API-Level Attacks on Embedded Systems

Mike Bond Ross Anderson

2nd May 2001

Abstract

A whole new family of attacks has recently been discovered on the application programming interfaces (APIs) used by security processors. These extend and generalise a number of attacks already known on authentication protocols. The basic idea is that by presenting valid commands to the security processor, but in an unexpected sequence, it is possible to obtain results that break the security policy envisioned by its designer. Such attacks are economically important, as security processors are used to support a wide range of services, from automatic teller machines through pay-TV to prepayment utility metering. Designing APIs that resist such attacks is difficult, as a typical security processor needs a substantial command set with several dozen commands that allow it to service a number of external and internal protocols. The attacks are also scientifically interesting; preventing them may become an important new application area for formal methods and design verification tools generally.

1 Introduction

A large and growing number of embedded systems make use of security processors to distribute control, billing and metering among devices with intermittent or restricted online connectivity. The more obvious examples include:

- the smartcards used to personalise mobile phones and to manage subscribers to satellite-TV services;
- microcontrollers used as value counters in postal meters and in vending machines to prevent fraud by maintenance staff; and
- cryptographic processors used in networks of automatic teller machines (ATMs) and point-of-sale equipment to encipher customers' personal identification numbers (PINs).

Behind these visible applications there may also be several layers of back-end systems which must prevent fraud by distributors, network operators and other participants in the value chain.

A good example is given by the prepayment electricity meters used to sell electric power to students in halls of residence, in the third world, and to poor customers

in rich countries [4]. They are typical of the many systems that once used coin-operated vending, but have now switched to tokens such as magnetic cards or smartcards. The principle of operation is simple: the meter will supply a certain quantity of energy on receipt of an encrypted instruction, then interrupt the supply. The instructions are created in a token vending machine, which knows the secret key of each local meter. One of the design goals is to limit the loss if a vending machine is stolen or misused; this enables the supplier to entrust vending machines to marginal economic players, ranging from student unions to third-world village stores.

The common solution is to build the vending machine round a tamper-resistant cryptographic processor, which contains the meter keys and a value counter. The value counter enforces a credit limit; after that much electricity has been sold, the machine stops working until it is reloaded. This requires an encrypted message from a controller one step higher up the chain of control – it would typically be issued by the distributor after payment by the machine operator. If an attempt is made to tamper with the value counter, then the cryptographic keys should be erased so that the token vending machine will no longer work at all. Without these controls, fraud would be much easier, and the theft of a vending machine might compel the distributor to re-key all the meters within its vend area. There are other security processors all the way up the value chain, and the one at the top – in the headquarters of the power company – may be controlling payments of billions of dollars a year.

A similar arrangement can be found in networks of *automatic teller machines* (ATMs) and *point-of-sale* (POS) devices; a tamper-resistant processor called a *security module* contains the cryptographic keys used for communicating with terminal equipment and also for verifying PINs in the incoming transactions.

No matter what the application of the security processor, its API sits on the boundary between trusted and untrusted environments, and is the point where cryptography, protocols, access controls, and operating procedures must all come together to enforce its security policy. Thus, understanding threats to these APIs is essential.

2 Attacks on Cryptographic APIs

Our research into attacks on cryptographic APIs started off with an examination of ways in which banking security modules can be manipulated [3], and proceeded to a study of attacks on more general-purpose tamper-resistant processors [6]. Recently, we have also found attacks on security processors used in utility prepayment applications.

It will be convenient to explain these attacks in historical order, but the attacks on both old and new systems support a number of more general interests:

- First, there has been a good deal of work on verifying crypto protocols, which are typically sets of 3-5 transactions exchanged by two principals. But in many real systems, these techniques must be extended to the dozens or even hundreds of transactions supported by the actual cryptographic service provider (whether smartcard, cryptoprocessor, or software library).

