Model Checking IBM’s Common Cryptographic Architecture API

by

Gavin Keighren
Abstract: We present results from the application of a model checker to the analysis of the API used by a number of security modules in Automated Teller Machine networks — IBM’s Common Cryptographic Architecture API. We show that it is capable of rediscovering all known attacks on the API, using models containing a greater set of API commands.

We also analyse the set of recommendations released, in response to one of the discovered attacks, by IBM and show that, under certain assumptions, they do not prevent the attack. We use a revised set of assumptions, under which the attack is prevented, to determine a number of our own recommendations primarily aimed at the design and implementation of the API.

Finally, we discuss various issues concerning the analysis of security APIs, based on our experiences of carrying out the work presented.

Keywords: Model Checking, Security APIs, Verification, Formal Methods.
1 Introduction

IBM’s Common Cryptographic Architecture (CCA) API \[4\] is used by hardware security modules (HSMs) in a significant number of automated teller machines (ATMs) across the world, as well as in the mainframe computers of many banks. The API is provided by IBM’s 4758 Cryptographic Coprocessor whose main task, in this setting, is to carry out PIN verification requests, although a large percentage of the commands facilitate the transfer of secret cryptographic keys in order to initialise a new device.

The CCA API, in line with other security APIs, is designed to carry out a series of operations involving sensitive data. The sensitive data is typically in an encrypted form outside of the security module, and only appears in the clear within the device itself. The goal of a security API is therefore to ensure that any sensitive data is not made available outside of the HSM itself, and that the data is only manipulated in a precisely controlled manner.

However, in 2001 Mike Bond discovered flaws in IBM’s CCA API which, among other things, allowed an attacker to obtain the PIN for any given account number \[1\]. The attacks were discovered through detailed analysis of the API by hand, and resulted from the manner in which it utilises the exclusive-or (XOR) function. Bond attempted to rediscover the attacks using the theorem prover SPASS, but due to the algebraic properties of XOR, was only able to do so using a simplified model, or by guiding the search.

Since then, researchers have been trying to develop verification tools and methods that are able to model security APIs and discover attacks automatically. It has only been very recently though that tools have been developed which handle XOR in an efficient manner, and thus are better suited to analysing security APIs.

Even still though, there are currently no tools which are designed specifically for the analysis of security APIs, only the closely related task of analysing security protocols — typically a more constrained problem.

1.1 A Note About PINs

The Personal Identification Number (PIN) for an account is calculated from the primary account number (PAN) by encrypting it under a DES key known as the PIN derivation key. The hexadecimal result is then converted into a four digit number by truncating it, and mapping the digits A-F to binary digits using a decimalisation table. The following example is taken from \[2\]:

<table>
<thead>
<tr>
<th>ACCOUNT NUMBER</th>
<th>4556 2385 7753 2239</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCRYPTED A/C NO.</td>
<td>3F7C 2201 00CA 8AB3</td>
</tr>
<tr>
<td>TRUNCATED VALUE</td>
<td>3F7C</td>
</tr>
<tr>
<td>DECIMALISATION TABLE</td>
<td>0123456789ABCDEF</td>
</tr>
<tr>
<td>DECIMALISED PIN</td>
<td>3572</td>
</tr>
</tbody>
</table>

This method of calculating PINs was developed by IBM in the 1970s, and meant that the PIN did not have to be stored in the magnetic strip of bank cards — ATMs would verify the PIN entered by carrying out the above process and comparing the result. The flaws discovered by Bond allow an attacker to encrypt arbitrary data (i.e. a PAN) under the PIN derivation key, and thus obtain the associated PIN — thus violating one of the security goals of the API.

2 Overview of IBM’s Common Cryptographic Architecture API

At its heart, the CCA is a key management system, which provides commands that use encrypted keys to achieve desired functions. A 168-bit triple-DES key, known as the master key, is stored in the security module’s tamper proof memory and is used to encrypt all other keys which are then kept on the host computer. These other keys, known as working keys, are used to perform the various functions provided by the CCA API, and have types associated with them.

In all, the CCA supports the following functions and features:

---

1\[http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml]
• Encryption and decryption of data, using the DES algorithm [7].
• Message authentication code (MAC) generation, and data hashing functions.
• Formation and validation of digital signatures.
• Generation, encryption, translation and verification of PINs and transaction validation messages.
• General key management facilities.
• Administrative services for controlling the initialisation and operation of the security module.

As a number of the provided commands are particularly sensitive, the CCA enforces an access-control system, whereby certain commands are only available under specific circumstances. It is, however, the responsibility of the device administrator to ensure that the correct separation of duty provided is upheld.

2.1 Working Keys

The CCA API uses four main types for classifying DES working keys, each of which is further sub-divided into more specific and restrictive types. Each type takes the form of a control vector — a bit-string that is the same length as the associated working key. A working key is stored outside of the security module, encrypted under the exclusive-or of the device’s master key and the control vector representing the type of the key. The main key types, and their uses, are as follows:

Data Keys
Keys of this type are used to encipher and decipher arbitrary data, as well as for the generation and verification of message authentication codes (MACs). Subtypes place greater restrictions on exactly which of these various functions a particular key can be used for.

PIN Keys
This type covers keys which are used for PIN block encryption, PIN block decryption, PIN generation and verification, and just PIN verification. A key cannot be of the general PIN type, but instead must be assigned a subtype that restricts its use to exactly one of the four operations mentioned.

Key Encryption Keys
These keys are used to encrypt and decrypt other working keys during transfer between security modules, and are divided into import and export types. Keys encrypted under an export key are referred to as external keys, as they must be imported into a security module before they can be used. Note that the transfer of a working key requires the same key encryption key to be present in both security modules — as an export key in one and as an import key in the other.

Key Generation Keys
The CCA API provides commands which generate DES keys, given an initial key, and will typically use the provided key to encrypt or decrypt a supplied piece of data. This type covers such initial keys and restricts them from being used with other commands (e.g. Encipher and Decipher) in order to prevent the value of the generated key being discovered.

The typing mechanism restricts the working keys which can be used for a particular command, for example, the PIN derivation key used in the verification of a customer’s PIN cannot be used with the Encipher command to encrypt arbitrary data.

2.2 API Commands

In this section, we present the core key management commands, along with some selected others that have traditionally been included in models of the API. The justification for only considering the commands listed is given in Appendix B. Note that the definitions are designed to capture the important semantics of the commands, rather than their precise syntax.

---

[2] See Appendix A for an overview of the notation used to describe the commands.
The following terms are used in this section to represent the various control vectors and cryptographic keys:

- **DATA**: Control vector for data keys
- **IMP**: Control vector for import-type key encryption keys
- **EXP**: Control vector for export-type key encryption keys
- **KGN**: Control vector for key generation keys
- **KP**: Control vector indicating that a key is only a key part, and not a complete key
- **KM**: The security module’s master key
- **kek**: An arbitrary key encryption key
- **key**: An arbitrary cryptographic key
- **new**: An unknown, randomly generated, new cryptographic key
- **type**: An arbitrary key type control vector
- **kpi**: Key part i (used to build an arbitrary key)
- **x**: Arbitrary (unencrypted) data

The exact steps that the security module carries out for each command have not been included, since the process is virtually the same in all cases. The master key and all control vectors are known to the security module, and any additional information required is either passed as a plaintext parameter, or is encrypted under a known key. For example, in the case of the *Key Import* command, the security module knows both KM and IMP so is therefore able to obtain kek from the third parameter. This key encryption key is then XOR-ed with the second parameter, type, and used to obtain key from the first parameter. Finally, key is encrypted under the exclusive-or of KM and type to produce the result that is returned by the command.

**Encipher**

User → HSM: \(x, \{key\}_{KM\oplus DATA}\)

HSM → User: \(\{x\}_{key}\)

Encrypts given plaintext with the supplied data key. The data key can either be of the general type, or one of the subtypes that permits data ciphering.

**Decipher**

User → HSM: \(\{x\}_{key}, \{key\}_{KM\oplus DATA}\)

HSM → User: \(x\)

Decrypts ciphertext which has been encrypted under the supplied data key. The data key can either be of the general type, or one of the subtypes that permits data deciphering.

**Key Import**

User → HSM: \(\{key\}_{kek\oplus type}, \text{type}, \{kek\}_{KM\oplus IMP}\)

HSM → User: \(\{key\}_{KM\oplus type}\)

Converts a key (of the given type) from encryption under the supplied import-type key encryption key to encryption under the local master key. Although not explicitly mentioned in the manual, the key being imported must be a complete key (i.e. it cannot be a key part).

**Key Export**

User → HSM: \(\{key\}_{KM\oplus type}, \text{type}, \{kek\}_{KM\oplus EXP}\)

HSM → User: \(\{key\}_{kek\oplus type}\)

Converts a working key from being encrypted under the local master key, to being encrypted under the supplied export-type key encryption key. Note that key must be able to be exported.
Key Part Import

**FIRST**

- **User → HSM:** $kp_1$, type
- **HSM → User:** $\langle kp_1 \rangle_{KM\oplus KP\oplus type}$

**ADD / MIDDLE**

- **User → HSM:** $\langle kp_1 \oplus kp_2 \rangle_{KM\oplus KP\oplus type}$
- **HSM → User:** $\langle kp_1 \oplus kp_2 \rangle_{KM\oplus KP\oplus type}$

**COMPLETE**

- **User → HSM:** $\langle kp_1 \rangle_{KM\oplus KP\oplus type}$, type
- **HSM → User:** $\langle kp_1 \rangle_{KM\oplus KP\oplus type}$

This series of commands builds up a working key from individual parts and can be used in one of two ways. Either the ‘first’, ‘middle’ and ‘last’ commands can be used, or the ‘first’, ‘add’ and ‘complete’ ones. In order to provide security through separation of duty, the commands are split into three groups, with individuals only allowed access to one. However, the ‘add/middle’ and ‘last’ commands are in the same group, so the combination using those ones allows a malicious insider to obtain a pair of keys with a known difference (a prerequisite for a number of the attacks discovered by Bond). The other series of commands ensures that the people responsible for inserting the key parts cannot obtain a final key, and the person who obtains the final key cannot modify it in any way.

**Key Translate**

- **User → HSM:** $\langle kek_1 \rangle_{type}$, $\langle kek_1 \rangle_{KM\oplus IMP}$, $\langle kek_2 \rangle_{KM\oplus EXP}$
- **HSM → User:** $\langle kek_1 \rangle_{type}$, $\langle kek_2 \rangle_{type}$

Translates a key from encryption under one key encryption key to encryption under another. The first key encryption key must be of import type, and the latter must be of export type.

