Integrity of Intention
(A Theory of Types for Security APIs)

Mike Bond and Jolyon Clulow

Computer Laboratory, University of Cambridge,
JJ Thompson Av., CB3 0FD, UK
{Mike.Bond, Jolyon.Clulow}@cl.cam.ac.uk

Abstract. The task of a security API is to allow users to process data and key material according to the designer's intentions, and to prevent any malicious sequence of commands from violating these intentions. Security APIs do this by attaching metadata to keys and data – *type information* – to record acceptable usage policy, which is checked by individual API commands in order to approve or deny a particular manipulation. But what actually is type information? This paper proposes a conceptual framework for understanding cryptographic type, and how it maintains the integrity of the designers intentions in an API. We describe four core conceptual components of type: FORM, USE, ROLE and DOMAIN. We compare our model to real-life security APIs, and argue that designing new systems within the bounds of the model improves safety, eliminating many common security issues.

1 Introduction

A *security API* is an API that uses cryptography to enforce policy on the user's manipulation of sensitive data and key material; they are conceptual extensions of both the crypto API and the security protocol. They are most often found in Hardware Security Modules (HSMs), devices which look after sensitive data in hostile environments where the operators of a device cannot necessarily be trusted. Security APIs have been studied academically since the 1990's, first by Longley and Rigby [11], and by the authors and Anderson from 2000 [4, 3, 7]. Most recently, formal analysis tools have been applied to check for security properties of APIs, and search for vulnerabilities. Steel [12], Ganapathy *et al* [8] and the authors have all taken different approaches to building trust in APIs through formal analysis. However none of this work has properly addressed a crucial practical issue for academic analysts and industrial designers alike: the need for a framework of concepts and terminology to understand and discuss different security APIs together.

In this paper, we lay down terminology and propose a generic model that tackles the central issue of *typing* – the separation of different sorts of keys and data from one another in Security APIs. What is type information for? When should you have it, and what should it look like? What goes wrong when a type system is poorly conceived?

In section 2 we first consider how to identify sensitive data and key material which needs to be protected by a Security API, and introduce the theme of "integrity of intention", as the goal of a type system. In section 3 we describe a conceptual framework for studying type information that we call the *four axis model*, and in section 4 we show that it yields a convenient categorisation of API failures. In section 5 we present case studies of security API type systems, viewed through the four-axis framework. We draw some conclusions in section 6.

2 Securing Inputs to Security APIs

The typical security API implements a diverse set of application specific functions, as well as providing key management for the basic crypto services you would find in a crypto API. When designing systems that work with application specific data and keys, the designer must first consider afresh what data and inputs are sensitive, and know how to protect them. In this discussion, we consider an arbitrary application specific function f implemented in the API of an HSM. The function accepts inputs x and y and returns the computed result $f(x,y)$.

$$\begin{aligned} User \rightarrow HSM &: \; x, \; y \\ HSM \rightarrow User &: f(x,y) \end{aligned}$$

At this stage, the API is simply providing a cryptographic service. However, suppose that the inputs to f are sensitive, for instance individual sealed bids in an auction for a house, whereas $f(x,y)$ (the winning bid and its value) is not. The designer could protect the secrecy of these bids by having the auctioneer work with encrypted inputs, all secured under some key K known only to the API. Determining that x and y should be protected is the first step to securing the API.

Unfortunately real world API inputs are incredibly diverse, and a single function may sometimes have ten different inputs! Read an HSM manual and you will see mention of a bewildering array of terminology: master keys, keys, control vectors, PINs, plaintext, variants, tokens, blobs, hashes, MACs, offsets, PVVs, and various control and formatting information. Some of these inputs will have straightforward sets of security requirements derived from their intrinsic value. For instance if an API stores and processes passwords for login to some web service– e.g. online banking or medical records – then the classic requirements on the input hold: *confidentiality* (the password stays secret), *integrity* (attacker can't change your password to something he knows) and *authenticity* (this is the right password for the right user). Other inputs that do not have intrinsic value may obviously be security critical too: for example, a storage key at the top of a hierarchy of keys would reveal all data stored under it if discovered, so it inherits policy from the data stored beneath it. Crucially, there is a large grey area of sorts of input where there is neither a direct security policy derived from their intrinsic value, nor any obvious argument that the inputs are inconsequential. The majority of inputs to an API tend to lie within this category – where the policy requirements upon them are only indirect, because they are small pieces of a larger system.