- Second, designing a secure and robust API is a fundamental challenge, which has until recently been overlooked by both formal methods and software engineering researchers – the bulk of whose work has been on avoiding implementation errors in the API, or verifying the correspondence of the API implementation to its specification. So API design and verification looks set to be the next basic research challenge.
- Third, many of the things that go wrong with secure systems happen at an interface between two or more kinds of protection mechanism. Crypto protocol failures tangle up the boundary between cryptography and access control; operating system security failures (and limitations) mean that applications often cannot exploit the protection features supported by a processor. In the same way, design flaws in crypto APIs occur at the even more complex interface between crypto, protocols, operating system access controls and specialist services such as value counters.
- Finally, a tamper-resistant device can be considered as just a high-quality implementation of an object that can only be invoked using its official methods, and whose internal variables remain inaccessible. Given that the object-oriented programming model is becoming popular, there may be more general lessons to be learned for robust programming.

So learning how to design security APIs properly is important, and especially so if we are to realise the potential of systems that distribute trust across a heterogeneous set of processes. It may thus be fundamental for large scale embedded systems. Mistakes can be horrendously expensive to rectify afterwards; once a system is as well entrenched as ATMs are, with over 600,000 devices from over a dozen vendors operated worldwide by perhaps 20,000 banks, changes may be next to impossible. Even in 2001, there is still a lot of fielded technology from the late seventies, and systems being designed today will remain in use for decades to come.

2.1 Early systems

ATMs were the 'killer application' that got cryptography into wide use outside of military and diplomatic circles. Following card forgery attacks on early machines in the late 1960s and early 1970s, IBM developed a system whereby the customer's PIN was computed from their account number by encrypting it using a key called the *PIN derivation key*. This system was introduced in 1977 with the launch of the 3614 ATM series [7] and is described further in [1, 2].

At the host end, some thought was given to the problem of protecting cryptographic keys against the bank's own systems staff. Merely embedding the cryptography in an application, and protecting it with access control mechanisms, was felt to be insufficient, as many programming and operations staff would be able to get at the key. So vendors started building cryptoprocessors that kept keys in tamper-resistant hardware and limited what could be done with them. The IBM 3848, for example, supported encrypted communications between a mainframe and a terminal without letting the mainframe programmers get at the key material [3, 9]. It was also adapted to control ATM networks, and evolved into the IBM 4758 product, described below.

The 3848 and similar devices contained tamper-resistant memory, implemented as RAM that was battery powered but had its power supply interrupted whenever the equipment was opened. This secure memory was not sufficiently sized to hold all the crypto keys that an application might use, and in any case had to be reloaded by hand after maintenance. So instead the secure RAM retained a small number of master keys, under which working keys were encrypted for storage outside the device. For example, the 3848 had three master keys; a working key encrypted under the first master key could be used to encrypt or decrypt data without restriction, while working keys encrypted under the other two were limited to 'local' and 'remote' use respectively. We'll give concrete examples of transactions involving restricted-use encrypted keys in the next section.

2.2 The Visa Security Module

The first generation devices such as the 3848 gave way in the mid 1980s to second generation products including the Visa Security Module (VSM), which was one of the most widely adopted. This is a cryptoprocessor whose function is to protect PINs transmitted over bank ATM networks. It was designed in the early eighties, and has many clones – including a software-compatible product analysed during the course of this research.

VISA's goal in promoting this technology was to persuade member banks to hook up their ATMs to VISA's network, so that a customer of one member bank could get cash from an ATM owned by another member. To do this, VISA had to make it harder for any of its member banks to lose money as a result of the dishonesty or negligence of someone at another member. This meant, inter alia, that no single employee of any bank in the network should learn the clear value of any customer's PIN. If PINs in transit to the verifying bank were simply managed in the software running on the banks' mainframes, then system programmers could learn the PIN of any customer who passed a transaction through their bank. They might then forge a card; or a customer could successfully defraud the bank by falsely disputing a transaction, claiming that some bank insider must be responsible. So the cryptographic systems used to compute and verify PINs had to support a policy of *shared control* [1, 2].