**Key Generate**

- **User → HSM:** type1 $\langle \cdot \rangle$, type2 $\langle \cdot \rangle$
- $$\langle \langle kek_1 \rangle_{KM\oplus IMP} \cdot \langle kek_2 \rangle_{KM\oplus EXP} \rangle$$
- **HSM → User:** $\langle \langle \cdot \rangle_{KM\oplus type} \rangle$
- $$\langle \langle \langle \langle \cdot \rangle_{KM\oplus IMP} \cdot \langle \langle \cdot \rangle_{KM\oplus EXP} \rangle \rangle \rangle \rangle$$

The **Key Generate** command has nine variants, each of which returns one or two copies of a randomly generated key, each with their own type, and each encrypted in one of three possible ways. A generated key can be encrypted under the local master key (for immediate use by the security module), it can be encrypted under a supplied import-type key encryption key (for re-importation and use at a later date), or it can be encrypted under an export-type key encryption key (for importation into another security module). In the case where two keys are generated, the combination of types is restricted — usually such that the two keys perform inverse operations, e.g. encryption and decryption.

When considered, we only included the simplest variant in our models — the one which returns a single key of the desired type, encrypted under the master key. See §4.4 for the reasoning behind this.

### 2.3 Access Controls

As a security measure, the CCA software provides role-based access controls, so as to limit the commands which any particular user has access to, as well as place restrictions on when they are able to access the system. A user of the security module is assigned to be a member of one of the specified roles, and once logged in, inherits the privileges defined therein. This approach is in contrast to the alternative method of keeping individual user profiles, and has been chosen since it is often the case that a large number of users will share the same access rights.

The security module is required to have a default role which defines the capabilities of any user who has not logged on and authenticated with the system. Additional roles can be defined by the device admin-

---

3A user can either be a human individual or a computer process.
istrator, and are stored internally by the security module. With respect to the key management commands, the following roles will typically be defined:

**General User**
This role will have very few extra privileges over the default role, and will typically only allow the use of existing keys with certain operations, e.g. data keys with the *Encipher* and *Decipher* commands.

**Security Officer 1**
IBM recommend that the process of importing a new key from clear key parts should be carried out by different users whose individual capabilities are not sufficient to mount an attack. The key parts are typically loaded into a device by security officers, and this role only allows the use of the *Key_Part_Import* command to insert the first key part.

**Security Officer 2**
Users assigned to this role are only able to add subsequent cleartext key parts to an existing encrypted incomplete key.

**Security Officer 3**
In response to the attacks discovered by Bond, IBM modified the API in such a way as to separate the ability to add new key parts and to generate the completed key. The third security officer is not able to modify the incomplete key, but can turn it into a complete key, as well as verify that the completed key has not been tampered with.

It will also be likely that there are other roles which allow existing working keys to be exported or modified, and that allow new keys to be generated. In general, it is best if users are only provided with the bare minimum of privileges required to carry out their task, and that potentially dangerous combinations of commands are not enabled in the same role.

### 3 Modelling the API

The way in which the API is modelled depends on the tool, or tools, that will be used to carry out the subsequent checking. Recent research [12] has shown that attempting to formally verify security APIs which employ the exclusive-or function requires tools that handle the associated algebraic properties in an efficient manner. To this end, we selected the model checker CL-AtSe [16], although it was not the only factor in the decision process. CL-AtSe was developed as one of the back-ends in the Avispa tool set [15], which was specifically designed to be used for the analysis and verification of security protocols. It accepts models written in the High Level Protocol Specification Language (HLPSL) [9], which was also developed as part of the Avispa project. HLPSL is intended to be geared towards the modelling of communication and security protocols, and provides important primitives such as communication channels, cryptographic keys and security properties.

#### 3.1 HLPSL

In security protocol analysis, the typical scenario consists of two honest agents, and an *intruder* agent who is attempting to obtain secret information, or pose as an honest agent. In HLPSL, the protocol is modelled as a series of *roles* which are played by one or more of the honest agents, and possibly the intruder. The roles contain a series of transitions which define the behaviour of the particular agent playing that role.

The main roles are composed in a special role called the *session* which describes when they are active. It is usually the case that the roles are composed in parallel, meaning that they are all active at the same time. The session role is also used to define what roles are played by which agents, and communication protocol models will typically have more than one session with different agent-role assignments in each.

The agents interact with each other by passing messages, of a predetermined structure, across one or more communication channels. Currently, HLPSL only provides channels based on the Dolev-Yao model, where the intruder is able to eavesdrop on everything transmitted — meaning that all communication is effectively done via the intruder. Therefore, all unencrypted terms transmitted will be added to the intruder’s
knowledge, as will encrypted terms if the intruder has the necessary key to decrypt them. Furthermore, the intruder can potentially modify the contents of transmitted messages, as well as creating his own transmissions, and preventing others reaching their destination.

Two types of security properties are provided: secrecy of terms, and authentication on terms. The former allows for the specification of the agents for which knowledge of a particular term, or terms, is acceptable. If another agent, usually the intruder, obtains the information then the property has been violated. The latter property lets you specify that two communicating agents agree on the value of a particular term, or terms. This is typically used to check whether or not two agents can communicate securely — a violation occurs if one of the two parties is unknowingly communicating with a different agent than is intended.

As well as the communication channels and secrecy properties, HLPSL also provides a number of primitive types that are typically required in protocol analysis, including symmetric keys, public keys, and agents.

The model checkers do not work with HLPSL directly, but rather a lower level representation called the Intermediate Format [11]. This is automatically generated from the original HLPSL model, and models each transition as a rewrite relation on the intruder’s knowledge. That is, if the intruder knows the terms on the left-hand side, then he can obtain the terms on the right-hand side. This corresponds to the intruder forming a message using the terms on the left-hand side, sending it to the appropriate honest agent, and receiving the terms on the right-hand side in response. Other issues such as the internal state of the honest agents are modelled as additional pre-conditions of the specific relation.

3.2 CL-AtSe

CL-AtSe (Constraint Logic based Attack Searcher) has in-built support for a number of algebraic operators, including exclusive-or. Initially, a set of constraints are obtained which precisely define the transitions in the model, and during the search process, the effect of a particular transition occurring is computed by adding new constraints, as well as modifying or deleting existing ones. At each step, the constraints are checked in order to determine if any of the security properties have been violated.

The intruder is able to carry out off-line encryption and decryption using known terms, as well as being able to obtain the exclusive-or of any two known terms. These abilities are not provided explicitly, but rather are consequences of the unification algorithm employed. For example, if the exclusive-or of three terms is required to unify two equations (in order for the intruder to effect a transition) then they will unify if the intruder has those three terms as part of his knowledge.

The verification process is carried out for a bounded number of sessions, i.e. the bound determines the maximum number of times that each role can be run. Unfortunately, although HLPSL provides various cryptographic primitives, CL-AtSe only applies the exclusive-or operator to terms of the generic message type. This means that all models have had to be run with the typing information ignored — thus resulting in a larger search space as there are fewer restrictions on term unification.

3.3 API Representation

Unlike standard security protocol analysis, where the intruder is attempting to break a secure communication between two honest agents, the attack scenario for the CCA API consists only of the security module and the intruder. Furthermore, the security module is essentially stateless in that the result of a command only depends on its inputs. A consequence of this property is that there is no enforced ordering on the execution of the commands.

To begin with, the API commands were modelled as separate transitions within a single role, but this was problematic for CL-AtSe’s initial optimisation steps. Each role is explored to the full depth in order to determine any simplifications that can be made, but the non-deterministic nature of the transitions caused the runtime of this procedure to blow up.

To avoid this problem, the commands were instead modelled as separate roles, each containing just the one transition. Figure [1] shows how the KeyImport command was modelled. Recall that the KeyImport command takes as input the key to be imported (encrypted under a key encryption key), the type of the key,
role keyImport(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM

def=

local
K1 : symmetric_key, % An arbitrary key
KEK1 : symmetric_key, % An arbitrary key encrypting key
TYPE : nat % An arbitrary key type control vector

transition

keyImport.

Rcv({K1'}_xor(KEK1',TYPE').TYPE'.{KEK1'}_xor(km,imp_CV))
\ in(TYPE',KeyTypes)
=> Snd({K1'}_xor(km,TYPE'))

end role

Figure 1: The Key Import command modelled in HLPSL.

and the key encryption key. The security module then modifies the key so that it is encrypted under the local master key, and returns the result. These requirements are captured in the transition by the message patterns in the Snd and Rcv channels, where a primed value represents a free variable, a period denotes message concatenation, and {X}_{Y} means that X is encrypted under Y. Because the channels are based on the Dolev-Yao model, the intruder is able to send anything to the security module agent, and receive anything that it returns, and is therefore able to make use of the commands. As all of the commands are available at any given time, they were composed in parallel within the session role.

Full and partial keys in the CCA must belong to exactly one of the major key groups, and this is achieved by ensuring that the value of the variable TYPE' is suitably constrained. Although not explicitly documented in the manual, the majority of the commands will only accept full keys. The exception to this rule is the Key Part Import command which only accepts key parts, and the Key Test command which accepts either.

As noted earlier, CL-AtSe only considers the exclusive-or of terms that are of the generic message type. However, the API is modelled using the more specific primitives as a guide to the actual uses of the various terms.

The intruder’s initial knowledge contained all of the public control vectors and any other information which a legitimate user of the security module would have.

Appendix C contains a full listing of all the roles used in the experiments.

4 Rediscovery of Known Attacks

There are three significant attacks which have been found on IBM’s CCA API: Bond’s Key Import Attack\textsuperscript{4}[1, §5.1], his Import / Export Loop Attack [1, §5.2], and IBM’s attack which exploits the fact that the data control vector is actually zero (presented in [5, §2.7.2]). While strict procedural controls are able to prevent the attacks, the models provide concrete evidence of dangerous command combinations.

All attacks are presented as the intruder obtaining the PIN from an arbitrary known primary account number. While this requires an extra step in each attack, it is the most likely conclusion of the attacks within the financial setting, and therefore more realistic. Furthermore, this means that the attacks provide concrete evidence of one of the two main security goals of the API being broken — ensuring the secrecy of customer PINs. IBM’s attack also breaks the other main security goal — preventing the clear value of any cryptographic key being discovered.

\textsuperscript{4}Bond terms this attack the “Chosen Key Difference Attack on Control Vectors”.

