Analysing IBM’s Common Cryptographic Architecture API with a Protocol Analysis Tool

GAVIN KEIGHREN,1 GRAHAM STEEL and ALAN BUNDY
School of Informatics, University of Edinburgh

We present results from the application of CL-AtSe, a protocol analysis tool, to the problem of analysing the security of IBM’s Common Cryptographic Architecture (CCA) API. IBM’s CCA API is used by a number of security modules in Automated Teller Machine networks, and has been the focus of previous formal analysis attempts.

We show how a key-management API can be modelled in a way that facilitates efficient analysis by a security protocol analysis tool, and that CL-AtSe is capable of rediscovering all known attacks on the API, using models containing a larger subset of the API commands than in previous work.

We also analyse the set of recommendations released by IBM, in response to one of the known attacks, and show that under certain assumptions they may give rise to a different attack. We use CL-AtSe to refine a revised set of assumptions, under which the attack is prevented, and use these assumptions to determine a number of our own recommendations primarily aimed at the design and implementation of the API.

Finally, we discuss our findings and experiences with respect to how they could affect the design of future tools used to analyse security APIs.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification—Formal Methods; Model Checking; D.4.6 [Operating Systems]: Security and Protection—Access Controls; Cryptographic Controls

General Terms: Security, Verification

Additional Key Words and Phrases: Key Management, Protocol Analysis, Security APIs

1. INTRODUCTION

Security APIs are Application Programming Interfaces provided by devices called hardware security modules (HSMs) which are designed to carry out a series of cryptographic operations involving sensitive data. The goal of a security API is to ensure that any sensitive data does not appear in an unencrypted form outside of the HSM itself, and that the data is manipulated in a precisely controlled manner.

An attack on a security API is defined as a series of legal command calls which result in sensitive information being released. A number of attacks on IBM’s CCA API were discovered by Bond, and are presented together in [Bond 2001, §5]. The

Authors’ address: School of Informatics, The University of Edinburgh, Edinburgh, EH8 9LE.
E-mail addresses: g.keighren@sms.ed.ac.uk, graham.steel@ed.ac.uk, a.bundy@ed.ac.uk.
1 Supported by an EPSRC-funded Collaborative Training Account from The University of Edinburgh (EPSRC grant number EP/C537645/1).
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2007 ACM 0000-0000/2007/0000-0001 $5.00

attacks exploit the way in which the exclusive-or (XOR) function is utilised by the API, and allow a malicious insider to obtain the PIN for any account number. As part of his research, 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. As shown by Steel [2005], efficient handling of XOR is critical for analysing security APIs that make use of the function. Only very recently have verification tools been developed which have efficient in-built support for XOR. One such tool is CL-AtSe, a security protocol analyser developed as one of the back-ends in the AVISPA Tool Set. Security protocol analysis is a similar yet typically more constrained problem, and in this paper we investigate how CL-AtSe copes when applied to IBM's CCA API. Our aim is to determine the kinds of techniques, methods and features which a verification tool designed for the analysis of security APIs would require.

1.1 Contributions

The contributions of this paper include: a method for modelling security APIs as protocols so that they can be efficiently analysed by automatic tools; the first application of a protocol analysis tool capable of handling the algebraic properties of XOR to an industrial API; the first formal analysis of IBM's recommendations for preventing Bond's attacks, including a new attack on the revised CCA API discovered by CL-AtSe, and an improved set of recommendations which prevent it; and, an evaluation of the tool's performance, highlighting the theoretical and practical problems which must be addressed in order for formal security API analysis to continue to progress.

1.2 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. [2005] have demonstrated some success using Otter, and Steel [2005] has also shown promising results with a modified version of daTac. Courant and Monin [2006] used the proof assistant Coq to verify the security of Bond's proposed fixes to IBM's CCA API.

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 for stateful APIs, although at the time of writing, only a brief mention of his intentions is available [Herzog 2006]. 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 [2005] having looked into the automatic discovery of attacks involving parallel key searching, information leakage, etc.