The implementation of this policy was that each node in the system had to contain an approved cryptographic device to protect the customer PINs passing through, between which key material had to be set up under dual control. The PINs themselves were generated on printers attached physically to the security modules, and mailed out separately from the cards. Key shares for both ATM and interbank key setup were printed with these printers on the same sort of tamper-evident envelope stock as used for PIN issue. In the case of a link between a bank and an ATM, the bank's central security module would generate two or more key shares, to be carried by separate people to each ATM when it was initially brought online. These were combined together by bitwise exclusive-or to create a *terminal master key* (conventionally known as KMT), and further encryption keys would then be sent to the device encrypted under this master key. In a similar way, interbank keys were set up by hand-carrying three shares from one bank's security module to the other's.

2.3 Known-key attack

The upshot was that most bank security modules had a transaction to generate a key share and print out its clear value on an attached security printer. It also returned this value to the calling program, encrypted under a master key (which we'll call KM) which was kept in the tamper-resistant hardware:

Host $\longrightarrow$	VSM :	"Generate Key Share"
VSM $\longrightarrow$	printer:	KMT_i
VSM $\longrightarrow$	Host:	$\{KMT_i\}_{KM}$

The VSM had another transaction which combined two of the shares to produce a terminal key:

Host $\longrightarrow$	VSM:	"Combine Key Share", $\{KMT_1\}_{KM}, \{KMT_2\}_{KM}$
VSM $\longrightarrow$	Host:	$\{KMT_1 \oplus KMT_2\}_{KM}$

To generate a terminal master key, a programmer would use the first of these transactions twice followed by the second, giving KMT = KMT1 $\oplus$ KMT2. This version would be used by the host program to talk to the ATM, while the ATM created KMT directly when the two shares were entered manually.

The protocol failure is that the programmer can take any old encrypted key and supply it twice in the second transaction, resulting in a known terminal key (the key of all zeroes, as the key is exclusive-or'ed with itself):

Host $\longrightarrow$	VSM:	"Combine Key Share", $\{KMT_1\}_{KM}$, $\{KMT_1\}_{KM}$
VSM $\longrightarrow$	Host:	$\{KMT_1 \oplus KMT_1\}_{KM}$
	=	$\{0\}_{KM}$

There are now several ways in which an exploit can be implemented. One of the simplest uses a transaction that enables a programmer to encrypt the PIN key under a terminal master key, so that an ATM can verify customer PINs while the network is down. So now the programmer can obtain the PIN key encrypted under the all-zero key, decrypt it using his own computer, and is then able to compute any customer's PIN.

2.4 A 'two-time type' attack

While the above attack was found by inspection, the following one was found by formal methods – by writing a program that mapped the possible key and data transformations between different key types, computing the transitive closure under these, and scanning the composite operations for undesirable properties.

Like the 3848, the VSM enforces a type system on working keys. An interbank master key can only do certain transactions, different from those permitted for a terminal master key. As in the 3848, this is enforced by having separate master keys to encrypt separate key types.

The VSM has nine key types rather than the 3848's three, but these are still not really enough to express the syntax of the underlying application. For example,

terminal master keys and PIN derivation keys are treated as the same type.

It turns out that reusing a key type can be as dangerous as reusing a key in a one-time cryptosystem. Just as the Soviet re-use of key material during World War 2 led to what Bob Morris beautifully describes as the 'two-time pad', so the re-use of the terminal master key type for PIN generation keys makes it into a 'two-time type' that opens up another neat attack. One use of the terminal master key is to protect the transmission of a *terminal communications key* (KC) to an ATM from the host VSM. This type of key is used to compute MACs on messages, and as the message cleartext is assumed to be freely available anyway, there are no restrictions on the use of a KC for encryption or decryption. So, for convenience, there is also a transaction that allows a clear key to be entered into the system as a KC (that is, encrypted under the relevant master key, which for simplicity's sake we'll call KMC).