7
4.1 Key Import Attack

The Key Import Attack can be carried out when a new key is to be transferred to a security module, and requires a modified key encryption key (KEK) that has a known difference with the one used to encrypt the key being imported. The attack is carried out as follows:

1. ‘LAST’ KEY PART IMPORT

\[
\text{User} \rightarrow \text{HSM}: \text{KP3} \oplus \text{oldType} \oplus \text{newType} , \{ | \text{KP1} \oplus \text{KP2} | \} \oplus \text{IMP} \oplus \text{KP} , \text{IMP} \\
\text{HSM} \rightarrow \text{User}: \{ | \text{KP1} \oplus \text{KP2} \oplus \text{KP3} \oplus \text{oldType} \oplus \text{newType} | \} \oplus \text{IMP} \oplus \text{KP}
\]

The intruder, who is responsible for adding the final key part, XORs in a known difference to his key part. That difference is the XOR of the original and desired control vectors of the key whose type is to be changed. He then uses the ‘last’ Key Part Import command to add in the altered key part and obtain the modified KEK.

2. KEY IMPORT

\[
\text{User} \rightarrow \text{HSM}: \{ \text{key} \} \oplus \text{oldType} , \text{newType} , \{ | \text{key} | \} \oplus \text{IMP} \oplus \text{KP} \\
\text{HSM} \rightarrow \text{User}: \{ | \text{key} | \} \oplus \text{IMP} \oplus \text{KP}
\]

Next, he imports the key being transferred, using the modified KEK, and claiming that it has the desired control vector.

3. ENCIPHER

\[
\text{User} \rightarrow \text{HSM}: \text{PAN} , \{ | \text{PDK} | \} \oplus \text{IMP} \oplus \text{KP} \\
\text{HSM} \rightarrow \text{User}: \{ | \text{PAN} | \} \oplus \text{IMP} \oplus \text{KP}
\]

If the key being transferred is a PIN derivation key, then this attack could be used to change it to a data key and thus allow it to be used to encipher arbitrary data. If the arbitrary data was a primary account number (PAN), then the result would be the PIN for that account (see §1.1).

The attack works because of the manner in which the HSM processes the different parameters of the Key Import command. Initially, the third packet is decrypted to obtain the (tampered) KEK, which is subsequently XOR-ed with the provided control vector, DATA. Due to the cancellation properties of XOR, this results in KEK $\oplus$ PIN, which can then be used to correctly decrypt the PDK. Finally, the PDK is output as a key of the given type (i.e. DATA).

4.2 Import / Export Loop Attack

The Import / Export Loop Attack works by first exporting a key from the security module, then changing its type as it is re-imported, using the same method as the Key Import Attack. The attack proceeds as follows:

1. ‘LAST’ KEY PART IMPORT

\[
\text{User} \rightarrow \text{HSM}: \text{IMP} \oplus \text{EXP} , \{ | \text{UKEK1} | \} \oplus \text{IMP} \oplus \text{KP} , \text{IMP} \\
\text{HSM} \rightarrow \text{User}: \{ | \text{UKEK1} \oplus \text{IMP} \oplus \text{EXP} | \} \oplus \text{IMP} \oplus \text{KP}
\]

By providing a random value as the existing key part, the ‘last’ Key Part Import command can be used to conjure a pair of related keys. The command is used a second time with zero as the new key part to turn the conjured key part into a working key as well. Alternatively, the pair of related keys can be obtained when using the ‘last’ Key Part Import command to build up a key encryption key from separate parts, where UKEK is replaced by the XOR of the preceding key parts.

2. KEY IMPORT

\[
\text{User} \rightarrow \text{HSM}: \{ | \text{UKEK2} | \} \oplus \text{IMP} \oplus \text{KP} , \{ | \text{UKEK1} | \} \oplus \text{IMP} \oplus \text{KP} \\
\text{HSM} \rightarrow \text{User}: \{ | \text{UKEK2} | \} \oplus \text{IMP} \oplus \text{KP}
\]
The second step is to use the Key Import command to conjure a key part in two forms — one encrypted under the master key, and one encrypted under the supplied key encryption key. The command is then used again, this time with the other key encryption key from step 1, to obtain UKEK2 as an export key part.

Note however, that this step is impossible in practice, as the Key Import command will not accept key parts. However, this impossibility is not documented in the CCA manual and only came to light in 2003, when Paul Youn presented a series of potential new attacks to IBM (see [13, §4.1]).

3. Key Export

\[
\begin{align*}
\text{User} & \rightarrow \text{HSM}: \{[\text{key}]_{\text{KM} \oplus \text{type}}, type, [\text{UKEK2}]_{\text{KM} \oplus \text{EXP}} \} \\
\text{HSM} & \rightarrow \text{User}: \{[\text{key}]_{\text{UKEK2} \oplus \text{Type}} \}
\end{align*}
\]

Having turned the export-type key part obtained in the previous step into a complete key, using the ‘Last’ Key Part Import command, the intruder can then export the desired key from the security module, providing that it is able to be exported.

4. Key Import

\[
\begin{align*}
\text{User} & \rightarrow \text{HSM}: \{[\text{key}]_{\text{UKEK2} \oplus \text{oldType} \oplus \text{newType}}, [\text{UKEK2} \oplus \text{oldType} \oplus \text{newType}]_{\text{KM} \oplus \text{IMP}} \} \\
\text{HSM} & \rightarrow \text{User}: \{[\text{key}]_{\text{UKEK2} \oplus \text{newType}} \}
\end{align*}
\]

Once the import-type key part obtained in step 2 has the desired difference XOR-ed in, using the ‘Last’ Key Part Import command, it can be used to change the type of the exported key upon re-import.

As before, this attack allows the intruder to change the type of the PIN derivation key so that it can be used with the Encipher command to encrypt primary account numbers.

4.3 IBM Attack

The attack discovered by IBM engineers exploits the fact that the data control vector is actually a string of binary zeroes, and thus \( X \oplus \text{DATA} = X \). The attacker is able to obtain the clear value of an export-type key encryption key, and therefore easily decrypt any other keys exported under it, such as the PDK. The attack proceeds as follows:

1. ‘Last’ Key Part Import

\[
\begin{align*}
\text{User} & \rightarrow \text{HSM}: \{\text{DATA} \oplus \text{EXP}, [\text{UKEK1}]_{\text{KM} \oplus \text{IMP} \oplus \text{KP}}, \text{IMP} \} \\
\text{HSM} & \rightarrow \text{User}: \{[\text{UKEK1} \oplus \text{DATA} \oplus \text{EXP}]_{\text{KM} \oplus \text{IMP}} \}
\end{align*}
\]

The intruder first conjures a pair of import-type key encryption keys with the known difference of DATA \oplus EXP, in the same manner as for the first step of the Import / Export Loop Attack.

2. Key Import

\[
\begin{align*}
\text{User} & \rightarrow \text{HSM}: \{[\text{UKEK2} \oplus \text{UKEK1} \oplus \text{EXP}], [\text{UKEK1}]_{\text{KM} \oplus \text{IMP}} \} \\
\text{HSM} & \rightarrow \text{User}: \{[\text{UKEK2} \oplus \text{EXP}]_{\text{KM} \oplus \text{EXP}} \}
\end{align*}
\]

Two forms of an export-type key encryption key are conjured, using the Key Import command, before the other UKEK1 is used to convert the type of the external key to DATA upon import.

3. Key Export

\[
\begin{align*}
\text{User} & \rightarrow \text{HSM}: \{[\text{UKEK2} \oplus \text{KM} \oplus \text{DATA}], [\text{UKEK2}]_{\text{KM} \oplus \text{EXP}} \} \\
\text{HSM} & \rightarrow \text{User}: \{[\text{UKEK2} \oplus \text{UKEK2} \oplus \text{DATA}] \}
\end{align*}
\]

The next step is to export the data-type UKEK2 under itself. Since the data control vector is a string of binary zeroes, this is equivalent to having UKEK2 encrypted under itself. The intruder has UKEK2 as a data key, so is able to decrypt \([\text{UKEK2}]_{\text{UKEK2}}\) and obtain the value of the key.
4. **Key Export**

\[ \text{User} \rightarrow \text{HSM}: \{ \text{key}\}_{\text{KM} \oplus \text{type}}^{|} \text{UKEK2}\}_{\text{EXP} \oplus \text{type}}^{|} \]

As the intruder has UKEK2 as an export-type key encryption key, he can export the desired key and decrypt the result himself in order to obtain the clear value of the exported key.

This attack is more serious than the previous two as the intruder could learn the unencrypted value of the PIN derivation key, thereby allowing him to calculate PIN numbers without requiring access to the security module.

### 4.4 Modelling the Attacks

The ‘first’ `Key_Part_Import` command was disabled in all models, since a trivial attack exists if the intruder has access to it in addition to the ‘last’ `Key_Part_Import` command. Together, these two commands allow the intruder to create a known export type key encryption key, export the desired key (e.g. the PIN derivation key) which can then be decrypted. The ‘last’ `Key_Part_Import` command cannot be disabled since all three attacks require it at some point.

Only one form of the `Key_Generate` command was included in the models — the one which returns a single working key of the desired type. The other versions of the command produce keys that are meant to be imported into another security module, or re-imported into the original one at a later date. As such, they do not result in the intruder gaining keys of any practical use, and therefore also just serve to increase the search space.

For the Import / Export Loop Attack, and the IBM Attack, three variants were modelled, capturing whether both, the first, or neither of the initial key conjuring steps are assumed to already have been carried out. Traditionally, the intruder is provided with both sets of conjured keys. In order to mimic the key conjuring process, extra roles were added that represented a successful conjuring attempt that returned the conjured key as well as the command output. For example, the transition for the role corresponding to the `Key_Import` command being used to conjure a key was as follows:

```
conjureUnknownKeyUsingKeyImport.
Rcv(TYPE'.{KEK1'}_xor(km,imp_CV)) /
  in(TYPE',KeyTypes)
=|>
  K1' := new() /
  Snd({K1'}_xor(KEK1',TYPE').{K1'}_xor(km,TYPE'))
```

Lastly, when modelling the IBM attack, we initially hard-wired the data control vector to zero, but this uncovered a minor bug in CL-AtSe which prevented the attack from being found. As a workaround, an additional role was used with the following two transitions:

```
t1.Rcv({xor(X',data_CV})_Y') =|> Snd({X'}_Y')
t2.Rcv({X'}_xor(Y',data_CV)) =|> Snd({X'}_Y')
```

Unfortunately, because the knowledge is expressed as a role, the attacker is limited in the number of times that he can use it, since the search is bounded to a set number of role instances. This does not stop the attack being found however, as it is only required once, although it may prevent other attacks being found under different circumstances.

### 4.5 Results

The system used to obtain the following results had a 3.6GHz Intel Xeon processor with 3.5Gb RAM, running Fedora Core 3. Two sets of options were used with CL-AtSe — the first to make it do a breadth-first search (to look for the shortest attack), and the second to make it carry out a depth-first search.