But for now let us assume the sensitive inputs have been identified, and can be presented in encrypted form, as shown below:

$$\begin{aligned} User \rightarrow HSM &: \{x\}_K,\ \{y\}_K \\ HSM \rightarrow User &: \quad f(x,y) \end{aligned}$$

The confidentiality requirement above must be treated seriously, and basic enciphering of an input with a block cipher may not be sufficient. If the protocol notation curly braces are taken to mean CBC encryption, we still have the danger that identical inputs can be matched by their ciphertext, and that random ciphertexts may decrypt to a meaningful input – both problems which have been the undoing of PIN processing APIs in the past [5, 2]. We thus need randomised, non-malleable encryption. The example below adds a random confounder R_c and a hash of all the data $H()$ within the encryption.

$$\begin{aligned} User \rightarrow HSM &: \{R_c, x, H(R_c, x)\}_K,\ \{R'_c, y, H(R'_c, y)\}_K \\ HSM \rightarrow User &: \qquad\qquad\qquad f(x,y) \end{aligned}$$

Even with randomised, non-malleable encryption, things might still go wrong. In our auction with sealed bids, only the winning bid is supposed to be revealed, yet if the auctioneer also submits x and y twice to fake auctions, he can calculate $f(x,x)$ and $f(y,y)$ so discover the value of any bid.

Of course, the API designer probably never intended that the same bid could be submitted as both inputs to f. Checking that the full range of inputs to f yields no compromising output is the second step of securing the input design, yet a deeper question is revealed – when the API was designed did the security policy it was designed against actually require secrecy of bids in every case? For instance, suppose there is only a single sealed bid for an auction, was that bid really supposed to remain secret? We introduce the concept of *intention* as the ideal version of the device security policy.

Definition. INTENTION represents the true and idealised aims of a designer when producing an API, which may be updated retrospectively. We define INTENTION in the same way as it is used in Anderson's definition of an API attack [1], as *"a malicious sequence of commands which result in an output the designer could not possibly have intended"*. A security policy document may be explicit or implicit, it may be wrong, it may have gaps. INTENTION is never wrong, it need not be explicit, or even consciously known to the designer during conception of the design: when faced with an observed potential vulnerability the designer can retrospectively decide his intention. However, we do make an important constraint: INTENTION must be *consistent*. Conflicting intentions must not exist, nor be later brought into existence.

We now have a standard against which we can check input combinations for badness, albeit one which may be a moving target. Note though that the much harder part of the checking task is to ensure that inputs to an API command cannot yield compromising output *in conjunction with the other functions* provided by the API. But considering our example problem, how do we address the problems of the same bid being submitted twice, and more generally that the inputs to a generic function f cannot be used in arbitrary ways.

$$\begin{array}{ll} User \rightarrow HSM : & \{R_c, x, t, H(R_c, x, t)\}_K,\ \{R'_c, y, t', H(R'_c, y, t')\}_K \\ HSM \rightarrow User : & \qquad\qquad f(x, y) \end{array}$$

This brings us to the key feature of a security API – *type*. Both inputs now have a field t which describes the sort of input that x or y are. Regardless of the specific values of x and y the API can now enforce that they are not improperly processed. In the case of the sealed bid auction, there could be "A" bidders and "B" bidders, encoded in the type information, where the policy is that no two "A" bids or "B" bids can ever be played off against each other.

This idea of adding *type* information sounds simple enough, but when turning the example into a real world system, the complexity explodes. Having "A" bidders and "B" bidders is all very well, but auctions will typically have variable numbers of participants. Now what about bids offered in different currencies, must the currency identifier be part of the cryptographic type? What exchange rate is used? The most up-to-date value? Yet now we have an extra input to the API command, the exchange rate between currencies. What are the consequences of varying that maliciously? Maybe it is not a secret, but it must not be malleable to the auctioneer, so it must have authenticity and freshness.

