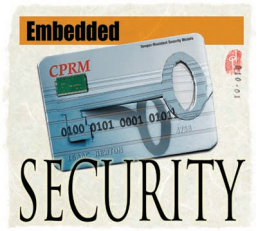
API-Level Attacks on Embedded Systems



We have recently discovered a whole new family of attacks on APIs used by security processors. These attacks are economically important, as security processors are used to support a wide range of services—from ATMs to prepayment utility metering—but designing APIs that resist them is difficult.

Mike Bond
Ross Anderson
Cambridge University

A whole new family of attacks has recently been discovered on the application programming interfaces (APIs) used by security processors. These extend and generalize 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 its designer envisioned. Our 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 them is difficult, as a typical security processor needs several dozen commands in order to service a number of external and internal protocols. Our attacks are also scientifically interesting; preventing them may become an important new application area for formal methods and design verification tools generally.

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

- smart cards used to personalize mobile phones and to manage subscribers to satellite-TV services;
- microcontrollers used as value counters in postal meters and 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 that must prevent fraud by distributors, network operators, and other participants in the value chain.

A good example is the prepayment electricity meters used to sell electric power to students in residence halls and to poorer customers in general.[1] They are typical of the many systems that once used coin-operated vending, but have now switched to tokens such as magnetic cards or smart cards. The principle of operation is simple. The meter will supply a certain quantity of energy on receipt of an encrypted instruction. The instructions are created in a token vending machine, which knows the secret key of each local meter. One design goal is to limit the loss if a vending machine is stolen or misused; this lets the supplier 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 around a tamper-resistant cryptographic processor, which contains the meter keys and a value counter. The value counter enforces a credit limit whereby the vending machine stops working upon reaching that limit and must be reloaded. Reloading requires an encrypted message from a controller one step higher up the chain of control, and is typically done by the distributor after payment by the machine operator. If anyone attempts to tamper with the value counter, the cryptographic keys are erased and the vending machine will no longer work. Without these

0018-9162/01/$10.00 © 2001 IEEE

No matter what the application of the security processor, its API sits on the boundary between trusted and untrusted environments.

controls, the theft of a vending machine might compel the distributor to re-key all the meters within its vend area.

A similar arrangement can be found in networks of 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. It is the point where cryptography, protocols, access controls, and operating procedures must all come together to enforce its security policy. Thus, understanding the threats to these APIs is essential.

ATTACKS ON CRYPTOGRAPHIC APIS

Our research into attacks on cryptographic APIs started with an examination of ways in which banking security modules can be manipulated[2] and proceeded to a study of attacks on more general-purpose tamper-resistant processors.[3] Recently, we have also found attacks on security processors used in utility prepayment applications.

Explaining these attacks in historical order is convenient, but the attacks on both old and new systems are interesting for a number of general reasons:

- First, there has been a good deal of work on verifying crypto protocols, which are typically sets of three to five 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 smart card, 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 focused on avoiding errors in the API implementation, and verifying its correspondence to its specification. Thus, API design could 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 a processor supports. In the same way, design flaws in crypto APIs occur at the even more complex interface between crypto, security protocols, operating system access controls, and specialist services such as value counters.
- Finally, a tamper-resistant device can simply be considered as a high-quality implementation of an object that can only be invoked using its official methods and whose internal variables remain inaccessible. Given the popularity of the object-oriented programming model, there may be more general lessons to be learned for robust programming.

Thus, learning how to design security APIs properly is important, especially if we are to realize the potential of systems that distribute trust across a heterogeneous set of processes. It may thus be fundamental for large-scale embedded systems. Rectifying mistakes afterwards can be horrendously expensive. Once a system is as well entrenched as ATMs are—with over 600,000 devices from more than 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 1970s, and systems being designed today will remain in use for decades to come.

EARLY SYSTEMS

ATMs were the "killer application" that got cryptography into wide use outside military and diplomatic circles. Following card forgery attacks on early machines in the late 1960s and early 1970s, IBM developed a system in which the customer's PIN was computed by encrypting their account number using a key called the *PIN derivation key*. This system was introduced in 1977 with the launch of the 3614 ATM series.[4-6]

IBM 3848 cryptoprocessor

At the host end, vendors gave some thought 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.[2,7] It was also adapted to control ATM networks and evolved into the IBM 4758 product.

The 3848 and similar devices contained tamper-resistant memory, implemented as battery-powered RAM that had its power supply interrupted whenever the

equipment was opened. This secure memory was not large enough to hold all the crypto keys that an application might use, and in any case had to be reloaded by hand after maintenance. Therefore, 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, whereas working keys encrypted under the other two were limited to local and remote use respectively.