Unfortunately, because of the way in which CL-AtSe enforces the search bound, the results for attacks requiring multiple uses of the same command suffered. Rather than the bound giving a limit to the number of commands allowed in the attack, it gives a limit on the number of sessions that are allowed. That is, if a command is required three times for a particular attack, then the bound must be at least three. However,
this means that the maximum length of attacks considered is the product of the bound and the number of roles in the session. This typically results in a much higher search bound than is necessary, as it is not strictly tied to the length of the attack being searched for.

Although some conclusions are given in this section, a more general discussion of what can be drawn from these results is given in §6.

4.5.1 Key Import Attack

As well as the described model, containing all of the key management commands described in §2.2, a version was used that is equivalent to Youn et al.’s model given in [13] and [14]. They only included the Encipher, Decipher, Key Import, Key Export, and ‘last’ Key Part Import commands along with the version of the Key Generate command that produces a single working key of the desired type. Table 1 shows the results of running CL-AtSe on both models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Search Strategy</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Youn et al.</td>
<td>Breadth-First</td>
<td>200</td>
<td>64</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>114</td>
<td>15</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>14</td>
<td>6</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Breadth-First</td>
<td>70</td>
<td>34</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>34</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>20</td>
<td>10</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>10</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 1: Results for the rediscovery of Bond’s Key Import Attack using CL-AtSe. The right-hand column in each pair refers to the results of running CL-AtSe with the initial protocol optimisation step disabled.

On a marginally slower system, Youn et al. showed that they were able to discover the attack using Otter in a range of times from 0.2s to 28s depending on various factors. As can be seen from the above results, CL-AtSe is able to find the attack quicker, even with the model that contains more API commands. The fact that the analysed states and the reachable states are the same for the full model is due to the ordering of the commands being optimal. While a different ordering does not change the number of reachable states, it does affect the number of states analysed, and also the overall run-time (although not enough to cause the above results to be an unfair comparison).

Additionally, the intruder model in CL-AtSe does not restrict the attacker to only applying the XOR operator to unencrypted terms, or prevent him from re-encrypting an already encrypted term. Both these limitations increase the size of the search space that has to be checked, and it is therefore believed that had similar restrictions been possible, the run-times would have been faster still.

4.5.2 Import / Export Loop Attack

A total of six versions of the variant where the intruder is provided with both sets of conjured keys were checked with CL-AtSe. The first two contained only the commands required for the attack, the second two included the Encipher, Decipher, Key Import, Key Export and Key Translate commands as well as the ‘add’ and ‘last’ Key Part Import commands. This pair is referred to as the ‘Standard Commands’ model in the table below. The third pair added the version of the Key Generate command which returns a single operational key. Within each pair, the second model provided the intruder with a data key, whereas the first did not. The reason behind these different versions of the model was to observe how the differences affected CL-AtSe’s performance. Table 2 shows the results of running CL-AtSe on all six versions of the models for the first variant of Bond’s Import / Export Loop Attack.

The results show that the addition of the data key to the intruder’s initial knowledge has a significant effect on the run time, due primarily to the fact that the intruder is thus able to encrypt any known term using the Encipher command, and obtain a new term. This can be seen by the jump in the number of reachable states.

However, the biggest jump is caused by the inclusion of the Key Generate command, since it allows the intruder to add additional working keys, of any type, to his knowledge and thus results in a far greater number of possible command calls being made available.
Overall, the run-time is most sensitive to the number of working keys in the intruder’s knowledge, so it is therefore of paramount importance that he is not able to obtain unnecessary keys. Note that ‘unnecessary’ means keys which do not cause the attacker to gain anything by having knowledge of them. For example, two completely unrelated keys of the same type are unlikely to both be of use to the intruder.\(^5\)

In [12], Steel used a model that included the Encipher, Decipher, Key Import, Key Export, ‘last’ Key Part Import and Key Translate commands, and only the PIN derivation key (i.e. no data key). He was able to find the attack using a modified version of daTac in 1.47s. CL-AtSe is slightly slower on an equivalent model, taking 7.23s using breadth-first search.

Unfortunately, the only other model in which the attack was found was the simplest version of the variant where just the first pair of conjured keys were provided. Additionally, the attack was only found using depth-first search with the initial simplification steps enabled, the results of which are shown in table 3.

4.5.3 IBM Attack

As noted earlier, a minor bug in CL-AtSe meant that the intruder is not able to provide zero (or even \(X \oplus X\)) as a parameter to any of the commands. While this does not prevent the attack from being found, it does make it harder to do so, since the data control vector could not be hard-wired to zero. Table 4 gives the results for the first variant of the attack, where the intruder is provided with both sets of conjured keys.

\(^5\)The obvious exception to this is when the two keys are part of separate related key sets.

### Table 2: Results for the rediscovery of the first variant of Bond’s Import / Export Loop Attack using CL-AtSe. The right-hand column in each pair refers to the results of running CL-AtSe with the initial protocol optimisation step disabled.

<table>
<thead>
<tr>
<th>Model</th>
<th>Search Strategy</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rqd. Cmds.</td>
<td>Breadth-First</td>
<td>88</td>
<td>49</td>
<td>3.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>35</td>
<td>5.52</td>
</tr>
<tr>
<td>+ Data Key</td>
<td></td>
<td>299</td>
<td>283</td>
<td>132.64</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>Breadth-First</td>
<td>805</td>
<td>348</td>
<td>&gt;3600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5857</td>
<td>4066</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>Breadth-First</td>
<td>161384</td>
<td>68565</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>+ Data Key</td>
<td></td>
<td>415673</td>
<td>201331</td>
<td>&gt;3600</td>
</tr>
</tbody>
</table>

Table 3: Results for the rediscovery of the second variant of Bond’s Import / Export Loop Attack using CL-AtSe. All other models required too much time or memory, and are not shown.

The reason for the other models running out of time or memory is almost certainly due to the large number of valid keys that the intruder can add to their knowledge. However, we believe that this is the first time that formal methods have been used to rediscover the attack, when the intruder has only been given one of the related key sets, and has to conjure the other one.

4.5.3 IBM Attack

As noted earlier, a minor bug in CL-AtSe meant that the intruder is not able to provide zero (or even \(X \oplus X\)) as a parameter to any of the commands. While this does not prevent the attack from being found, it does make it harder to do so, since the data control vector could not be hard-wired to zero. Table 4 gives the results for the first variant of the attack, where the intruder is provided with both sets of conjured keys.

\(^5\)The obvious exception to this is when the two keys are part of separate related key sets.
<table>
<thead>
<tr>
<th>Model</th>
<th>Search Strategy</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rqd. Cmds. 1</td>
<td>Breadth-First</td>
<td>454</td>
<td>355</td>
<td>805.61</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>454</td>
<td>355</td>
<td>798.78</td>
</tr>
<tr>
<td>Rqd. Cmds. 1</td>
<td>+ Data Key</td>
<td>-</td>
<td>-</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>MEM</td>
</tr>
<tr>
<td>Rqd. Cmds. 2</td>
<td>Breadth-First</td>
<td>293</td>
<td>220</td>
<td>37.51</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>293</td>
<td>220</td>
<td>37.61</td>
</tr>
<tr>
<td>Rqd. Cmds. 2</td>
<td>+ Data Key</td>
<td>293</td>
<td>225</td>
<td>68.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>293</td>
<td>225</td>
<td>68.29</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>Breadth-First</td>
<td>664</td>
<td>338</td>
<td>49.88</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>664</td>
<td>338</td>
<td>49.75</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>+ Data Key</td>
<td>1114</td>
<td>688</td>
<td>120.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1114</td>
<td>688</td>
<td>123.34</td>
</tr>
<tr>
<td>All Cmds.</td>
<td>Breadth-First</td>
<td>60358</td>
<td>19121</td>
<td>1192.56</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>60358</td>
<td>19121</td>
<td>943.04</td>
</tr>
<tr>
<td>All Cmds.</td>
<td>+ Data Key</td>
<td>-</td>
<td>-</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Table 4: Results for the rediscovery of the first variant of IBM’s Attack using CL-AtSe. The right-hand column in each pair refers to the results of running CL-AtSe with the initial protocol optimisation step disabled.

The reason for the two sets of required commands is that two versions of this attack exist. In the second case, the Key Translate command is used to directly obtain the conjured key encrypted under itself:

\[
\text{User} \rightarrow \text{HSM}: \{ \text{UKEK2}_\text{UKEK1} \oplus \text{DATA} \}, \{ \text{UKEK1}_\text{DATA} \oplus \text{EXP} \}, \{ \text{RM}_\text{IMP} \}, \{ \text{UKEK2}_\text{RM}_\text{EXP} \} \\
\text{HSM} \rightarrow \text{User}: \{ \text{UKEK2}_\text{UKEK2}_\text{DATA} \}
\]

This is not discussed in [5], and we are therefore led to believe that this version of the attack may not previously have been known.

The second version does not take as long to find because each required command is only used once, whereas in the first version, the Key Import command is used twice. This means that there is a lower bound on the search for the second version. It is also the attack found when the ‘standard’ and ‘all’ command sets are used.

Table 5 shows the results for the second variant of IBM’s attack, where just the first pair of conjured keys is provided, although only the simpler version of the attack that uses the Key Translate command could be discovered. However, we believe that this is the first time that this variant of the attack has been rediscovered using formal methods.

Sadly, CL-AtSe was unable to find the full attack, as given by the third variant of this model, which required both sets of related keys to be conjured. However, that attack requires a depth bound of two, and as was shown with the Import / Export Loop Attack, this can cause a severe blow-up in the resources required.

5 Verification of IBM Recommendations

In response to Bond’s discovery of the Key Import Attack, IBM released a set of three recommendations [10] designed to prevent it from being carried out, covering suggested command usage, the security module’s access control system, and general procedural safeguards. However, they are presented as informal guidelines, and it is not clear which ones are necessary or sufficient to prevent the attack. Furthermore, the recommendations had never been formally verified to ensure that the required secrecy properties held.
The Key Import Attack does not require collusion between multiple individuals, and as such, the recommendations only attempt to prevent single-party attacks. However, one of them may not actually prevent the attack, and could even potentially expose the system to a more serious security breach, where the clear value of a key can be obtained.

The analysis of the recommendations highlighted a number of issues that may undermine their effectiveness, as well as clarifying a few points regarding their deployment. These have been drawn together into a concrete set of recommendations which aim to remove any remaining uncertainty over what needs to be done in order to prevent the various identified attacks on the CCA API.

Note that all verification runs were done using breadth-first search.

### 5.1 Recommendation 1

IBM’s first recommendation is to use public key techniques to transfer the initial key encryption key (KEK). This approach ensures that the KEK is never present in the clear, and therefore cannot be modified. IBM state that transferring the KEK (or any key for that matter) using clear key parts is not recommended, and is only supported because it was a standard method in the past and there are customers who still prefer it.