For now we leave aside this example, and we devote the next section to the crucial issue of how to conceptually break down the amorphous type information t into the conceptual components which are best suited to implementing typical security API policies.

3 The Four-Axis (4AX) Model

We see from section 2 that keys and data need to be classified into *types* in order to ensure that they are not abused, yet what differences are there, and which differences should be recognised, and which ignored? Initial differences between cryptographic keys are obvious: an RSA key is not a 3DES key, and the bits of a 3DES key should not be interpretable by an API command as the 'p' and 'q' in the RSA calculation – thus RSA and DES keys should have different types. Likewise it might also be obvious that a key used for bulk data encryption must not be usable for exporting other keys, so these keys should have different types too.

The four-axis (4AX) model splits the type information of a key into four conceptual components: FORM, USE, ROLE and DOMAIN. By analysing descriptions of types and writing them out in terms of these four concepts, a designer can spot potential shortcomings in his type system; he can make sure it has all the differentiation it needs, but no more detail and flexibility than is necessary. The FORM concept captures the obvious aspect of type: the binary bit-string making up a key or some data will have varying structure and must be interpreted differently. Any API, regardless of security relevance has to keep track of this, and in the world of programming languages, this might correspond to a native type, such as `long`, `int`, or `char[8]`. The USE concept is about *policy enforcement*: the aspect that makes a security API different from an API just providing cryptographic services. The concept of ROLE augments USE, capturing dynamic changes in key usage. A key that is fundamentally used to achieve one particular goal might need to be used in different ways by different parties, or used differently over time. The DOMAIN concept captures the uniqueness of specific instances of Security APIs (inputs for my API will not necessarily be accepted as inputs for yours), as well as the circumstances for interaction (for instance during top-level key exchange).

The examples below show two keys described first in English, then with an example representation of type from a typical API that processes them, followed by a 4AX representation of the same data.

Key One: A PIN derivation DES key in a bank ATM network, for local verification of PINs, on a Thales RG7000.

Type: Keypair 14&15 (TMK/PIN)

```
Form:   Single DES
Use :   PIN Derivation
Role:   Verify
Domain: KM (implicit)
```

Key Two: A 1024-bit private RSA signing key, for user Fred Jones in an applet code-signing CA, on a Chrysalis Luna CA3, accessed using PKCS#11

Type:

```
CKA_SUBJECT = Key Two
CKA_SIGN = TRUE
CKA_KEY_TYPE = CKK_RSA
CKA_MODULUS_BITS = 1024

Form  : RSA, Private, 1024Bit
Use   : Signature
Role  : Signature Creation
Domain: Belongs to Fred Jones (explicit)
```

3.1 Form

FORM is data explaining the mathematical interpretation of the binary data. It normally consists of an identifier for the cryptographic algorithm with which the data was intended to be used, and possibly specific other parameters for the algorithm (such as RSA modulus size).

In an ideal world, the data structure containing key material should have the form information integrated, and the form information should be considered part of the value of a key. However, since the first HSMs and APIs used homogeneous key forms (all single DES, for instance), there is a history of only implicit mention of the key form in the type information.

An arbitrary line can be drawn between key value data and key FORM data: a byte containing the length of a data field forming a component of an RSA key might be considered as much a part of the key value as part of the FORM data. However there is a way to draw the line meaningfully: when such a data field clearly has a relationship to the security of the key, it should be considered FORM information. A 512-bit RSA is weaker than a 1024-bit key, so the modulus length is FORM data. However, a flag representing whether or not the subsequent data field is big endian or little endian can only be considered value data for the key.