Restricting the intruder's knowledge, while still ensuring completeness, is also an active area of research, with Cortier and Steel [2006] having recently proven the validity of one of the more common ways of achieving this.
1.3 Paper Outline

In §2, we present an overview of IBM’s CCA API, highlighting the important features and commands, and in §3, we describe how we modelled the API. §4 describes the known attacks, the additions made to our API model in each case, and the results obtained. Our verification of IBM’s three recommendations are presented in §5, including individual analysis of the results and a list of our own recommendations based on the results. 1 §6 is a discussion of our experiences, and contains a number of points that we feel should be considered when tackling problems of this nature. Lastly, §7 presents our conclusions, related work and possible directions for future work.

2. OVERVIEW OF IBM’S COMMON CRYPTOGRAPHIC ARCHITECTURE API

IBM’s CCA API [IBM 2006] is provided by their 4758 Cryptographic Coprocessor — an HSM used in a large number of Automated Teller Machines (ATMs) across the world, and in the mainframes of many banks. The main task of the HSM, in this setting, is to carry out PIN verification requests, although many of the commands facilitate the transfer of secret cryptographic keys in order to initialise a new device.

At its heart, the CCA is a key management system, which provides commands that use encrypted keys to achieve desired functions, e.g. PIN processing. 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.

Additionally, because 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, or to specific individuals (see §2.4).

2.1 A Note About PINs

There are a number of different ways in which the Personal Identification Number (PIN) is calculated for an account. One of the more common methods was developed by IBM in the 1970s, and uses a DES key, known as the PIN derivation key, to initially encrypt the primary account number (PAN). The result is then converted into a four digit number by truncating it, and using a decimalisation table to map the hexadecimal digits to binary digits. Figure 1 (pg. 4) gives an example of this process.

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 — thereby violating one of the security goals of the API.

2.2 Working Keys

The CCA API uses four main types for classifying 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. Each 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.* Used to encipher and decipher arbitrary data, as well as for the generation and verification of message authentication codes (MACs). Subtypes restrict exactly which of these functions a particular key can be used for.

*PIN Keys.* Used for PIN block encryption, PIN block decryption, PIN generation, and 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 above four operations (although PIN generation keys can also be used to verify PINs).

*Key Encryption Keys.* Used to encrypt and decrypt other working keys during transfer between security modules, and divided into import and export sub-types. Keys encrypted in this manner are referred to as external keys, as they must be imported into a security module before they can be used.

*Key Generation Keys.* Used with certain API commands to generate DES keys. Such commands typically use the provided key to encrypt or decrypt a supplied piece of data. This type prevents keys used for this purpose from being used with other commands (e.g. **Encipher** and **Decipher**) in order to prevent the value of generated keys being discovered.

The typing mechanism restricts the working keys which can be used with a particular command. For example, a PIN generation key cannot be used with the **Encipher** command to encrypt arbitrary data.

### 2.3 API Commands

In this section, we present the core key management commands, along with 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 of [Keighren 2006]. The commands are represented in Alice-Bob notation, with Table I describing the terms used.

The steps that the security module carries out for each command are 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 passed as a plaintext or decipherable parameter. 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 returned by the security module.

**Fig. 1.** An example of how PINs are calculated using the IBM method, from [Bond 2004].
Table I. Notation used to represent the API commands.

<table>
<thead>
<tr>
<th>KM</th>
<th>The security module’s master key</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>Control vector for data keys</td>
</tr>
<tr>
<td>IMP</td>
<td>Control vector for import-type key encryption keys</td>
</tr>
<tr>
<td>EXP</td>
<td>Control vector for export-type key encryption keys</td>
</tr>
<tr>
<td>KP</td>
<td>Control vector indicating that a key is only a key part, and not a complete key</td>
</tr>
</tbody>
</table>

| kek    | An arbitrary key encryption key            |
| key    | An arbitrary cryptographic key             |
| new    | An unknown, new cryptographic key          |
| type   | An arbitrary key type control vector       |
| kpX    | A key part (used to build an arbitrary key) |
| x      | Arbitrary (unencrypted) data               |

| $\langle x \rangle_y$ | $x$ encrypted under $y$ (using symmetric encryption) |
| $x \oplus Y$         | The bit-wise exclusive-or of $X$ and $Y$ |

**Encipher**

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

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**

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

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

**Key Import**

\[ \text{User} \rightarrow \text{HSM} : \langle \text{key} \rangle_{\text{kek} \oplus \text{type}}, \langle \text{type} \rangle, \langle \text{kek} \rangle_{KM \oplus \text{IMP}} \]
\[ \text{HSM} \rightarrow \text{User} : \langle \text{key} \rangle_{KM \oplus \text{type}} \]

Converts a key (of the given type) from encryption under the supplied import-type key encryption key to encryption under the local master key.