Using the public key approach, the KEK to be transferred is encrypted under the public key of the security module that the KEK is being sent to. Once decrypted, all other keys are transferred as before, encrypted under this shared KEK. Although the paper describes two ways of providing two security modules with the same KEK — encrypting an existing KEK, and randomly generating a new one — only the latter is possible, because the suggested command for the former will not accept key encryption keys.

#### 5.1.1 Overview of KEK Transfer Process

Public keys used for encryption must first be registered with the security module — a two stage process designed to prevent a malicious individual from adding their own public key. The first step causes the security module to store a hash value for the key, and in the second step, this is checked against the value computed for the key being added.

The entire key transfer process, based on this recommendation, is as follows:

1. Use the `PKA.Key.Generate` command at the destination security module to obtain an RSA public-private key pair, retaining the private key within the module.
2. One individual uses the `PKA.Public.Key.Hash.Register` command to register a hash value for the public key at the source security module.
3. A second individual uses the `PKA_Public_Key_Register` command at the source security module to actually add the public key.

4. Use the `PKA_Symmetric_Key_Generate` command to create two versions of a random KEK at the source security module — one as an importer, encrypted under the previously registered public key, and one as an exporter, encrypted under the local master key.

5. Use the `PKA_Key_Import` command at the destination security module to import the randomly generated KEK.

6. Transfer all other keys using this common KEK.

It should be clear that no single individual should have access to both commands required to register a public key with the source security module. Such a situation would allow that person to register their own public key, then decrypt the randomly generated KEK, and thus obtain the clear value of any exportable key.

Additionally, the `PKA_Key_Generate` command allows for the private key to be returned in the clear. However, this is only possible if the administrator of the security module allows it, since such an ability is not enabled by default.

Using public key techniques may prevent an attacker from modifying an existing KEK, but it does not stop them from conjuring their own KEK with the `Key_Part_Import` commands. The ‘first’ and ‘last’, or ‘first’ and ‘complete’, versions of the command allow an attacker to introduce a known key into the security module. As part of the second recommendation, IBM state that there is no need for this command when public key methods are used to transfer the initial KEK, and it should therefore be disabled. However, the manual only suggests that a policy of dual control can be enforced, and does not highlight the dangers of allowing a single person to have access to both.

5.1.2 Checking the Recommendation

In order to check IBM’s first recommendation, a new role was added to the model that represented the `PKA_Symmetric_Key_Import` command, which has the following semantics (where `pk` is the public key of the security module):

```
PKA_SYMMETRIC_KEY_IMPORT
User → HSM: {key.type}pk
HSM → User: {key}KM⊕type
```

It takes an encrypted data block, containing the key to be imported and corresponding type information, and returns the key encrypted under the local master key. The data block is encrypted under the public key that corresponds to the security module’s private key.

In practice, the data block includes additional information such as a device-specific identifier that prevents the key from being re-imported into the exporting security module. However, only the key and type information were considered in the model, since the other items can be safely ignored for the purposes of the verification process.

It was also assumed that the public key for the destination security module had already been securely registered with the originating device, and that the intruder was responsible for loading the key encryption key.

5.1.3 A Potential Attack

Having modelled IBM’s first recommendation, CL-AtSe found a rather simple attack that allows the clear value of a key being transferred to be discovered. It involves adding a known export-type key encryption key into the security module, before using it to export the key being transferred. The attack relies on the intruder being able to create encrypted data blocks that will be accepted by the `PKA_Symmetric_Key_Import` command.
The precise format of the data block is given in the CCA manual, along with the steps required to encrypt and decrypt it. In order to encrypt the data block, the attacker must have access to the security module’s public key. The only other apparent restriction is that the unique identifier cannot be the same as the one of the security module for which the key is to be imported. Since the clear value of the arbitrary key is used, the intruder can ensure that the parity bits are valid, and therefore guarantee that it will be accepted by the security module.

The attack can be carried out in two ways, only one of which requires the attacker to have the key being transferred:

1. Create a data block for an export-type key encryption key, and encrypt it under the public key of the appropriate security module in the manner described in the CCA manual.
2. Use the \texttt{PKA\_Symmetric\_Key\_Import} command to import the known key into the security module.
3. Obtain the key encrypted under the known KEK in one of the following ways:
   a. use the \texttt{Key\_Translate} command to convert the key being transferred to encryption under the known export-type KEK
   b. use the \texttt{Key\_Export} command to export the target key from the security module.
4. Since the value of the export-type KEK is known, the key can be decrypted.

This is a very simple attack which CL-AtSe found in under a second, using breadth-first search, on a model containing all of the commands (except for those listed as disabled in \texttt{4.4}). We reported this vulnerability to IBM, who conceded that the attack was possible, and intend to change the documentation to reflect this. However, they argue the attack would have to be carried out by an insider, and that the vulnerability is intrinsic to public key schemes.

Our first thought to prevent the attack, was to stop the \texttt{PKA\_Symmetric\_Key\_Import} command from accepting encrypted key blocks for export-type KEKs. However, an attack still exists, as the intruder can introduce a known import-type KEK, then use the \texttt{Key\_Import} command to import a known export-type KEK. The attack then proceeds as above, from step 3.

This led us to consider preventing any user from having access to both the \texttt{PKA\_Symmetric\_Key\_Import} and \texttt{Key\_Import} commands. We created two models, each one allowing access to only one of these functions, and discovered no further attacks for the bounds checked. The results are shown in table 6.

<table>
<thead>
<tr>
<th>Enabled Command</th>
<th>Bound</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Key_Import}</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>20</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>27</td>
<td>3</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>34</td>
<td>4</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>41</td>
<td>5</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>48</td>
<td>6</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>55</td>
<td>7</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>62</td>
<td>8</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>69</td>
<td>9</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>76</td>
<td>10</td>
<td>0.08</td>
</tr>
<tr>
<td>\texttt{PKA_Symmetric_Key_Import}</td>
<td>1</td>
<td>31</td>
<td>5</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>456</td>
<td>90</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8751</td>
<td>1749</td>
<td>514.27</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Table 6: Results for the verification of IBM’s first recommendation, with our additional constraints, using CL-AtSe. The right-hand run-time column refers to the results of running CL-AtSe with the initial protocol optimisation step disabled.
5.1.4 Analysis of Results

The results for the model where the intruder just has access to the Key Import command show that the number of reachable states increases by one, in line with the bound. This is due to the fact that the intruder only has the knowledge to use that command and import an external key. Once imported, the key cannot be used with any other command and so the intruder is only able to keep repeating the import step. Since no further new terms can be added to the intruder’s knowledge, the verification takes a trivial amount of time.

Note however, that this argument may not apply when the intruder is able to use the available commands to conjure keys.

The results for the second model, where the intruder has access to the PKA_Symmetric_Key_Import command, shows that he is able to carry out a far greater number of command calls. Although the intruder can generate a large number of known keys, none of them are export-type KEKs. In order to change their type using Bond’s Key Import Attack, the intruder requires access to the Key Import command, but it is not available to him. Therefore, although he can continue to generate a large number of known keys, none of them can be used to effect any attack on the existing keys.

If the known keys were used for some other purpose, then the intruder’s knowledge of them may have some value, but we assume that only properly installed keys would be used in other instances. As above though, this argument may not apply when the intruder is able to use the available commands to conjure keys.

5.2 Recommendation 2

IBM’s second recommendation is to use the access control system to ensure that no single person is able to execute the commands required for the attack. These are the Key_Part_Import and Key_Import commands, which are used to modify the key encryption key, and to change the type of the key being imported, respectively.

Recall that security module users are assigned to a user profile which determines the set of commands that they are allowed to execute. Therefore, if these two commands are not enabled together in any profile, then the attack cannot be mounted. The Key_Part_Import command has four versions, allowing for finer grained usage restrictions, and as mentioned previously, the ‘first’ and ‘last’, or ‘first’ and ‘complete’, versions should not be enabled in the same role.

5.2.1 Example KEK Transfer Procedure

In their recommendations paper, IBM provide an example of how a key encryption key can be transferred securely using clear key parts. It details the roles and responsibilities of five people (A – E), although they concede that, in some environments, person A and person E may be the same individual. Assuming that only two key parts are used, the five people carry out the transfer as follows:

1. Person A generates the two clear key parts, using the Random_Number_Generate command, and the key verification pattern (KVP) for the complete key encryption key (KEK). The first key part is given to person B, the second to person C, and the verification pattern to persons C and E.
2. Person B enters the first key part into the destination security module.
3. Person C enters the final key part into the destination security module, and calculates the KVP to make sure that the key has not been modified. If the key is made up of more than two parts, then the intermediate key parts are added by people with equivalent privileges to Person C, although they will not have to check the KVP.
4. Person D exports the key to be transferred from the source security module, encrypted under KEK.
5. Person E checks the KVP for the KEK to ensure that it was loaded correctly into the destination security module, and then imports the key being transferred.

It is assumed that the key parts are loaded into both security modules in the same way. In Bond’s Key Import Attack, the malicious insider was effectively playing the part of persons C and E. Note also that
person A has access to all parts of the key encryption key. If this person is also responsible for transferring the keys, as IBM state may be the case, then they would potentially be able to obtain the clear value of those keys.

5.2.2 Checking the Recommendation

Of the five people who are involved in the recommended key transfer process, only person B, person C and person E are considered to have access to the destination security module. A model was created for each of these individuals, with the appropriate restrictions on which commands are available to them.

Person B does not have access to the ‘middle’ and ‘last’ versions of the Key Part Import command, or the Key Import command, person C cannot use the ‘first’ Key Part Import command, and person E is prevented from using any version of the Key Part Import command. Furthermore, person C does not have access to the key being transferred, although he was given the imported key.

Although these restrictions are weaker than those suggested in the paper — where only the necessary commands are available to each person — they all ensure that at least one of the three requirements for the attack are missing. That is, none of them give the attacker access to a Key Part Import command, the Key Import command and the key being transferred.

In addition, the model for person B used a slightly modified version of the ‘first’ Key Part Import command which not only returned the encrypted key part, but the completed key under the assumption that the remaining key parts were not tampered with. This was to reflect the possibility that the individual could return to the security module once all the key parts had been imported.

The models checked for the intruder being able to change the type of the PDK (the Key Import Attack), obtain the clear value of a secret key, and obtain the arbitrary PAN encrypted under the PDK. The results of the verification runs using the ‘standard’ command set are shown in table 7, with no attack being found up to the bounds checked.