However, there is also a transaction that allows a KC to be decrypted from KMC and re-encrypted under any terminal master key. This is designed to allow existing KCs to be sent out to ATMs in the field following re-keying. However, taken together with the fact that a PIN derivation key can be passed off as a terminal master key, it sets up an attack. Recall that a customer PIN is, in effect, their *primary account number* (PAN) encrypted under the PIN derivation key (say, KP): $PIN = \{PAN\}_{KP}$. So an attacker will enter the PAN into the system as a KC, by encrypting it under KMC:

Host $\longrightarrow$ VSM :	"Encrypt Comms Key" , PAN
VSM $\longrightarrow$ Host:	$\{PAN\}_{KMC}$

The second step is to get the VSM to take the encrypted PAN (which is now considered to be a KC) and re-encrypt it under a terminal master key. However, instead of supplying $\{KMT\}_{KM}$, we supply the PIN key $\{KP\}_{KM}$:

Host $\longrightarrow$ VSM :	"Translate Comms Key to KMT", $\{PAN\}_{KMC}$, $\{KP\}_{KM}$
VSM $\longrightarrow$ Host:	$\{PAN\}_{KP}$

The answer, $\{PAN\}_{KP}$, is just the PIN.

When keys are used for complex purposes, their security assumptions can also become complex and escape the untutored intuition. For example, there is a tendency to assume that encrypted data is no longer sensitive. But in this case, where the key is for PIN derivation, the result *is* a sensitive value – the customer's PIN.

A general problem with many common key-typing systems is that once a single key of a given type is compromised, all material at the next level down the hierarchy (i.e. encrypted with a key of this type) can be compromised too. For example, once any KMT or any KP is found, all keys output by a transaction that encrypts under this key type could be compromised. It is difficult to avoid this sort of vulnerability without a radical redesign. In the specific case of the VSM it is worse: because there is a transaction which encrypts one terminal master key under another, compromising a single KMT will also compromise all its neighbours at the same level.

The Prism security module [12], which is widely used in utility metering applications, is most successful in limiting damage from this threat. Each child key is bound together with the register number of its parent, so compromise of a parent only compromises the parent's direct children. There are flexibility and scalability issues with this approach (the module has a limited number of internal registers), but it is a step in the right direction.

Compromises may also cross type boundaries. For example, the VSM allows the export of PIN derivation keys over interbank links so that VISA can do stand-in PIN verification for a bank whose network is down. The result is that the compromise of an interbank key can allow a programmer to extract PIN generation keys (an incident of this nature is mentioned in [2]).

Type system design touches on a number of issues familiar from elsewhere in security engineering. Information-flow-based security policy models are an example. The policy statement 'the value of KP must never become known' is broadly equivalent to the statement that KP is at *High* in a multilevel secure system. Thus, if KP can be encrypted under KMT, KMT is also *High*. One is indeed reminded of the problems encountered by the designers of multilevel secure systems, in that classification schemes tend to classify either so little that the system is insecure, or so much that it is not usable. However, the analogy is not perfect, as an opponent who can get a known value of some KMT can break the system. 'Write-up' can be as dangerous as 'write-down', and the extent to which information flow policy ideas can be applied to key management systems appears to be an interesting open research problem.

2.5 Meet-in-the-Middle Attacks

The *meet-in-the-middle* attack exploits the fact that to abuse all the keys of a certain type, it's usually only necessary to get one of them. This leads to a natural time-memory tradeoff in keysearch.

Legacy cryptoprocessors typically use DES for all but a small number of high-level transactions, and many modern ones offer it as a backwards-compatibility mode. So a key can be found with an effort of about 2^{55}. Systems are therefore vulnerable to anyone who can organise a few thousand people to donate spare cycles to a keysearch effort. But it is often much worse than that! Many cryptoprocessors will happily generate a lot of keys of the target type. 2^{16} key generation transactions take somewhere between a few minutes and a few hours on the devices examined, and this can reduce the work involved in finding one of these keys to 2^{39}, which takes only a few days on a home PC.

The attack itself is straightforward. An identical test pattern is encrypted under each key, and the results recorded. The same test pattern is encrypted under each trial key and the result 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. 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. Using a hash table, the comparison stage can be made almost free.