Note that in many implementations of key type systems there is often some redundancy in the encoding of key material and FORM data. This makes sense from the point of view of extensibility and modularity of the code, but can lead to confusion when studying API specifications of data formats, in understanding which field is taken to be authoritative, and what to do in event of a mismatch. For example the key below might be stored with a *type* data field containing a constant which means 3-key 3DES, yet it also has a field stating the length of the key as 24 bytes. Using the 4AX model usually avoids this sort of redundancy issue.

```
KEY1    key type = 1 (3-key 3DES)
        key length = 24 bytes
        key value = ...
```

PSEUDO-FORM is also useful terminology (though not a laudable concept) – this is data concerning the interpretation of a key when it is not stored in atomic form in particular implementation. For instance, a 3DES key may have left and right halves, but if the API only keeps these separate for implementation convenience, the difference is only PSEUDO-FORM.

3.2 Use

USE is the most fundamental aspect of type. It captures the core of the designers intention as to what the key or data is *for*. Often the USE of a key is best described in English language, for instance “for communications” or “for signing certificates to approve new software patches”. USE is a static component of key type; each key has exactly one USE, and a key can never change USES.

More subtle changes of type which correspond to the dynamic aspects of the informal "use" of a key are captured by the ROLE concept. Example USES could include:

– for terminal communications
– for encrypting VPN links
– for deriving customer PINs from PANs
– for deriving unique device keys for embedded devices
– for encrypting other keys for offline encrypted backup
– for verifying PINs using the PVV method
– for verifying PINs using the IBM 3624 method

An interesting observation about USE is that it rarely comes for free just from the process of making an implementation. If you are coding a crypto API that will do AES, 3DES, RSA and elliptic curves, then chances are your data structures will include FORM data already – this is so if the user accidentally gives a handle to a 3DES key rather than an RSA key, then the code will spot it and fail gracefully rather than causing a nasty crash. Likewise if the crypto API supports multiple sessions (as a kind of DOMAIN), then temporary keys will probably include a session identifier so that one application doesn't muddle up its keys with another application. However, in a crypto API – an API that does not do policy enforcement – the API has no interest or need in storing the USE of the key; it is up to the application to keep track of what the key is *for*.

There are of course grey areas where the definition of use is difficult to evaluate. For instance IBM PIN derivation algorithms "IBM 3624" and "IBM 3624+offsets" are different algorithms, and though they work similarly, you would ideally like to prevent the offset functionality from acting on PINs that do not need to be changed. However, if you make 3624 and 3624+offsets different USES, then you preclude yourself from ever changing your mind and allowing people with fixed PINs to upgrade to a changeable PIN system. Therefore it might sometimes be acceptable to encode certain USE-like attributes in ROLE if in the long term you anticipate a migration might happen.

3.3 Role

ROLE is the dynamic component of key type. Example ROLES for a key could include "for encryption" or "for decryption", or "export to other domains permitted". To understand the idea of ROLE in key type, consider first a Role-Based Access Control (RBAC) system. Each real-world person holds one and only one identity, they are a single "user". However, that user's role within an organisation and within an IT infrastructure can change over time, and a user may assume multiple roles, possibly concurrently. On the other hand, a user who exists on paper, but has never actually operated the system might not have any roles allocated. Rather than considering the keys as the objects in a role-based access control system, we consider them as the users. A key has one and only one value, but it can take on zero or more roles, and these roles can change over time. With respect to key role, the API treats real world-people as resources, and decides whether or not a key (through assuming a particular role) should be granted access to a user. This reversal of perspective is of much more practical than focusing on roles of people, as the divisions there tend to be more straightforward[1].

There is a danger that ROLE may take on policy concepts that really are better put in USE. For instance a designer might create a ROLE "PIN verification" and a ROLE "Data Encryption" to describe possible tasks for a 3DES key. He could then have a rule saying that a key may not have the "PIN verification" role at the same time as the "Data encryption" role. Such rules on the role transitions are basically re-inventing the USE concept within the framework of ROLE: this is a bad thing. We thus constrain our definition to say that two ROLES must never be *conflicting.* That is, there must never be a rule saying that a key may not possess one ROLE if it possesses another. It is quite acceptable for a ROLE to be meaningless for a particular key USE, however. One rule of thumb (admittedly ugly) is to put as much of your type system as feasible into USE, without breaking its definition, and then put whatever is left-over into ROLE.