Visa Security Module

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 widely adopted. The VSM is a cryptoprocessor whose function is to protect PINs transmitted over bank ATM networks. It was designed in the early 1980s and has many clones—including a software-compatible product analyzed during our 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 bank. One goal was that no single employee of any bank in the network should learn the clear value of any customer's PIN, but if PINs in transit to the verifying bank were simply managed in the software running on the banks' mainframes, system programmers could learn the PIN of any customer who passed a transaction through their bank. A programmer might then forge a card; or a customer could successfully defraud the bank by falsely disputing a transaction and 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.[5,6]

ATM network designers implemented this policy by having each node in the system contain an approved cryptographic device to protect the customer PINs in transit and setting up key material under dual control between nodes. The PINs were generated on printers attached physically to the security modules and mailed out separately from the ATM cards. Key shares for both ATM and interbank key setup were printed on the same sort of tamper-evident envelope stock 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 (XOR) to create a *terminal master key* conventionally known as KMT; subsequent encryption keys would then be sent to the device encrypted under this master key. Similarly, participating banks set up interbank keys by hand-carrying three shares from one bank's security module to the other's.

Thus, 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* (KM) that was kept in the tamper-resistant hardware:

$$\begin{aligned} \text{Host} &\rightarrow \text{VSM} : \text{“Generate Key Share”} \\ \text{VSM} &\rightarrow \text{Printer} : KMT_i \\ \text{VSM} &\rightarrow \text{Host} : \{KMT_i\}_{KM} \end{aligned}$$

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

$$\begin{aligned} \text{Host} \rightarrow \text{VSM} &: \text{“Combine Keys”}, \{KMT_1\}_{KM}, \\ &\quad \{KMT_2\}_{KM} \\ \text{VSM} \rightarrow \text{Host} &: \{KMT_1 \oplus KMT_2\}_{KM} \end{aligned}$$

To generate a terminal master key, a programmer would use the first of these transactions twice followed by the second, giving $KMT = KMT_1 \oplus KMT_2$. The host program would use this version to talk to the ATM, while the ATM created a KMT directly when the two shares were entered manually.

Known-key attack

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 XORed with itself):

$$\begin{aligned} \text{Host} \rightarrow \text{VSM} &: \text{“Combine Keys”}, \{KMT_1\}_{KM}, \\ &\quad \{KMT_1\}_{KM} \\ \text{VSM} \rightarrow \text{Host} &: \{KMT_1 \oplus KMT_1\}_{KM} \\ &\quad = \{0\}_{KM} \end{aligned}$$

There are now several ways in which an exploit can be implemented. One of the simplest uses a transaction that lets a programmer encrypt the PIN key under a terminal master key so that an ATM can verify customer PINs while the network is down. In this attack, the programmer obtains the PIN key encrypted under the all-zero key and decrypts it using his own computer. This enables him to obtain every customer's PIN.

Two-time type attack

While the above attack was found by inspection, we found the following attack using formal methods—by writing a program that mapped the possible key and data transformations between different key

Visa had to make it harder for any of its member banks to lose money as a result of dishonesty at another bank.

When keys are used for complex purposes, their security assumptions can also become complex and escape the untutored intuition.

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, which are different from those permitted for a terminal master key. As in the 3848, the VSM enforces this policy 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 enough to express the syntax of the underlying application. For example, terminal master keys and PIN derivation keys are treated as the same type.

Reusing a key type can be as dangerous as reusing a key in a one-time cryptosystem. Just as the Soviet reuse of key material during World War II led to what Bob Morris beautifully describes as the "two-time pad," so reusing 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 (message authentication codes) on messages. 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 reencrypted under any terminal master key. This allows existing KCs to be sent out to ATMs in the field following rekeying. 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 PIN is, in effect, a customer's primary account number (PAN) encrypted under the PIN derivation key (KP):

$$\text{Host} \rightarrow \text{VSM} : \text{“Encrypt Comms Key”}, PAN$$
$$\text{VSM} \rightarrow \text{Host} : \{PAN\}_{KMC}$$

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

$$\text{Host} \rightarrow \text{VSM} : \text{“KC to KMT”}, \{PAN\}_{KMC}, \{KP\}_{KM}$$
$$\text{VSM} \rightarrow \text{Host} : \{PAN\}_{KP}$$

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—that is, encrypted with a key of this type—can be compromised too. For example, once any KMT or 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 that encrypts one terminal master key under another, compromising a single KMT will also compromise all its neighbors at the same level.

The Prism security module,[8] 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 can 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.[6]

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 "KP is at High in a multilevel secure system." Thus, if KP can be encrypted under KMT, KMT is also high. This is a reminder 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 a 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 is an interesting open research problem.

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 naturally to a time-memory trade-off in key search.