<table>
<thead>
<tr>
<th>Model</th>
<th>Bound</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person B</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>2</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>19</td>
<td>3</td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24</td>
<td>4</td>
<td>3.75</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>29</td>
<td>5</td>
<td>31.56</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>34</td>
<td>6</td>
<td>333.02</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>Person C</td>
<td>1</td>
<td>29</td>
<td>4</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>113</td>
<td>18</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>413</td>
<td>68</td>
<td>58.22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>Person E</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>19</td>
<td>3</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24</td>
<td>4</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>29</td>
<td>5</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>34</td>
<td>6</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>39</td>
<td>7</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>44</td>
<td>8</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>49</td>
<td>9</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>54</td>
<td>10</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 7: Results for the verification of IBM’s second recommendation using CL-AtSe. The right-hand run-time column refers to the results of running CL-AtSe with the initial protocol optimisation step disabled.

With the addition of the Key Generate command, CL-AtSe was only able to verify that no attack existed.
for a bound of 1 for each person, with higher bounds requiring more memory than was available.

5.2.3 Analysis of Results

The results for person B show that the reachable states only increases by one each time, in line with the bound. On closer inspection of the commands available to the intruder, this is due to the fact that he only has the necessary terms in his knowledge to use the (modified) ‘first’ Key\_Part\_Import command. By using that command, he can obtain terms of the following form:

\[ kp \oplus \text{KEK} \oplus \text{KP} \oplus \text{IMP} \]

where the left-hand term is the intermediate encrypted key part returned by the command, and the right-hand term is the subsequently completed full key. Note that \( kp \) is the clear key part provided as input to the command, and \( \text{KP} \) is the first key part given to the intruder.

Neither of these formats correspond to input parameters of any of the available commands, are are therefore useless to the intruder. As a result, all he can do is keep generating terms of the above form, with varying values for \( kp \) and \( \text{type} \). Since the intruder cannot generate terms which can be used, he will never be able to mount an attack of any form.

Note however, that this argument may not apply when the intruder is able to use the available commands to conjure keys.

While the results for person C suggest that they are able to carry out a much greater range of command calls, they too are quite limited in what additional knowledge can be obtained. Initially, the intruder is able to call the ‘middle’ and ‘last’ Key\_Part\_Import commands, adding new terms of the following format to his knowledge:

\[ kp \oplus \text{KEK} \oplus \text{KP} \oplus \text{IMP} \]

where the terms are returned by the ‘middle’ and ‘last’ versions of the Key\_Part\_Import command respectively. Similarly to before, \( kp \) is the clear key part provided as input to the commands, and \( \text{KP} \) is the final key part given to the intruder.

Only the left-hand term can be used by the intruder, as he does not have the required additional knowledge to use the commands that would accept the right-hand term. The left-hand term can only be used with the two Key\_Part\_Import commands, and therefore does not provide the intruder with anything useful — he can only continue to create terms of the above form. The number of reachable states reflects how many different options the intruder has when calling the Key\_Part\_Import commands.

Again however, this argument may not apply when the intruder uses the available commands to conjure keys.

The results for person E are almost identical to those for person B, and for the same reason. He only has the required knowledge to use the Key\_Import command with the key being transferred, and thus obtain the imported key. The imported key cannot be used with any commands and therefore, the intruder can only carry out the same import step again. Since no further new terms can be added to the intruder’s knowledge, the verification takes a trivial amount of time.

As with the previous two cases though, this argument may not apply when the intruder uses the available commands to conjure keys.

Models where the intruder was able to use the available commands to conjure keys were checked, but CL-AtSe was only able to verify that no attack existed for person B and a bound of 1. All other runs required more memory than was available, due to the large number of keys that the intruder could generate.

5.3 Recommendation 3

IBM’s third recommendation is to ensure that no single person has the opportunity to carry out the steps necessary to the attack, by tightly controlling the environment in which keys are entered. The most obvious
course of action which IBM recommend is to distribute each key part to two separate individuals — one of whom enters the key part, and the other who verifies that the entered data is the same as his copy. Another option is to monitor and log each key entry operation, so that suspicious actions can be traced to the person responsible.

With respect to the commands provided by the CCA API, a key verification pattern can be distributed to an individual who is not responsible for entering the key parts. This person can then verify that the key has been correctly added. The goal here is to ensure that the export-type key encryption key used by the source security module is the same as the import-type key encryption key used by the destination security module. If they are identical, then no type change can take place when the key being transferred is imported.

The last point that IBM make in this recommendation is that the people responsible for entering the key parts are generally not systems programmers and would therefore be unable to make use of the modified key. Overall, IBM argue that, while the attack may be possible in theory, sufficient procedural restrictions will make it impossible in reality.

5.3.1 Checking the Recommendation

The only part of this recommendation which directly relates to API usage is the suggestion that the loading of the initial key encryption key should be verified before additional keys are transferred to the security module.

This was included implicitly in the model, by modifying the Key Import command so that it would only accept the correct key encryption key. This simulates the assumption that an incorrect key would be identified and deleted from the security module, before it could be used to import any of the keys being transferred.

The intruder was provided with the same initial knowledge and capabilities as required for the Key Import Attack, i.e. the final key part, the initial encrypted key parts, and the key being transferred. The results are shown in table 8.

<table>
<thead>
<tr>
<th>Bound</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46</td>
<td>8</td>
<td>0.03</td>
</tr>
<tr>
<td>2</td>
<td>687</td>
<td>136</td>
<td>5.13</td>
</tr>
<tr>
<td>3</td>
<td>13133</td>
<td>2625</td>
<td>2827.35</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Table 8: Results for the verification of IBM’s third recommendation using CL-AtSe. The right-hand runtime column refers to the results of running CL-AtSe with the initial protocol optimisation step disabled.

Note that the security of this recommendation relies on the assumption that the intruder is not able to generate the modified KEK, take a copy of it, then delete it and generate the correct KEK, before using the copy to import the key being transferred once the verification procedure has been carried out.

As outlined in the recommendation, proper procedural and environmental controls should prevent any one individual from gaining the opportunity to generate the necessary modified KEK, or at the very least detecting when such a event has occurred.

5.3.2 Analysis of Results

The commands initially available to the intruder are the same as for person C in recommendation two, although he is given the PIN derivation key before it is imported rather than being given it in an already imported state. However, while this means that the intruder can now use the Key Import command from the second step onwards (once he has created a complete import-type key encryption key), the model restricts him to only using the correct KEK.

As a result, the variety of import-type KEKs and KEK-parts which the intruder can generate are useless. Therefore, once the intruder has generated the correct key encryption key and imported the PDK, there is nothing more of use that he can do.
Of course, as with the equivalent analysis of the results from recommendation two, this argument may not apply when the intruder uses the available commands to conjure keys.

### 5.4 Overall Conclusions

We have shown that, in certain scenarios, each one of IBM’s recommendations may not prevent someone from modifying the value of the initial key encryption key being transferred to a security module. However, our experiments were primarily considered the aspects of the recommendations that related to the API, that is the suggested availability of commands and assumed knowledge of the individual. As IBM point out, tight procedural controls are able to prevent an attack. What we have shown is that these procedural controls are necessary, and it is not enough to rely solely on the separation of duty provided by the `Key Part Import` command.

The first recommendation — to use public key techniques — potentially opens the door to a more serious attack. Even still though, logging when users call the `PKA Symmetric Key Import` command would detect inappropriate use, thus acting as a deterrent to potential attackers.

The second recommendation — to make proper use of the access control system — gives the greatest protection against the attack, provided that the individuals involved in the key loading process are not provided with dangerous combinations of commands. Furthermore, the individual who is responsible for generating the key parts must not be involved in any other part of the process. Provided that these conditions are met, IBM’s suggested transfer procedure is sufficient to prevent an attack.

The third recommendation — ensuring that the attacker never has the opportunity to execute the attack — should work in theory, although simply using the `Key Test` command to verify the key encryption key may not be enough.

It should be made clear that these conclusions, and the results presented in this chapter, do not take into consideration the possibility of an attacker using brute-force key-breaking techniques such as parallel key search (see §3.3.1). However, such attacks require a prolonged period of exposure to the security module, and a significant number of repeated command calls. Therefore, it is likely that procedural controls such as logging of command usage would detect, and potentially prevent, these kind of attacks.

#### 5.4.1 Safety Precautions

The analysis of known attacks, and of IBM’s recommendations, have shed light on a number of important points concerning the CCA API and its use. While the majority have already been identified, they are significant enough to collectively present again here:

- The ‘first’ and ‘last’, or ‘first’ and ‘complete’, versions of the `Key Part Import` command should never be enabled in the same role. Together these commands allow for arbitrary known keys to be introduced into the security module. This restriction could quite easily be enforced by the CCA.
- The person responsible for importing working keys into a new security module should not also be responsible for adding any of the parts of the key encryption key used in the transfer. This can be enforced at the API level by ensuring that the `Key Import` command cannot be enabled in any role that has one of the `Key Part Import` command variants enabled.
- The person(s) responsible for generating the key parts should not be involved with any other parts of the key transfer process, except for possibly the verification of the key encryption key. This cannot be enforced by the CCA, since the key parts can be generated without a security module.
- For greater security, key parts should be added twice, by different people. The key should then only be used if both complete versions are the same. Once again, such a procedure could be enforced by the API. For example, each version of the key could be stamped with a unique identifier, and the ‘complete’ `Key Part Import` command, which would now require two incomplete keys, could use these to determine that they are actually distinct. Obviously, the unique identifier would have to be in an encrypted form.
The different versions of the \textit{Key\_Part\_Import} command should only be enabled for a short time when the initial key encryption key is being built up. The ‘last’ and ‘complete’ forms can be used to conjure key parts, which can then be modified using the ‘last’ or ‘add’ and ‘complete’ forms.

Public key techniques should only be used if individuals cannot create the data block used to import keys with the \textit{PKA\_Symmetric\_Key\_Import} command. Additionally, care should be taken to ensure that the correct public key is being used to transfer the key encryption key.

No person should have access to the two commands required to register a public key with the security module, as this would allow for arbitrary public keys to be added. As before, the CCA could enforce this.

Each role should only provide the minimal set of commands required to undertake the designated job, and individuals should not be assigned to more than one role.

Where appropriate, roles should also place a restriction on the number of times that the user can execute particular commands. For example, there is no legitimate reason for someone who is adding a key part to use the command more than once.

It should not be possible for a single person to create or modify roles, or enable any of the commands. This would prevent security module administrators from abusing their positions.

### 6 Discussion

One of the aims of this research was to investigate how the analysis of security APIs differs from the analysis of conventional security protocols, and to determine how existing model checking tools perform on real-world examples. This section presents a discussion of these topics, based on our experiences over the course of this work, and covers various points from across the analysis, modelling, and verification stages.