In effect, the keysearch machine and the cryptoprocessor attack the key space

from opposite sides, and the effort expended by each meets somewhere in the middle.

This attack can compromise eight out of the nine types used by the VSM, as there are no limits or special authorisation requirements on key generation. The Prism security module permits an interesting variation, which even allows a top-level master key to be cracked with circa 2^{39} effort. The module's master key is manually loaded from multiple shares, and a test vector returned after the loading of each share, to ensure that it has been received correctly. The flaw is that any user can continue to XOR in chosen shares, receiving the same test vector encrypted under each in the response. With a few hours access, 2^{16} different variants of the master key can be created, along with the set of test vectors required for a meet-in-the-middle attack. We implemented this as an experiment and succeeded in extracting the master key from a device.

3 Breaking Current Cryptoprocessors

Third-generation cryptoprocessors such as the 4758 aim to achieve much more with their APIs than their predecessors. The 4758 is supplied with a default financial architecture – the *Common Cryptographic Architecture* (CCA) – which has 150 or so transactions, supporting a great range of banking applications; it is designed to be backwards compatible with the 3848 and to provide much of the functionality of the VSM as well [8, 10, 11]. Prism supplies crypto modules which all support a default transaction set (of 25 or so commands) and can then provide application specific extensions if required by their clients [12]. Of course, the more complex and customisable the transaction set is, the more the opportunity for designers to make mistakes.

Roger Needham called this process "the inevitable evolution of the Swiss army knife". There is a tendency for any computer architecture to become so versatile that it becomes difficult or impossible to follow the principle of least privilege, or even to understand which architectural features are security-relevant. Cryptoprocessors are unsettlingly like word-processing macro languages in this respect! The existence of backwards compatibility modes also complicates matters; they not only perpetuate old problems, but cause new problems too.

3.1 Type casting attacks

We mentioned that interbank keys are typically carried from one security module to another in the form of three shares, which in legacy equipment are simply xor'ed together to give a single DES key. In the case of the Prism device, the master key could be broken by running the further shares through a range of values, but the VSM was not vulnerable in this way, as a check value on the combined key was generated along with the key shares and had to be entered along with them to activate the new combined key (we'll discuss this in more detail later). One might think that the problems of managing key shares were by now understood.

The IBM 4758/CCA supports a key transfer procedure of this type, but there are a few strings attached. The transaction `Key_Part_Import` is used, in inde-

pendently authorisable modes – `Load_First_Key_Part` and `Key_Part_Combine`. By assigning permission for each of these modes to different users, a dual control policy can be implemented. When $n > 2$, things are not quite so simple. The first user is given `Load_First_Key_Part` and the remaining users `Key_Part_Combine`. One might think that this gives n-fold shared control, as all n shareholders can collude to discover the value of the communications key; but any single `Key_Part_Combine` holder can collude with the `Load_First_Key_Part` holder, in order to enter a chosen key into the cryptoprocessor. So it really gives only dual control.

But that is not all. CCA introduced a new key type mechanism known as *key control vectors*, with the laudable aim of supporting more key types, and more flexible key types, than previous products [10, 11]. For present purposes, a control vector is simply a string containing key type information that is exclusive-or'ed with the master key. The working key KW of type CV is stored under master key KM as the token $\{KW\}_{KM \oplus CV}$; when it is presented to the cryptographic processor, the claimed control vector is xor'ed with the current master key, the token is decrypted, and the parity of the result is checked to ensure that it is a valid key. Some control vectors are pre-specified (such as 'PIN generation key'), while new ones can be specified by the application designer.

The goals of this design included providing a reasonable amount of backwards compatibility with processors such as the 3848 and the VSM. One of the tricks that can be used to import encrypted keys from other systems is known as *pre-XOR type-casting* which allows the types of transferred keys to be modified during import: it involves simply XOR'ing the difference between the two control vectors to a key-importing key used to import the chosen key. In normal operation, the difference is introduced with an extra `Key_Part_Combine` operation once the final key is present. The vulnerability we noticed is that any individual key share holder can modify his key share at will, so although the absolute value of the key would remain unknown, the key share holder would be able to set up the keys required for a type-casting attack.

