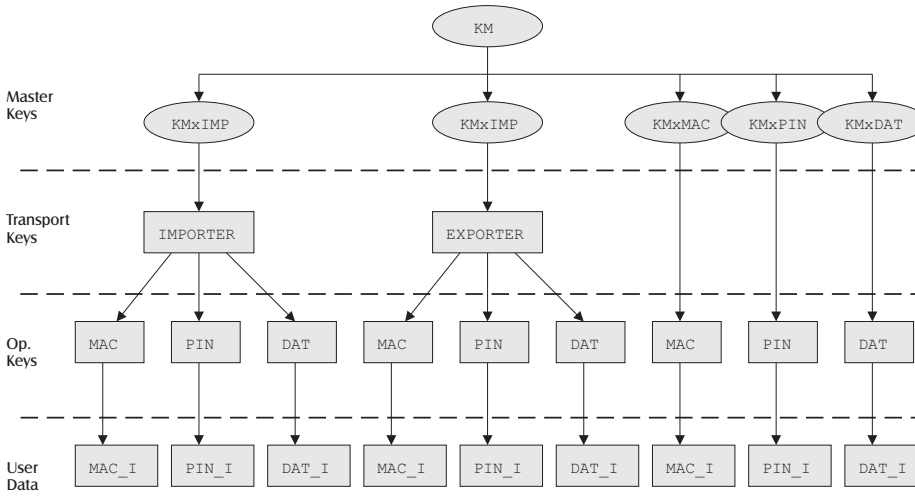

Fig. 4. The 4758 CCA key hierarchy

the control vector with the key used to encrypt, and including an unprotected copy for reference.

5.1 4758 CCA – Key Import Attack

One of the simplest attacks on the 4758 is to perform an unauthorised type cast using IBM's 'pre-exclusive-or' type casting method [1]. 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 cryptoprocessor is open to abuse. More subtle type changes are worthy of mention, 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 3.5). 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 cryptoprocessor 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)	$E_{KEK1 \oplus old\_CV}(KEY)$, old_CV
Import Process	(3)	$D_{KEK2 \oplus new\_CV}(E_{KEK1 \oplus old\_CV}(PKEY)) = PKEY$

A successful attack 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 [4], 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.

5.2 4758 CCA – Import/Export Loop Attack

The limitation of the key import attack described in 5.1 is that only keys sent from other cryptoprocessors are at risk from the attack, because these 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 cryptoprocessor, 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) $IMPORTER1 = RAND$
$IMPORTER2 = RAND \oplus (importer\_CV \oplus exporter\_CV)$
(2) $E_{IMPORTER1 \oplus importer\_CV}(KEKP)$
(3) $D_{IMPORTER2 \oplus exporter\_CV}(E_{IMPORTER1 \oplus importer\_CV}(KEKP)) = KEKP$

(4) $EXPORT\_CONVERT = KEKP$
(5) $IMPORT\_CONVERT1 = KEKP \oplus (source1\_CV \oplus dest1\_CV)$
$\cdots$
$IMPORT\_CONVERTn = KEKP \oplus (source1\_CV \oplus destn\_CV)$

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 [1], but this attack employs a relationship between keys known only to the attacker, so conventional audit fails.

5.3 4758 CCA – 3DES Key Binding Attack

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 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 permissions to make 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 5.1). 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.

6 Conclusions

The cryptoprocessors examined have disappointing dependency upon tight procedural controls in the operating environment – they have failed to realise the full potential of tamper-resistant enclosure. It is strange that the transaction sets of both simple, highly-specialised cryptoprocessors and flexible, complex cryptoprocessors have both been found vulnerable to an individual corrupt insider. Perhaps this is because in security the design rule 'keep it simple' collides with the need for explicitness. The complex systems fail to keep it simple, and the simple ones simplify too severely. The design heuristics presented below may go against the grain of the 'keep it simple' or 'be explicit' principles individually, but the best solution has to be a compromise. In the best case these heuristics go a long way to avoiding security pitfalls, and in the worst case, the heuristics at least reveal the areas in which compromises must be made.

6.1 Design Heuristics

– Known or chosen keys should not be allowed into the key hierarchy.

- Avoid related key sets. If you must have them, keep relationship secret to the cryptoprocessor, or generate them dynamically from a single key.
- Ensure there is a trusted path to the cryptoprocessor available for the issue of sensitive commands.
- Do not rely on key parity bits for integrity checking: the chance of accidental success is too high.
- Do not allow transactions to produce 'garbage': results with no clearly defined meaning when the inputs are invalid. This frustrates analysis.
- Keep access control as fine grain as possible: highly flexible transactions are dangerous without highly flexible access control for them.
- Avoid types whose roles cross hierarchical boundaries.
- If using encryption with short key lengths, limit membership levels of types to avoid the meet in the middle attack, or prevent test vector generation.
- Impose restrictions on key generation to limit the attackers options.
- Ensure that keys are 'atomic' : permitting manipulation of key parts is dangerous.
- Be explicit when generating your type system.
- Don't try to infer type information from a random number: ambiguity is inevitable.

6.2 Future Directions

The VSM and CCA architectures have been shown to be unsatisfactory, and a skeletal toolkit has been presented for analysing these shortcomings. Research awaiting publication includes the application of the new attack techniques to more transaction sets, and future research includes the enlargement of the analysis toolkit, and the long-term aim of designing a transaction set which is resistant to these modes of failure, and is well balanced between simplicity and explicitness.

Acknowledgements

The author wishes to thank (alphabetically) Ross Anderson, Richard Clayton, George Danezis and Larry Paulson for their assistance in verifying and understanding the consequences of these attacks. Work on the VSM was inspired by a talk given by Ross Anderson [7]. The research was conducted thanks to the generous funding of the UK Engineering and Physical Sciences Research Council (EPSRC).

References

1. IBM 4758 PCI Cryptographic Coprocessor, CCA Basic Services Reference And Guide, Release 1.31 for the IBM 4758-001
2. S.M. Matyas, 'Key Handling with Control Vectors', IBM Systems Journal v. 30 n. 2, 1991, p. 151-174

3. S.M. Matyas, A.V. Le, D.G. Abraham, 'A Key Management Scheme Based on Control Vectors', IBM Systems Journal v. 30 n. 2, 1991, pp. 175-191
4. IBM Comment on 'A Chosen Key Difference Attack on Control Vectors', Jan 2000
5. NSM Developers Manual, Computer Security Associates (Pty.) Ltd. , July 1990
6. 'Security Requirements for Cryptographic Modules' Federal Information Processing Standards 140-1
7. 'The Correctness of Crypto Transaction Sets' R. Anderson, April 2000