Legacy cryptoprocessors typically use DES (the Data Encryption Standard) for all but a small number of high-level transactions, and many modern ones offer it as a backwards-compatibility mode. A key can be found with an effort of about 2^{55}. Systems are therefore vulnerable to someone who can organize 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 are 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. Using a hash table, the comparison stage is 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 authorization requirements on key generation. The Prism security module permits an interesting variation, which even allows a top-level master key to be cracked with about 2^{39} effort. The module's master key is manually loaded from multiple shares, and a test vector is 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 variant in successive responses. 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.

BREAKING CURRENT CRYPTOPROCESSORS

Third-generation cryptoprocessors such as the IBM 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 VSM functionality as well.[9-11] By comparison, Prism supplies crypto modules that all support a default transaction set of 25 or so commands. Both vendors are prepared to customize the transaction set if required by their clients.[8] Of course, the more complex and customizable the transaction set is, the more the opportunity for designers to make mistakes.

Roger Needham called the process by which APIs get progressively more complex "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.

There is a tendency for any computer architecture to become so versatile that it becomes impossible to follow the principle of least privilege.

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 XORed together to give a single DES key. With the Prism module, combining in further chosen shares to create a range of values allowed us to break the master key, but the VSM was not vulnerable in this way. It used a check value on the combined key along with each key share, which had to be entered along with them to activate the new combined key.

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 independently authorizable 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 to enter a chosen key into the cryptoprocessor, so it really gives only dual control.

But that is not all. The CCA introduced *control vectors*, a new key type mechanism with the laudable aim of supporting more key types and more flexible key types than previous products.[10,11] A control vector is simply a string containing key type information that is XORed 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 KW is presented to the crypto-

Whenever we use a combining function with arithmetic properties, all dependent protocols should be checked for potentially unpleasant side effects.

graphic processor, the claimed control vector is XORed 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 prespecified (such as "PIN generation key"), but application designers can specify new ones.

The goals of this design included providing a reasonable amount of backwards compatibility with processors such as the 3848 and the VSM. One trick 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 XORing 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 shareholder can modify his key share at will. Although the absolute value of the key would remain unknown, the key shareholder 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 is not at all obvious how to do this.

One approach would be for one bank to generate a key and check value, split the key into three shares, and send each by courier to bank B, where they are reassembled and the check value tested. If any shareholder had modified his key share (accidentally or deliberately), the check value 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. Thus, somebody else must do the verification. For example, you could 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 of these sets increases, the risk of collusion attacks is increased, not decreased. 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 does not stop the final key shareholder 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 problem is that having a separate and final testing stage can only work if testing is necessary before use. For example, we can build the key internally within the cryptoprocessor and require a correct check value before releasing it, so that partial, unverified keys are not returned to the host. (This is how the VSM works—although the check values are only six decimal digits.) Alternatively, we 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 a combining function raised the complexity of transaction set verification.

How to import key shares properly

So how can we import key shares safely so that the only attack requires collusion between all n shareholders? One solution 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 perform nested encryptions on the first share, using the successive shares as keys. No single key shareholder can introduce a known difference between the loaded key and the intended key:

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

We can come up with many variant schemes, some with distinct testing stages or detailed contextual information in each share (for example, share number, destination module, time stamp, and so on), 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 authorizable. If any two users share the same transaction for key share entry, the work of one would reproducible by the other, so n-1 key shareholders 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 authorized 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 correct check value, that key becomes live.

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

Backwards compatibility and key-binding attacks

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 is implemented as "encrypt, decrypt, encrypt" with single DES, so if the multiple input keys used are the same, 3DES 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 functions.

The 4758 CCA has a subtle implementation problem with 3DES. It uses a two-key mode of 3DES (one key for both encryptions, the other for the decryption), but it does not properly bind together these two halves of the whole 3DES key. Each half has an associated control vector, 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 we 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, we can encrypt test values under (KAL, KXR) to recover KXR and then under (KXL, KXR) to discover KXL. Our meet-in-the-middle technique works well with this attack. Provided we can find the value of a single key half and encrypt a reasonable number of known test values, we can break all the DES keys of interest in the device (including keys that 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 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 is thus an unknown pseudorandom value (the cryptoprocessor then manually adjusts the parity). So our 2^{16} test values can be automatically computed as fast as we can supply different input values to the device and store the responses. We refer to this feature as key conjuring.[3]

A nonexportable 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 this would be a valuable long-term key, justifying the expense. In fact, a brute-force effort in software could search for all nonexportable keys in the same pass.

Designing public-key protocols is notoriously hard, and their interaction with tamper-resistant embedded devices is by no means fully explored.