In response to an early draft of this work, IBM suggested that testing keys for integrity on import is the route to avoid the latter attack. But it's not at all obvious how to do this.

One approach would be for one bank to generate a key and test vector, split the key into three shares, and send each by courier to bank B, where they are reassembled and the test vector checked. If any shareholder had modified his key share (accidentally or deliberately), the test vector would not match, and the key exchange process could be aborted.

The difficulty comes in binding the testing operation to the completion of the import process. If you let the final shareholder test the key, he might approve a modified key into the system. So the verification must be done by somebody else. For example, one might require anyone who uses a key to test it first. But then a type-casting attack can be performed when any one user colludes with any one key share holder. When the size of either these sets increases, the risk of collusion attacks is increased, not decreased! So the security of the system decreases as the key is split into more parts, and as we add more users.

A deeper objection to IBM's proposed solution is that even if all keys are checked

before use, this still doesn't stop the final key share holder from generating two complete keys, one true key and one key with the intended difference. The true key would then be passed on for testing and use, and the bogus one used for an attack.

The core of the problem is that having a separate and final testing stage can only work if testing is necessary before use. For example, one may build the key internally within the cryptoprocessor, and require a correct test vector before releasing it, so that partial, unverified keys are not returned to the host. (This is how the VSM works – although the test vectors are only six decimal digits.) Alternatively, one could make a type distinction between verified and unverified keys. Indeed, key verification appears to have been introduced as a measure against accidental errors in key share entry, rather than malicious modification of keys. It requires more careful attention in future designs.

One conclusion to draw is that whenever we use a combining function with arithmetic properties, all dependent protocols should be checked for potentially unpleasant side-effects of these properties. In other words, IBM's choice of combining function raised the complexity of transaction set verification.

3.2 How to import key shares properly

So how can we import key shares safely, in such as way that the only attack requires collusion between all n shareholders? The first solution that comes to mind is to use a cryptographic hash function instead of XOR to combine the shares. With this method, a shareholder who modifies his keypart can only introduce a random difference between the loaded key and the intended key.

$$K = H(S1, S2, S3)$$

However, this method is not suitable where an already existing key must be shared – the Si cannot be calculated from an already chosen K. In that case, one possible approach might be to decrypt K under two successive shares and use the result as the third share – a verification value. No single key share holder can introduce a known difference between the loaded key and the intended key.

$$K = \{\{\{S1\}_{S2}\}_{S3}$$

One can come up with many variant schemes, some with distinct testing stages, detailed contextual information in each share (e.g. share number, destination module, timestamp ...), but there is an important requirement to put upon the key share entry method, before we are home and dry. No matter what the key share combination method, the transaction for each share entry must be distinct and independently authorisable. If any two users were to share the same transaction for key share entry, then the work of the former would reproducible by the latter, so $n-1$ key share holders could collude to mount an attack in an n share system. The 4758/CCA method uses the same transaction for all but the holder of the first key share, so the maximum n is 2. The transaction set must allow the cryptoprocessor to keep track of how many distinct users have contributed to the key.

The procedure used with some VSM clones remains a model of good practice. When an interbank key is generated, three officials stand round the machine; a special 'security manager' key is inserted to put the equipment into a highly authorised state; three key mailers are produced, each with a key component and the (same) check value on the (combined) key; these mailers are taken to the correspondent bank and entered; if the three keys combine into one with the check value, the key becomes live.

However, optimisations of this simple procedure seem to be dangerous. If key shares are not entered simultaneously in an atomic transaction, then binding the component transactions becomes a problem. The situation is further confused where the confidentiality and integrity of the key are treated separately. For example, any system with a single user authorised as the tester allows a key to be damaged by collusion between the tester and any key share holder.