**Key Export**

\[ \text{User} \rightarrow \text{HSM} : \langle \text{key} \rangle_{KM \oplus \text{type}}, \langle \text{type} \rangle, \langle \text{kek} \rangle_{KM \oplus \text{EXP}} \]
\[ \text{HSM} \rightarrow \text{User} : \langle \text{key} \rangle_{\text{kek} \oplus \text{type}} \]

Converts a working key (which must have export permissions) from being encrypted under the local master key, to being encrypted under the supplied export-type key encryption key.

**Key Translate**

\[ \text{User} \rightarrow \text{HSM} : \langle \text{key} \rangle_{\text{kek1} \oplus \text{type}}, \langle \text{type} \rangle, \langle \text{kek1} \rangle_{KM \oplus \text{IMP}}, \langle \text{kek2} \rangle_{KM \oplus \text{EXP}} \]
\[ \text{HSM} \rightarrow \text{User} : \langle \text{key} \rangle_{\text{kek2} \oplus \text{type}} \]

Translates a key from encryption under an import-type key encryption key to encryption under an export-type key encryption key.
Key Part Import

User → HSM : $kp1$, type
HSM → User : $\{kp1\}_HSM^{type\oplus KP}$

User → HSM : $kpNew$, $\{kpOld\}_HSM^{type\oplus KP}$, type
HSM → User : $\{kpOld\oplus kpNew\}_HSM^{type\oplus KP}$

User → HSM : $kpNew$, $\{kpOld\}_HSM^{type\oplus KP}$, type
HSM → User : $\{kpOld\oplus kpNew\}_HSM^{type\oplus KP}$

User → HSM : $\{key\}_HSM^{type\oplus KP}$, type
HSM → User : $\{key\}_HSM^{type\oplus KP}$

‘First’ version

‘Add/Middle’ version

‘Last’ version

‘Complete’ version

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 more than one individual to obtain completed working keys. The other combination 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 Generate

User → HSM : $type1 [type2] \{kek1\}_HSM^{IMP} \{kek2\}_HSM^{EXP}$
HSM → User : $\{new\}_HSM \{kek1 | kek2\}_HSM^{type1} \{new\}_HSM \{kek1 | kek2\}_HSM^{type2}$

The Key Generate command has nine variants, which return 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.

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

2.4 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

---

3A user can either be a human individual or a computer process.
module is assigned to be a member of one of the specified roles, and once logged in, inherits the privileges defined therein.

The security module has 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 administrator, 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 an incomplete key, but can turn it into a completed working key.

It is also likely that there will be other roles which allow existing working keys to be exported or modified, and that allow new keys to be generated.

3. MODELLING THE API

Steel [2005] has recently shown that the algebraic properties of XOR must be handled efficiently, in order for the verification of security APIs which use the function to be tractable. To this end, we selected the model checker CL-AtSe, a back-end in the AVISPA tool set specifically designed to be used for the analysis and verification of security protocols and which supports XOR using the Baader-Schulz unification algorithm [Baader and Schulz 1992]. It accepts models written in the High Level Protocol Specification Language (HLPSL) [AVISPA Project 2003], which was also developed as part of the Avispa project. HLPSL has been designed to model 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 who is attempting to obtain secret information, possibly by posing as an honest agent. In HLPSL, a protocol is modelled as a set of roles which can be played by any of the honest agents, and 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.

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 effect-
ively 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 authentic-
ation 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.

3.2 CL-AtSe

CL-AtSe (Constraint Logic based Attack Searcher) has in-built support for a num-
ber of algebraic operators, including XOR. 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. Protocols are verified over a bounded number
of sessions, i.e. the bound determines the maximum number of times that each role
can be run. An overview of CL-AtSe is given in [Turuani 2006].

Unfortunately, while 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 due to there being fewer restrictions on term
unification. Note that while this increases the risk of false positives being returned,
it does not result in false negatives.

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
security APIs 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 second property is that there is no
enforced ordering on the execution of the commands.