3.4 Domain

DOMAIN type information is concerned with *trust, risk, and responsibility.* DOMAIN information can be very obvious sometimes: this is **your** key, this is **my** key. They may both be stored in HSMs, having exactly the same USE, same FORM – yet one is for me to decrypt my email, and one is for you to decrypt yours.

Sometimes this DOMAIN information will be explicit – if we share the same HSM, we will likely need to log on as different user-IDs to access the different keys, and in the key type data itself the identity of the owning user may be stored. However, sometimes the DOMAIN information will be entirely implicit – for instance if we have different HSMs, with unique master keys used by each to encrypt externally stored data, then embedded in the implementation of the external storage is the identity of the master key, which in turn describes the ownership (or domain) of the key.

The DOMAIN of a key can of course change, but because this necessarily puts a new party in charge of trust and responsibility, then when DOMAIN changes we must rely on the good will and correct operation of the receiving party to maintain the type information.

Our definition of DOMAIN can capture access control used for virtualisation – multiple non-interactive API instances, but has it's limitations: it does directly capture the idea of a shared keys and data, but it can represent shared keys with an artificial domain for each pair of key-sharers.

[1] For example, IBM's Common Cryptographic Architecture (CCA) API has a full RBAC system for users, which is practically never used in genuine deployments.

4 Understanding API Attacks with the 4AX Model

Up until now, the only categorisation of attacks has been a crude one, focused on the general complexity of the methodology. This yields three categories:

– *Pure API Attacks* – are those that rely only on shortcomings of the API type system, and do not interact with the specific crypto algorithms or data formats being used. They can thus be modelled whilst retaining the perfect cryptography assumption often used in protocol analysis. Most of them have the character of protocol attacks.
– *Cryptographic API Attacks* – are those which exploit shortcomings of the cryptographic procedures used to bind type to the keys and data, and to protect the confidentiality and integrity of the data itself.
– *Information Leakage Attacks* – are those which operate usually on a single command to identify a particular response which is dependent on some secret data, and then repeat it to slowly leak more and more information about the secret.

However, the four-axis model yields a different, more illuminating categorisation of API attacks which – rather than considering the attack methodology – looks at what aspect of the type system (FORM, USE, ROLE or DOMAIN) is violated by the attack.

Classic attacks on key FORM are the 3DES key binding attacks [2]. In these attacks, a 3DES key is split into two single length DES keys allowing each to then be attacked independently, or conversely an apparent 3DES key can be conjured by repeating a single length DES key [2]. More generally, it is an attack against key FORM if keys intended for use with by a given strong cipher can be manipulated into being used by a weaker cipher [6]. Public key cryptography can be similarly vulnerable when the FORM information can be manipulated. For example, the re-interpretation of RSA key data structures incorrectly, yielding bad key data which rapidly leaks the original key when used to encrypt [10]. The tendency for attacks against FORM to result in the compromise of the key itself is a defining characteristic.

In contrast, violating the intention of the given USE or ROLE of a key typically compromises any data protected by that key, but can never compromise the key itself. Examples of attacks against USE include using the same key as input to competing PIN verification algorithms to recover the secret PIN [7]. The type-casting attack against the IBM CCA API [2] succeeded in creating a common key that could be used to encrypt and export other keys, and then treat the result as data ciphertext which it would decrypt.

Manipulating the DOMAIN of a key changes the identity of the entity responsible for the key (the entity whom you trust to keep the key secure). Attacks on DOMAIN thus work to seize access to a key, for instance if an open handle in one users session can be hijacked by another user. If the destination DOMAIN is entirely controlled by the attacker, the secrecy of the key will of course be compromised; however some APIs enforce policies that no-one can bypass, no matter what their authority. Therefore the destination DOMAIN may permit abuse of the key, but not extraction to cleartext.