3.3 Backwards compatibility and the key binding attack

Since the 3848, concerns about the vulnerability of DES to keysearch have led cryptoprocessor designers to support triple DES (3DES), often with two keys and sometimes with three. 3DES has the property that, if the multiple input keys used are the same, it performs exactly as single DES, thus providing backwards compatibility. Export licensing pressures originally limited 3DES to top-level master key operations and to irreversible operations such as computing the check digits for use on bank card magnetic strips, but it is now used for more and more functions.

The 4758 CCA has a subtle implementation problem with 3DES, in that it does not properly bind together the halves of its 3DES keys. Each half has an associated type, which distinguishes between left halves, right halves, and single DES keys. However, the type system does not specifically associate the left and right halves of a particular key. The result is that one can use keysearch to discover the halves of a 3DES key one at a time. For example, if we know KAL and KAR, and wish to discover KXL and KXR, then we can encrypt test values under (KAL, KXR) to recover KXR and then under (KXL, KXR) to discover KXL. It turns out that our meet-in-the-middle technique works well with this attack. Provided we can find out the value of a single key half, and encrypt a reasonable number of known test vectors, we can break all the DES keys of interest in the device (including keys which do not have export permissions).

A 4758 backwards-compatibility feature allows us to get the known key half we need for this attack. This feature gives the option to generate replicate 3DES keys – keys with both halves having the same value. Again, the meet-in-the-middle attack cuts the effort from about 2^{55} to about 2^{40}. The attacker generates 2^{16} replicate keys sharing the same type as the target key, and then searches for the value of two of them. The halves of the two replicate keys can then be exchanged to make a 3DES key 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.

In the particular case of the 4758/CCA, generating a large number of keys is essentially free. The IBM products have for years used key formats without any plaintext padding, so that keys could be generated simply by choosing some value and submitting it as an encrypted key. The decrypted result will thus be an unknown pseudorandom value. The cryptoprocessor would then manually adjust the parity. So our 2^{16} test values can be computed as fast as we can supply random (or counter) values to the device and store the responses. We refer to this feature as *key conjuring* [6].

A 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. In fact, a brute force effort in software could search for all non-exportable keys in the same pass.

4 Possible Future Research

The latest cryptoprocessors have forsaken manual secret-key exchange and use public key cryptography to exchange symmetric transport keys. It is not clear how much things change; shared control is still required to achieve the same level of assurance. Getting the procedural controls right for public key exchange may turn out to be at least as difficult, because of the counterintuitive twists introduced by the asymmetry of the underlying mechanism. The design of public key protocols is notoriously hard, and their interaction with tamper-resistant embedded devices is by no means fully explored.

A related issue is the design of formats for keying material. One might expect that a key being transported should be padded with a checksum, and with freshness information such as a nonce or date. However, many designs have failed because keys are encrypted first and their contextual information tacked on afterwards, often using mechanisms that break. For example, failures of protocols that use public key encryption before signature are discussed in [5]. There may be a psychological factor at work here, in that designers feel a clear key is 'radioactive' and must be shielded as soon as possible by encryption. Be that as it may, the design of key formats is another opportunity for research.

Another related issue is trusted path. One of the reasons that top-level key management seems more robust in the VSM than in the 4758 appears to be that the former has a terminal physically attached to the device, at which management operations are conducted, as well as a printer at which key components are output. The VSM has a supervisor password to control this access; one clone goes further, with separate physical keys for routine security operations (such as printing customer PINs and ATM keys) and top-level ones (such as generating interbank keys). This means that the holder of the top-level key can ensure that all three key share holders are physically present at the device while the whole operation is done atomically.

The 4758, on the other hand, works as a PC peripheral, so it seems to have been natural for the designers to make management operations more flexible. However, the trust boundary for key management may also include operating system access control, virus protection, network security and so on – so it's less clear what value a tamper-resistant cryptoprocessor adds! The interaction of trusted path with shared control and environmental issues promises to be even harder in a world of ubiquitous computing.