Initially, the API commands were modelled as separate transitions within a single
role. However, CL-AtSe first explores each role to the full depth in order to de-
termine any simplifications that can be made, and the non-deterministic nature of
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

Fig. 2. The Key Import command modelled in HLPSL.

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 one transition. Figure 2 shows how the Key Import command was modelled. Recall that the Key Import command takes as input the key to be imported (encrypted under a key encryption key), the type of the key, 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.

Since 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, so is therefore able to make use of the commands. All of the commands are available at any given time, so were therefore composed in parallel within the session role.

Full and partial keys 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.

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.

A full listing of all the roles used in our experiments is given in [Keighren 2006, Appendix B].

4. REDISCOVERY OF KNOWN ATTACKS

There are three significant attacks which have been found on IBM’s CCA API: Bond’s Key Import Attack\(^4\) [Bond 2001, §5.1], his Import/Export Loop Attack [Bond 2001, §5.2], and IBM’s attack which exploits the fact that the data control vector is actually zero (presented in [Chulow 2003, §2.7.2]).

All attacks are presented here 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

\(^4\) Bond terms this attack the “Chosen Key Difference Attack on Control Vectors.”
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.

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

User → HSM : \(KP3 \oplus oldType \oplus newType, \{KP1 \oplus KP2\}^{KM \oplus IMP \oplus KP}^{KM \oplus IMP} \)

HSM → User : \(\{KP1 \oplus KP2 \oplus KP3 \oplus oldType \oplus newType\}^{RM \oplus IMP}^{RM \oplus IMP} \)

The intruder, who is responsible for adding the final key part, KP3, XORs a known difference into 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

User → HSM : \(\{\text{key}\}^{KEK \oplus oldType} \oplus newType, \{\text{KEK} \oplus oldType \oplus newType\}^{KM \oplus IMP}^{KM \oplus IMP} \)

HSM → User : \(\{\text{key}\}^{KM \oplus newType}^{KM \oplus newType} \)

Next, he imports the key being transferred, using the modified KEK (where \(KEK = KP1 \oplus KP2 \oplus KP3\) from above), and claiming that it has the desired type.

(3) Encipher

User → HSM : PAN , \(\{PDK\}^{KM \oplus DATA}^{KM \oplus DATA} \)

HSM → User : \(\{PAN\}^{PDK}^{PDK} \)

If the key being transferred is a PIN derivation key (PDK), then this attack can be used to change it to a data key thereby allowing it to be used to encipher arbitrary data. Recall that a primary account number (PAN) encrypted under the PDK gives the PIN for that account (see §2.1).

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, newType. Due to the cancellation properties of XOR, this results in \(KEK \oplus oldType\), which can then be used to correctly decrypt the key being imported. This key is then output as a working key of the new type.

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

User → HSM : IMP ⊕ EXP , \( \{ \text{UKEK1} \}_{\text{KM} \oplus \text{IMP}} \), IMP
HSM → User : \( \{ \text{UKEK1} \oplus \text{IMP} \oplus \text{EXP} \}_{\text{KM} \oplus \text{IMP}} \)

By providing a random value as the existing key part, the intruder can use the ‘last’ Key Part Import command to conjure\(^5\) 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.

(2) Key Import

User → HSM : \( \{ \text{UKEK2} \}_{\text{UKEK1} \oplus \text{IMP} \oplus \text{KP}} \), IMP ⊕ KP , \( \{ \text{UKEK1} \}_{\text{KM} \oplus \text{IMP}} \)
HSM → User : \( \{ \text{UKEK2} \}_{\text{KM} \oplus \text{IMP} \oplus \text{KP}} \)

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 with the other key encryption key from step 1, to obtain UKEK2 as an export-type key part. Note however, that this step is impossible in practice, as the Key Import command will not accept key parts. 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 [Youn 2004, §4.1]).

(3) Key Export

User → HSM : \( \{ \text{key} \}_{\text{KM} \oplus \text{type}} \), \( \{ \text{UKEK2} \}_{\text{KM} \oplus \text{EXP}} \)
HSM → User : \( \{ \text{key} \}_{\text{UKEK2} \oplus \text{type}} \)

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.