#### 6.1 Differences Between Security APIs and Security Protocols

In general, there are a number of basic differences between security APIs and security protocols that result in the analysis of the former being conceptually somewhat simpler:

- There are only two agents — the security module and the intruder.
- The intruder communicates directly with the security module.
- The intruder is only trying to obtain sensitive pieces of data (including clear values of cryptographic keys).
- The security module is typically stateless — it is simply a reactive agent that responds in a deterministic manner to commands from the user.

This last point, in combination with the fact that the intruder’s knowledge is monotonic, can be used to potentially reduce the number of terms that need to be considered at each step of the analysis. This is discussed in further detail in \cite{6}.

However, there are a couple of points which result in security APIs being considerable harder to verify:

- There is no enforced ordering on the interactions between the intruder and the security module.
- Security modules may make use of operators with non-trivial algebraic properties, such as XOR.

The non-determinism resulting from the first of these causes the search space to be far larger, and is the primary reason for security API analysis being so difficult. The latter point is also responsible for increasing the search space, although the effects can be tempered by efficient handling of the algebraic properties.
6.2 Modelling Security APIs

As was shown by the experiments to rediscover known attacks, the most important factor in minimising
the search space is to ensure that the intruder does not start with, and is not able to generate, unnecessary
terms. With respect to the modelling of the API, this amounts to ensuring that superfluous commands are
not included, or at the very least, are only added once the smaller command set has been verified.

This can be determined by looking at the format of the input and output parameters of the commands. Commands which do not output terms that can be used by the intruder as input to other commands, and cannot be combined to form such a term, only serve to give the intruder an excessive amount of useless
knowledge. This was one of the reasons why the Key Test command was not included in our models.

Furthermore, the output from commands which add arbitrary terms of a particular form (e.g. the
Key Generate command) need to be carefully considered. It may be the case that multiple versions of
essentially equivalent terms (e.g. many data keys with unknown, and unrelated, values) will be of no use to
the intruder. This is why our models did not include all variants of the Key Generate command.

In future though, as the tools used to carry out the verification process become better, and computing
resources become more powerful and possess greater amounts of memory, then the points outlined in this
section will be of lesser importance. After all, if it is possible to prove, beyond a shadow of a doubt, that no
attacks exist given the entire command set, then there is no need to identify which ones are can be safely
ignored.

However, these points give an insight into particular methods that future purpose-built verification tools
may employ in order to be able to check entire API command sets.

6.3 Model Checking Tools

One of the biggest disadvantages that CL-AtSe has over theorem provers, when applied to our models,
is caused by how it interprets the idea of a bound on the search. While theorem provers will typically
increment the number of commands that are applied in succession, CL-AtSe increments the number of
times that a command can be applied. This means that the upper bound on the length of the attack being
searched for will increase by the number of commands that have been modelled. For example, in the Import
/ Export Loop Attack (which has seven steps), a couple of the commands are required twice, so because
there are eight commands in the model, CL-AtSe will consider certain paths with up to sixteen commands.

This issue would be avoided by having each command as a separate transition within one big role, but
that causes CL-AtSe to spend a massive amount of time trying to determine optimisations based on all
possible interleavings of the various commands. Note that this is distinct from the initial simplification
step, which can be disabled.

The fact that the intruder’s knowledge is monotonic means that, at any given point, the API commands
simply define how additional terms can be obtained. At the each step of the search, it will be possible to
carry out the exact same command call(s) as in the previous step, and possibly some new ones. It should be
clear that, in order for the search to be efficient, only the new command calls should be considered. That
is, for breadth-first search, command calls that do not involve at least one new term can be ignored (as they
will already have been considered), and for depth-first search, only commands that add new terms should
be candidates (otherwise they will not progress the search).

From the perspective of unification, this means that, in the breadth-first case, the part of the unifier
which corresponds to the input parameters of the command must involve at least one term obtained in the
previous step. Similarly, in the depth-first case, the part of the unifier which corresponds to the return data
from the command must contain at least one term obtained in the previous step.

Another issue that needs to be considered is the in-built support for less abstract features, such as key parity,
and how it affects other operators. For example, if the parity was modelled by two functions on the key, say
even () and odd (), then the exclusive-or operator would have to contain the following additional rules:

6The intruder’s ability to generate new terms is discussed in §6.4.
\[
even(X) \oplus \even(Y) = \even(X \oplus Y) \\
\even(X) \oplus \odd(Y) = \odd(X \oplus Y) \\
\odd(X) \oplus \odd(Y) = \even(X \oplus Y)
\]

One of the other back-ends in the AVISPA tool set, OFMC [17], accepts user-defined theories, such as the one given above, allowing for the kind of flexibility usually afforded to theorem provers. Sadly though, we were unable to use OFMC in our experiments, as our models exposed a problem with the way the unification process was implemented.

### 6.4 Intruder Capabilities

As well as being able to use commands to obtain additional knowledge, the intruder can use his own abilities to do so, free from any interaction with the device providing the security API. The intruder’s own ability to generate new terms is generally responsible for the huge size of the search space, as the number of terms that he can add at any given step is typically infinite. The reason for this is that the intruder can take any two known terms and encrypt each one under itself, and under the other. This can then be repeated indefinitely. Adding in an operator like exclusive-or simply exaggerates this problem further, especially when the cancellation properties are not taken into account.

However, Cortier and Steel have recently shown that only a very small subset of all these possible terms need to be considered [6]. This subset can be generated by restricting the abilities of the intruder to only taking the exclusive-or of unencrypted terms, and not using encrypted terms in new encryptions. That is, only add \(X \oplus Y\) and \(\{\text{unknown}\}_Y\) if both \(X\) and \(Y\) are unencrypted terms. This restriction results in the set of new terms that the intruder can generate at any given step being finite.

Currently, with CL-AtSe at least, the intruder is able to provide arbitrary new terms of the different types provided. However, he is not able to provide arbitrary new encrypted terms, where certain values are unknown. For example, given the \texttt{Key Import} command, the intruder cannot provide \(\{\text{unknown}\}_{\text{KEK}} \oplus \text{type}\) as the key to be imported.

Allowing the intruder to do this would mean that he could conjure input parameters for potentially any command. Of course, such an ability would have to be tempered in order to prevent the intruder conjuring the precise value of a key. Using the \texttt{Key Import} command as an example again, the intruder should not be able to conjure the import-type key encryption key, since in reality there is a \(1 \in 2^{112}\) chance of getting it right.

A consequence of this is that there would be no need to explicitly add variations of the commands which capture the conjuring process. Although such an ability would increase the search space by some factor, it would increase the accuracy of the model.

### 7 Conclusions and Future Work

We have demonstrated the first application of a model checker (CL-AtSe) to the problem of security API analysis, and showed that it is able to rediscover all known attacks on IBM’s Common Cryptographic Architecture API.

CL-AtSe was able to discover a variant of one of the attacks which we believe had not previously been known. Part of the reason for this is that the \texttt{Key Translate} command, which was required for the attack, has not traditionally been modelled. This quite clearly shows the importance of developing tools that can handle as many commands as possible.

The second part of our experiments involved verifying the recommendations published, in response to Bond’s Key Import Attack, by IBM. They had never been formally verified before, and it was unknown whether or not they did indeed prevent the attack. We showed, that under certain assumptions, the attack was still possible, as was a more serious attack. By revising our assumptions, we were able to show that the recommendations were sufficient to prevent any known API attack. These revised assumptions formed...
the basis for our own set of recommendations that were aimed at the design, implementation, and use of the API itself.

Unfortunately, we were unable to verify the recommendations beyond a bound of 1, when the intruder was assumed to be able to use the available commands to conjure keys. This was in line with the overall perception that any command which added a large number of keys to the intruder’s knowledge caused the search space to be too large.

CL-AtSe was designed to verify properties of security protocols, and part of our work was concerned with determining how it coped with the more complex task of analysing security APIs. While the results achieved with CL-AtSe were generally better than with other methods, the experiments highlighted a number of issues that should be considered in the design of tools used specifically for the analysis of security APIs.

Overall, while our results show that progress is being made, it is clear that there is still more to be done before API command sets can be verifed in their entirety without having potentially restrictive assumptions placed upon the intruder.

7.1 Related Work

The application of formal methods to the analysis of security APIs has traditionally involved the use of theorem provers, since they provide much greater flexibility than model checkers. This flexibility has been necessary to enable the algebraic properties of operators such as exclusive-or to be handled efficiently, and to properly model the abilities of the intruder. Youn et al. have demonstrated some success using Otter [14], and Steel has also shown promising results with a modified version of daTAC [12].

The other line of research, as demonstrated in this paper, is the use of tools designed for security protocol analysis, as they already employ an intruder-based communication model. Herzog is currently pursuing this approach, although at the time of writing, only a brief mention of his intentions is available [8]. A fair amount of recent work has focused on exactly how security APIs differ from security protocols in terms of the analysis required, with Bond and Clulow having looked into the discovery of lower level attacks [3].

Restricting the intruder’s knowledge, while still ensuring completeness, is also an active area of research, with Cortier and Steel having recently proven the validity of one of the more common ways of achieving this [6].

7.2 Future Directions

Based on our experiences while undertaking the research presented in this paper, we foresee research continuing in the following directions:

- The development of purpose-built verification tools for the analysis of security APIs.
- Investigation into ways of representing less abstract features of security APIs in order to look for other classes of attacks, or to rely less on the ‘perfect encryption’ model.
- Investigation into ways, either automatic or manual, to reduce the search space by means of identifying which terms are of no use to the intruder, and can be safely ignored.
- The development of verification methods that are able to prove that no attack exists for an unbounded number of command calls, ideally with a formal proof provided as output.
- The development of more realistic intruder models that incorporate abilities such as key conjuring through command misuse.
References


A Command Notation

The notation used to represent the semantics of the API commands in this thesis is based on the standard *Alice-Bob* notation. It is used to represent the exchange of messages between two agents, traditionally called Alice and Bob:

\[ A \rightarrow B : M \quad A \rightarrow B : M_1, M_2 \quad A \rightarrow B : M_1, M_2, \ldots, M_n \]

where the concatenation of two messages is represented by a period, and multiple messages sent at the same time are separated by commas. Encrypted messages are represented as follows:

\[ \{M\}_{key} \]

where *key* is a public or symmetric key. Finally, the exclusive-or of two terms is represented by:

\[ T_1 \oplus T_2 \]

For example, the *Encipher* command accepts a message from the user, along with a symmetric key that is encrypted under the exclusive-or of the security module’s master key and the data type control vector:

\[ \text{User} \rightarrow \text{HSM} : x, \{key\}_{\text{KM}\oplus\text{DATA}} \]

\[ \text{HSM} \rightarrow \text{User} : \{x\}_{key} \]

The security module then returns the message encrypted under the symmetric key.