5 Describing APIs with the 4AX Model

We studied an extensive range of Security APIs before building the 4AX model. Amongst these were financial APIs, including IBM's Common Cryptographic Architecture (CCA), the VSM and RG7000 APIs, the Atalla NSP8000 series API, and the nCipher payShield API. We looked at key management APIs for certification authorities and general purpose policy specification, such as RSA's PKCS#11 and the nCipher nCore API. Finally, we considered application specific APIs, including that of the Prism TSM200 and the Trusted Computing Groups TPM specifications. There is not room to explore in detail how the type system from each API compares to our four axes: we will give a case study of the IBM CCA, and try to briefly bring out the salient features of other APIs.

5.1 The IBM Common Cryptographic Architecture

IBM's CCA architecture was a strong influence in the design of our model. It uses a highly condensed representation of type – the control vector, which is a 64 bit string[2]. Yet despite such a brief representation, the subcomponents of a control vector map reasonably well to the 4AX axes, and are capable of fine-grain control over usage policies. IBM splits the bits of a control vector into categories: the *main key type*, the *subtype*, the *usage restrictions*, and the *form*. The CCA assembles together default configurations of the more complex components of the type system to make what it calls "generic key types" – a simple default type system within a complex one. This secures the API against only simple attacks, but makes programming and legacy compatibility easier.

Main key type is close to 4AX USE. It identifies four broad categories of usage: "data operation", "PIN processing", "key-encrypting", and "key generation". However, because some keys with really quite fundamentally different algorithmic uses are clumped together, it can be seen that the idea behind the main key type is more to group together sorts of keys which are commonly used with particular subsets of the transaction set (e.g. all keys pertaining to the task of processing PINs fall under the same "main key type", even though a working key for transmitting PIN blocks performs a very different operation to a PIN derivation key.

The key *subtype* corresponds partly to ROLE, for instance in the case of key-encrypting-keys, where the subtype differentiates between KEKs for import and KEKs for export; and partly to USE: for instance the category "PIN keys" described earlier is subdivided into particular uses using the key subtype. All asymmetry of keys in the CCA is implemented the subtype level.

[2] Control Vectors are described in detail in Appendix C of the CCA Basic Services Manual [9]

The *key form* bits encode 4AX FORM (at least as far as symmetric keys are concerned), but go past this level in that they stray into implementation details – PSEUDO-FORM. The form of a key can either be "DES" or "3DES", will some implementation specific data as well, such as *left or right half*, and two storage forms for single DES keys (*single-length* or *double-length replicate*), to aid with legacy system compatibility programming.

DOMAIN information in the CCA is largely implicit. Ownership, responsibility and trust associated with a key are bound via the master key under which the key is encrypted. The CCA works with only a single DOMAIN[3]. DOMAIN concepts include overall responsibility, bound in the overall master key used, domains for key life-cycle management (old, current and new master key registers), and domains of mutual trust and responsibility, captured implicitly in the specification of a shared transport key.

5.2 Further PIN Processing and Payment APIs

The Visa Security Module (VSM) and RG7000 share a very simple type system, compared to the CCA. There are approximately twelve types, each represented entirely implicitly, through encryption under one of twelve pairs of master keys. With such a limited set of master keys only a coarse granularity of type exists, roughly corresponding to USE in the 4AX model. Originally all keys were symmetric single DES keys, so there is no FORM information, neither is there the complexity of ROLES. The RG7000 since added limited support for public key cryptography in an ad-hoc manner, and uses key variants to make some basic PSEUDO-FORM distinctions.

Meanwhile Atalla's NSP Series HSMs use an integer number to represent type (ranging from 0–31), and also use implicit typing of keys using variants. The particular variant (XORed with the master-key for a type) distinguishes between DES and 3DES keys, and left and right halves of 3DES keys. The integer range of types could be considered as a flattened 4AX system: for example types 16 and 17 both have the same USE for crypto operations on foreign data, but one is encrypt-only, the other decrypt-only. Meanwhile key 19 verifies MACs but key 20 is totally different – a PIN encryption key.