FUTURE RESEARCH

The latest cryptoprocessors have forsaken manual secret-key exchange and use public-key cryptography to exchange symmetric transport keys. However, it is not clear how much things have changed because shared control is still required to achieve the same level of assurance. Getting the procedural controls right for public-key exchange may be at least as difficult, because of the counterintuitive twists introduced by the asymmetry of the underlying mechanism. Designing 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. We might expect that a key being transported should be padded with a checksum and with freshness information such as a nonce or date.

Designing security APIs is a new field of research that has significant industrial and scientific importance.

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 were discussed in previous work.[12] 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 reason that top-level key management seems more robust in the VSM than in the 4758 is 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 shareholders 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 can 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 ways that cause assumptions to no longer hold. For example, we might not be too concerned about card forgery attacks on 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, 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 defraud a single account. This is a (deliberately) blatant example, but there are many more subtle ones.[5] 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—an electronic purse, a postal meter, and the SIM card of a mobile phone—how do we go about building a secure system using them? 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 build a robust, secure system out of less dependable components? A related question is whether there is any deep philosophical difference between access control in a host CPU, a cryptoprocessor, and an application. Perhaps this new space of application problems will give valuable new insights.

Designing security APIs is a new field of research that has 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.

Some of the design failures we have touched on are new, such as the key-binding problems with triple-DES, type-casting attacks, subtle interactions with backwards compatibility modes, and new types of chosen-key and meet-in-the-middle attacks. Others are variants of problems already encountered elsewhere, such as trusted-path issues and using 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 failures were found by inspection, others by applying 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 "generalized" 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 more than just cryptoprocessors. The API is "where the rubber hits the road," as it is where cryptography, protocols, operating system access controls, and operating procedures all come together—or fail to. It truly is a microcosm of the security engineering problem. Many 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. ✸

Acknowledgments

The authors thank (alphabetically) Johann Bezuidenhoudt, Richard Clayton, George Danezis,

Steve Early, Peter Landrock, Larry Paulson, and Don Taylor for input and feedback. Mike Bond was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) and Marconi plc, while Ross Anderson was a joint investigator on the EU-sponsored G3Card project.

References

1. R.J. Anderson and S.J. Bezuidenhoudt, "On the Reliability of Electronic Payment Systems," *IEEE Trans. Software Eng.*, May 1996, pp. 294-301.
2. R.J. Anderson, "The Correctness of Crypto Transaction Sets," *Proc. 8th Int'l Workshop Security Protocols*, Lecture Notes in Computer Science, vol. 2133, Springer-Verlag, Heidelberg, 2001, pp. 123-139.
3. M. Bond, "Attacks on Cryptoprocessor Transaction Sets," *Proc. Workshop Cryptographic Hardware and Embedded Systems* (CHES 2001), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, Heidelberg, 2001, pp. 220-234.
4. *IBM 3614 Consumer Transaction Facility Implementation Planning Guide*, IBM document ZZ20-3789-1, 2nd ed., IBM, Armonk, N.Y., 1977.
5. R.J. Anderson, *Security Engineering—A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, New York, 2001.
6. R.J. Anderson, "Why Cryptosystems Fail," *Comm. ACM*, Nov. 1994, pp. 32-40.
7. C.H. Meyer and S.M. Matyas, *Cryptography: A New Dimension in Computer Data Security*, John Wiley & Sons, New York, 1982.
8. http://www.prism.co.za
9. *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide for the IBM 4758-001*, Release 1.31, IBM, Armonk, N.Y., 1999; http://www.ibm.com/security/cryptocards/bscsvc02.pdf.
10. S.M. Matyas, "Key Handling with Control Vectors," *IBM Systems J.*, vol. 30, no. 2, 1991, pp. 151-174.
11. S.M. Matyas, A.V. Le, and D.G. Abraham, "A Key Management Scheme Based on Control Vectors," *IBM Systems J.*, vol. 30, no. 2, 1991, pp. 175-191.
12. R.J. Anderson and R.M. Needham, "Robustness Principles for Public Key Protocols," *Proc. 15th Ann. Int'l Cryptology Conf.* (Crypto 95), Lecture Notes in Computer Science, vol. 963, Springer-Verlag, Heidelberg, 1995, pp. 236-247.

Mike Bond *is a member of the security group at the University of Cambridge Computer Laboratory. His research interests include the design, analysis, and verification of security APIs and tamper-resistant hardware. More of his recent work has been presented at CHES 2001.*

Ross Anderson *leads the security group at the Computer Laboratory, Cambridge University. He has written on a number of security topics including cryptography, security protocols, tamper resistance, and applications. He is the author of the book* Security Engineering: A Guide to Building Dependable Distributed Systems *(John Wiley & Sons, New York, 2001).*

Contact the authors at {Mike.Bond,Ross.Anderson}@cl.cam.ac.uk.