B Full List of CCA Key Management Commands

In table 9, we list all the external commands in the CCA symmetric key management set. These are taken from the ‘Verb List’ in the CCA Manual [4, Appendix F], sections 1 (DES Key Processing and Key Storage verbs) and 2 (Data Confidentiality and Data Integrity Verbs). Note that we do not deal with public key management commands; the only exception being the *PKA_Symmetric_Key Import* command used in the model of IBM’s first recommendation. The purpose of this table is to justify our models, by showing which commands have been excluded and why. We have four reasons for excluding commands:

1. They would not normally be enabled, because doing so would compromise security. The command *Clear_Key_Import* is an example of this.

2. They generate new, unknown keys whose values are not related in any way to existing keys. Although we had versions of the models which included the simplest variant of the *Key_Generate* command, where appropriate, we preferred to provide the intruder an unknown key of each type in his initial knowledge. As a result, generating new unknown keys became redundant because the API is specified by a Horn-clause system with no disequality tests, and therefore any attack using two different unknown keys of the same type will still be an attack if the same key is used.

3. They do not generate any key material, for example commands used to generate key test values and MACs.

4. Their functionality is subsumed by that of another command that is included in the model

Commands excluded for reasons 2 and 3 may be used in non Dolev-Yao style attacks, such as parallel key search attacks [1, §3.1], but these are outside the scope of the analysis in this paper.
<table>
<thead>
<tr>
<th>Command</th>
<th>Included</th>
<th>Excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Disabled</td>
<td>Generation</td>
</tr>
<tr>
<td>Clear_Key_Import</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Control_Vector_Generate</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cryptographic_Variable_Encipher</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data_Key_Export</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data_Key_Import</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diversified_Key_Generate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Export</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Key_Generate</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Key_Part_Import</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Key_Storage_Initialisation</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DES_Key_Record_Create</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DES_Key_Record_Delete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DES_Key_Record_List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DES_Key_Record_Read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Record_Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Token_Build</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Token_Change</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Token_Parse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key_Translate</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Multiple_Clear_Key_Import</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random_Number_Generate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prohibit_Export</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decipher</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Digital_Signature_Generate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digital_Signature_Verify</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encipher</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_Generate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_Verify</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDC_Generate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One_Way_Hash</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Symmetric Key Management Commands
C  HLPSL Model of IBM’s CCA API

This appendix contains the HLPSL representation of the commands that were used across the all of the experiments. The modified versions of the ‘first’ KeyPartImport and KeyImport commands used in the verification of IBM’s second and third recommendations are not given. The roles are split into three sections covering the API commands, those used to conjure keys, and any additional roles used. Furthermore, the ‘session’ and ‘environment’ roles are given for these commands along with the intruder’s default knowledge.

Command Roles

role encipher(HSM : agent, Snd, Rcv : channel(dy))
played_by HSM
def=

local
  X : message, % Arbitrary data
  K1 : symmetric_key % An arbitrary data key

transition
  encipher.
  Rcv(X'.{K1'}_xor(km,dataCV))
  => Snd({X'}_K1')
end role

role decipher(HSM : agent, Snd, Rcv : channel(dy))
played_by HSM
def=

local
  X : message, % Arbitrary data
  K1 : symmetric_key % An arbitrary data key

transition
  decipher.
  Rcv({X'}_K1'.{K1'}_xor(km,dataCV))
  => Snd(X')
end role

role keyImport(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM
def=

local
  K1 : symmetric_key, % An arbitrary key
  KEK1 : symmetric_key, % An arbitrary key encrypting key
  TYPE : nat % An arbitrary key type control vector

transition
  keyImport.
  Rcv({K1'}_xor(KEK1',TYPE').TYPE'.(KEK1')_xor(km,impCV))
  /\ in(TYPE',KeyTypes)
  => Snd({K1'}_xor(km,TYPE'))
end role
role keyExport(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM
def=

local
  K1 : symmetric_key, % An arbitrary key
  KEK1 : symmetric_key, % An arbitrary key encrypting key
  TYPE : nat % An arbitrary key type control vector

transition

keyExport.
  Rcv({K1'}_xor(km,TYPE').TYPE'.{KEK1'}_xor(km,imp_CV))
  \ in(TYPE',KeyTypes)
  => Snd({K1'}_xor(KEK1',TYPE'))

end role

role keyPartImport1(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM
def=

local
  KP1 : symmetric_key, % An arbitrary key part
  TYPE : nat % An arbitrary key type control vector

transition

keyPartImport1.
  Rcv(KP1'.TYPE')
  \ in(TYPE',KeyTypes)
  => Snd({KP1'}_xor(km,kpart_CV,TYPE'))

end role

role keyPartImport2(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM
def=

local
  KP_old : symmetric_key, % An arbitrary old key part
  KP_new : symmetric_key, % An arbitrary new key part
  TYPE : nat % An arbitrary key type control vector

transition

keyPartImport2.
  Rcv(KP_new'.{KP_old'}_xor(km,kpart_CV,TYPE').TYPE')
  \ in(TYPE',KeyTypes)
  => Snd({xor(KP_new',KP_old')}_xor(km,kpart_CV,TYPE'))

end role
role keyPartImport3(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set) played_by HSM def=

  local
  KP_old : symmetric_key, % An arbitrary old key part
  KP_new : symmetric_key, % An arbitrary new key part
  TYPE : nat % An arbitrary key type control vector

  transition
  keyPartImport3.
  Rcv(KP_new'.{KP_old'}_xor(km,TYPE',kpart_CV).TYPE') /
  in(TYPE',KeyTypes)
  => Snd({xor(KP_old',KP_new')}_xor(km,TYPE'))

end role

role keyTranslate(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set) played_by HSM def=

  local
  K1 : symmetric_key, % An arbitrary key
  KEK1,KEK2 : symmetric_key, % Arbitrary key encrypting keys
  TYPE : nat % An arbitrary key type control vector

  transition
  keyTranslate.
  Rcv({K1'}_xor(KEK1',TYPE').TYPE'.{KEK1'}_xor(km,imp_CV).
  {KEK2'}_xor(km,exp_CV)) /
  in(TYPE',KeyTypes)
  => Snd({K1'}_xor(KEK2',TYPE'))

end role

role keyGenerateOP(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set) played_by HSM def=

  local
  K1 : symmetric_key, % An arbitrary key
  TYPE : nat % An arbitrary key type control vector

  transition
  keyGenerateOP.
  Rcv(TYPE') /
  in(TYPE',KeyTypes)
  => K1':=new() /
  secret(K1',key_val,{HSM}) /
  Snd({K1'}_xor(km,TYPE'))

end role
Key Conjuring Roles

role conjureKeyPart(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM
def=

  local
  KEY : symmetric_key, % Completed key that parts are conjured from
  TYPE : nat % An arbitrary key type control vector

  transition

  conjureKeyPart.
    Rcv(TYPE') /* in(TYPE',KeyTypes)
    -> KEY' := new() /\ secret(KEY',key_val,{HSM})
    \ Snd({KEY'}_xor(km,TYPE',kpart_CV)._xor(km,TYPE'))

end role

role conjureUnknownKey(HSM : agent, Snd, Rcv : channel(dy), KeyTypes : nat set)
played_by HSM
def=

  local
  KEK : symmetric_key, % An arbitrary key encrypting key, of type imp_CV
  TYPE : message, % An arbitrary key type control vector
  KEY_C : symmetric_key % The (unknown) conjured key

  transition

  conjureUnknownKey.
    Rcv(TYPE'.{KEK'}_xor(km,imp_CV))
    /* in(TYPE',KeyTypes)
    -> KEY_C' := new() /\ secret(KEY_C',key_val,{HSM})
    \ Snd({KEY_C'}_xor(KEK',TYPE')._xor(km,TYPE'))

end role

Additional Roles

role dataCVisZero(HSM : agent, Snd, Rcv : channel(dy))
played_by HSM
def=

  local
  X,Y : message % Arbitrary terms

  transition

  dataCVisZero_1.
    Rcv({xor(X',data_CV)}_Y')
    -> Snd({X'}_Y')

  dataCVisZero_2.
    Rcv({X'}_xor(Y',data_CV))
    -> Snd({X'}_Y')

end role
Session Role

role session(HSM : agent, KeyTypes : nat set) def=

  local
    SndHSM, RcvHSM : channel(dy), % Communication channels for the HSM
    PDK : symmetric_key % Unencrypted PIN derivation key

  init
    % hlpsl2if doesn’t like external constants being used to encrypt
    PDK := pdk

    % The intruder should not be able to obtain the PDK in the clear
    % secret(pdk, key_val, {HSM})

    % The intruder should not be able to obtain the arbitrary
    % primary account number encrypted under the PIN derivation key
    % secret({pan}_PDK, the_PIN, {HSM})

  composition
    encipher(HSM, SndHSM, RcvHSM)
    \ keyImport(HSM, SndHSM, RcvHSM, KeyTypes)
    \ keyExport(HSM, SndHSM, RcvHSM, KeyTypes)
    \ keyPartImport1(HSM, SndHSM, RcvHSM, KeyTypes)
    \ keyPartImport2(HSM, SndHSM, RcvHSM, KeyTypes)
    \ keyPartImport3(HSM, SndHSM, RcvHSM, KeyTypes)
    \ keyTranslate(HSM, SndHSM, RcvHSM, KeyTypes)
    \ keyGenerateOP(HSM, SndHSM, RcvHSM, KeyTypes)
    \ conjureKeyPart(HSM, SndHSM, RcvHSM, KeyTypes)
    \ conjureUnknownKey(HSM, SndHSM, RcvHSM, KeyTypes)
    \ dataCVisZero(HSM, SndHSM, RcvHSM)

end role
Environment Role

role environment() def=

    local
        KeyTypes : nat set % Set of key types
    
    const
        theHSM, i : agent,
        % Public control vectors
        data_CV : nat,
        imp_CV : nat,
        exp_CV : nat,
        kpart_CV : nat,
        pin_CV : nat,
        % HSM’s secret keys
        km : symmetric_key,
        kek : symmetric_key,
        pdk : symmetric_key,
        data : symmetric_key,
        % Arbitrary PAN
        pan : nat,
        % Things which should remain secret
        key_val : protocol_id,
        the_PIN : protocol_id
    
    init
        KeyTypes := {data_CV, imp_CV, exp_CV, pin_CV}
        intruder_knowledge = {% Control vectors
            data_CV, imp_CV, exp_CV, pin_CV, kpart_CV,
            % Working keys
            {pdk}_xor(km,pin_CV),
            {kek}_xor(km,imp_CV),
            {data}_xor(km,data_CV),
            % An arbitrary primary account number
            pan
        }

    composition
        session(theHSM, KeyTypes)

end role

goal
    secrecy_of key_val
    secrecy_of the_PIN
end goal

environment()