The API to Prism's TSM200 is unusual in that it departs from the traditional external key storage paradigm, instead storing up to 100 keys in registers in the clear within the HSM. The type information is thus held explicitly in the register, rather than being cryptographically bound as part of an encrypted token. The type metadata stored is as follows:

– Key Length (Single DES or Triple DES)
– Key Type (One ASCII Char)
– Parent Key Register ID (00-99)

[3] On mainframe installations, an entire HSM can be virtualised, implemented at the API level, but it is presented to the host as multiple physical devices.

The "Key Length" field corresponds to FORM, creating a binary classification of keys into DES/3DES. The "Key Type", ranging from A–Z, types corresponds almost exactly to USE; the exception is that certain types correspond to FORM information, marking them as right halves of 2-key 3DES keys. There is no usage of ROLE in the TSM200 API, and there is implicit DOMAIN in the master key, and in the hierarchy position of the parent key.

5.3 Public-Key APIs

Public-Key APIs are those which store primarily asymmetric keys, and use them regularly in key management functions. RSA's PKCS#11 is an important member of this category, though it is more of a framework than a fully fledged API. Implementers tend to add their own rules and extensions for key manage within the framework, but the framework ensures enough basic compatibility to permit standardised application access to signing keys. Typing metadata attached to keys consists of attribute and value pairs. The `CKA_KEY_TYPE` attribute is the top-level representation of FORM in PKCS#11. Particular key types will have additional attributes which constitute part of their FORM, dependent upon the sort of key (e.g. RSA keys have a `CKA_MODULUS_BITS` attribute). PKCS#11 does not support a broad range of default USES for keys, basically only data confidentiality, data integrity, wrapping and unwrapping (aka. import/export) and key derivation, which it marks through attribute fields which are set to `true` or `false`. It does however have multiple ROLES within these USES to support asymmetric links, and flow control.

A final interesting concept in PKCS#11 is that of LINEAGE : the API maintains a crude history of the attribute values of a key. It does this through `ALWAYS_SENSITIVE` and `NEVER_EXTRACTABLE` attributes which chart whether or not a key has been "always sensitive" (it has always had their `CKA_SENSITIVE` attribute set to true), and "never extractable" (it has never had its `CKA_EXTRACTABLE` attribute set to true).

nCipher's nCore API is another public-key centred API, and a very sophisticated one. However, much of its complexity relates to access control policies for keys, an issue which is outside the scope of the 4AX model. We did experiment with extending the 4AX system to model access control to keys, but found that the added model complexity undermined the advantages that can be gained from working with the model – those of security and clarity of understanding. Each key structure handled by the nCore API has a *type* and an *Access Control List (ACL)*.

The `key type` field is the closest corresponding concept to FORM. The key type specifies the algorithm that must be applied and includes specific form data such as RSA key length. Extra FORM data is carried in key *mechanism* restrictions – a combination of algorithm, padding and cipher mode of operation restrictions. The above are set during creation of a key and cannot be changed. The nCore API also binds an ACL to a key: this is a complex structure, split into permission groups, each of which has an owner, and possible restricted applicability to a specific nCipher device. The ownership and device information is DOMAIN information in 4AX model terms. The permission group itself contains ROLE and USE data.

As a consequence, the nCore API can present a key of the same value as having different different types when presented to different owners. Such a feature is well beyond the scope of the 4AX model: upon encountering such unprecedented flexibility, it begs the question of what this feature achieves, and whether it is really necessary. In the 4AX model FORM and USE are immutable, whereas the nCore API chooses only to permanently bind FORM. One last crucial feature of the nCore API type metadata does not normally reside with the key – each key has a *generation certificate* which attests to the ACL settings first bound to the key upon generation. We call this concept key ORIGIN.

This summary only scratches the surface of the nCore ACL system, which supports global and local usage limits, multiple virtual DOMAIN hierarchies, disjoint or overlapping, and special dedicated handling of archival and migration permissions.