Another issue is understanding protection dependencies. A common cause of real-life security failures is that an application evolves in some way that causes assumptions to no longer hold. For example, we might not be too concerned about card forgery attacks an electronic purse that only makes online transactions to merchants, as the threat model is almost identical to that of magnetic-strip card forgery; but if transfers are suddenly allowed between customer purses, then the mechanisms of hot cards and floor limits break down, and large-scale fraud is suddenly possible. Furthermore, the compromise of the master key from a single card can now break the entire system, rather than do fraud on a single account. This is a (deliberately) blatant example; there are many more subtle ones [1]. Ideally, we want better tools for tracking dependencies between protection goals, assumptions and mechanisms as systems evolve.

Finally, there are broader computer science issues. Given a number of embedded processors that enforce different security properties – say, an electronic purse, a postal meter and the SIM card of a mobile phone – how do we go about building a secure system? In other words, given N processors each supporting a different security policy, how do we compose them into something that supports yet another security policy? This *composition problem* is an old chestnut. So is the problem of the interaction of security and reliability: how can we built a robust, secure system out of less dependable components? There is also the related question of whether there any deep philosophical difference between access control in a host CPU, in a cryptoprocessor, and in an application. Perhaps this new space of application problems will give valuable new insights.

5 Conclusions

The design of security APIs is a new field of research, of significant industrial and scientific importance. The poor design of present interfaces prevents many tamper-resistant processors from achieving their potential, and leaves disappointing dependency on procedural controls – the design of which involves subtleties that are likely to be beyond the grasp of most implementers.

In this article, we have touched on a number of specific design failures. Some of these are new, such as the key binding problems with triple-DES, type casting attacks, subtle interactions with backward compatibility modes, and new types of chosen-key attack and meet-in-the-middle attack. Others are variants of problems already encountered elsewhere, such as trusted path issues and the use of combining functions such as XOR that have exploitable mathematical properties. Many involve the interface between different protection technologies, such as between type systems and cryptology, and between technical and procedural mechanisms for shared control. Some were found by inspection, and

others by the application of crude formal methods to the published transaction sets.

We are only starting to come to grips with the deeper, conceptual issues. It is unclear that a 'generalised' API will work: as we have seen, the natural accretion of functionality is one of the great enemies of security. Yet getting the API right is relevant for much more than just cryptoprocessors. The API is 'where the rubber hits the road', as it is where the cryptography, the protocols, the operating system access controls, and the operating procedures all come together – or fail to. It truly is a microcosm of the security engineering problem; a large number of tools can be brought to bear, and hopefully we will learn much of value about our existing techniques by applying them in this new problem space.

6 Acknowledgements

The authors wish to thank (alphabetically) Johann Bezuidenhoudt, Richard Clayton, George Danezis, Steve Early, Peter Landrock, Larry Paulson, and Don Taylor for input and feedback. The first author was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) and Marconi plc, while the second author was a joint investigator on the EU-sponsored G3Card project.

References

[1] RJ Anderson, 'Security Engineering - a Guide to Building Dependable Distributed Systems', Wiley (2001) ISBN 0-471-38922-6

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

[3] R Anderson, 'The Correctness of Crypto Transaction Sets', *Security Protocols – 8th International Workshop*, April 2000 (proceedings to appear in Springer LNCS series)

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

[5] RJ Anderson, RM Needham, "Robustness principles for public key protocols", in *Advances in Cryptology – Crypto 95* Springer LNCS vol 963 pp 236–247; `http://www.cl.cam.ac.uk/ftp/users/rja14/robustness.ps.gz`

[6] M Bond, 'Attacks on Cryptoprocessor Transaction Sets', in *Workshop on Cryptographic Hardware and Embedded Systems* (CHES 2001) (proceedings to appear)

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

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

[9] CH Meyer, SM Matyas, 'Cryptography: A New Dimension in Computer Data Security', Wiley, 1982

[10] SM Matyas, 'Key Handling with Control Vectors', IBM Systems Journal v 30 no 2, 1991, pp 151–174

[11] SM Matyas, AV Le, DG Abraham, 'A Key Management Scheme Based on Control Vectors', IBM Systems Journal v 30 no 2, 1991, pp 175–191

[12] `http://www.prism.co.za`