The Trusted Computing Group's *Trusted Platform Module (TPM)* is a low-cost HSM which can be used to strengthen desktop and mobile computing platforms, by providing secure storage, measurement and attestation functionality. It has a large API due to the measurement functionality, but relatively simple key management, based purely on public-key cryptography. Keys for immediate use are stored internally in the clear, and externally in encrypted form in a hierarchy built from the "Storage Root Key" (SRK) which remains in a special register in the clear, or from a completely arbitrary root key. Key structures contain the `TPM_KEY_USAGE` attribute, which is a perfect match to 4AX USE. It separates keys used for general-purpose signing, secure storage, identity management and binding. In addition to access control information for the key (which is outside the scope of 4AX), a set of `TPM_KEY_FLAGS` exist, some of which are access control relevant, and others corresponding directly to key ROLES: for instance, there is an export permission flag denoted `migratable`. FORM is largely implicit as all stored information is either an RSA keys or entirely foreign block data; there is also FORM in the RSA key length field within the `KEY_PARAMS` block. DOMAIN information is explicitly encoded in the `auth` and `migrationauth` fields, which contain passwords belonging to the identities who have permission to use (or migrate) a specific key.

6 Conclusions

We began this paper by looking at the reasons why type information becomes necessary to keep API inputs secure, and how it can grow in quantity and complexity. We believe that the decisions an API designer makes in how they structure their type information are of fundamental importance to the security and usability of the API, especially given incremental development or evolution of the API over its life-cycle.

Our four-axis model then proposed terminology and bounds within which an API designer can operate safely and coherently – this is not the same task as trying to develop a brand new approach to expressing key type. Indeed many of the concepts we use are not novel: we have tried to take the best of every API we have seen. The four-axis definitions of FORM, USE, ROLE and DOMAIN are a sterilised set of axes, manufacturer and implementation neutral, which can hopefully be a useful point of reference when designing for security. As demonstrated in sections 4 and 5 they can cast light on the principles underlying API vulnerabilities, and on unusual or potentially risky features of existing APIs. The reader should thus be armed with some fresh generic concepts to help on the journey to properly understand and conquer security API design.

7 Acknowledgements

We would like to acknowledge the generous funding of the CMI Institute and the Cecil Renaud Educational and Charitable Trust.

References

1. R. Anderson, "The Correctness of Crypto Transaction Sets", 8th International Workshop on Security Protocols, Cambridge, UK, April 2000
2. M. Bond, "Attacks on Cryptoprocessor Transaction Sets", CHES 2001, Springer LNCS 2162, pp. 220-234
3. M. Bond, "Understanding Security APIs", PhD Thesis, University of Cambridge, Jan 2004
4. M. Bond, R. Anderson, "API-Level Attacks on Embedded Systems", IEEE Computer, Oct 2001, Vol 34 No. 10, pp. 67-75
5. M. Bond, J. Clulow, "Encrypted? Randomised? Compromised? (When Cryptographically Secured Data is Not Secure)", Cryptographic Algorithms and Their Uses, Eracom Workshop 2004, Queensland Australia
6. J. Clulow, "On the Security of PKCS#11", CHES Workshop 2003, Cologne, Germany, LNCS 2779 pp. 411-425
7. J. Clulow, "The Design and Analysis of Cryptographic APIs for Security Devices", MSc Thesis, University of Natal, SA, 2003
8. V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, R. E. Bryant, "Automatic Discovery of API-Level Vulnerabilities", UW-Madison Computer Sciences Technical Report UW-CS-TR-1512, Jul 2004, http://www.cs.wisc.edu/ vg/writings/papers/tr1512.pdf
9. IBM, 'IBM 4758 PCI Cryptographic Coprocessor – CCA Basic Services Reference and Guide, Release 2.41 for the IBM 4758-002', available through http://www.ibm.com/security/cryptocards/
10. V. Klima, T. Rosa, "Attack on Private Signature Keys of the OpenPGP format, PGP programs and other applications compatible with OpenPGP", 22nd March 2001
11. D. Longley, S. Rigby, "An Automatic Search for Security Flaws in Key Management", Computers & Security, March 1992, vol 11, pp. 75–89
12. G. Steel, "Deduction with XOR Constraints in Security API Modelling", 20th Conference on Automated Deduction (CADE 20), July 2005, pp. 322–336