(4) Key Import

User → HSM : \( \{ \text{key} \}_{\text{UKEK2} \oplus \text{oldType}} \), \( \text{newType} \) , \( \{ \text{UKEK2} \oplus \text{oldType} \oplus \text{newType} \}_{\text{KM} \oplus \text{IMP}} \)
HSM → User : \( \{ \text{key} \}_{\text{KM} \oplus \text{newType}} \)

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 to encrypt primary account numbers.

\(^5\)Key conjuring is where an API command is called repeatedly with a random value in place of an encrypted key. When the decrypted key has the correct parity and is accepted by the security module, the random value corresponds to a valid, yet unknown, encrypted key. With key management commands, it is typically the case that the output will be a key which has some known relationship to the conjured key. See [Bond 2004, §7.2.3] for more details.

4.3 IBM Attack

The attack discovered by IBM engineers exploits the fact that the data control vector is actually zero, and thus $X \oplus DATA = X$. 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**
   
   User $\rightarrow$ HSM : DATA$\oplus$EXP, {\$UKEK1$\}_KM$\oplus$IMP$\oplus$EXP, IMP
   
   HSM $\rightarrow$ User : {\$UKEK1$\}_KM$\oplus$IMP

   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**
   
   User $\rightarrow$ HSM : {\$UKEK2$\}_UKEK1$\oplus$EXP, EXP, {\$UKEK1$\}_KM$\oplus$IMP
   
   HSM $\rightarrow$ User : {\$UKEK2$\}_KM$\oplus$EXP

   The **Key Import** command is then used to conjure two forms of an export-type key encryption key, before the other UKEK1 is used to convert the type of the external key to DATA upon import.

3. **Key Export**
   
   User $\rightarrow$ HSM : {\$UKEK2$\}_KM$\oplus$DATA, DATA, {\$UKEK2$\}_KM$\oplus$EXP
   
   HSM $\rightarrow$ User : {\$UKEK2$\}_UKEK2$\oplus$DATA

   The next step is to export the data-type UKEK2 under itself. Since the data control vector is zero, this is equivalent to having UKEK2 encrypted under itself. The intruder has UKEK2 as a data key, so is able to decrypt {\$UKEK2$\}_UKEK2 and obtain the clear value of the key.

4. **Key Export**
   
   User $\rightarrow$ HSM : {\$key$\}_KM$\oplus$type, type, {\$UKEK2$\}_KM$\oplus$EXP
   
   HSM $\rightarrow$ User : {\$key$\}_UKEK2$\oplus$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

This section describes how the API was modelled for the three attacks, and includes additional roles which were added in specific instances. We show how this was done without tailoring the models to make particular attacks easier to discover.
The ‘first’ Key Part Import command was disabled in all models, since all three attacks require the ‘last’ Key Part Import command and the CCA access control mechanism prevents these two commands from being active in the same role. This is because, together, they allow the intruder to create a known export-type key encryption key, and export and then decrypt any working key.

Only the form of the Key Generate command which returns a single working key of the desired type was included in the models. 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 provide the intruder with keys of any practical use, and just increase the search space.

For the Import/Export Loop Attack, and the IBM Attack, three variants were modelled, covering whether both, the first, or neither of the key conjuring steps had already been carried out. Traditionally, the intruder has been provided with both sets of conjured keys. To simulate the key conjuring process, extra roles were added to represent successful conjuring attempts and which 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:

\[
\text{conjureUnknownKeyUsingKeyImport.} \\
\text{Rcv(TY}PE\cdot\{\text{KEK1'}\}_\text{xor(km,imp}_{CV}\}) \land \text{in(TY}PE\cdot,\text{KeyTypes)} \\
=|> \text{K1'} := \text{new()} \\
\land \text{Snd}\{(\text{K1'})_\text{xor(KEK1',TY}PE\cdot)\cdot\{\text{K1'}\}_\text{xor(km,TYPE')}\}
\]

Lastly, when modelling the IBM attack, we initially changed the model so that the data control vector was explicitly given as 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:

\[
t1.\text{Rcv}\{(\text{xor(X',data}_{CV})\cdot Y')} =|> \text{Snd}\{(X'),Y')
\]
\[
t2.\text{Rcv}\{(X')_\text{xor(Y',data}_{CV})\} =|> \text{Snd}\{(X'),Y')
\]

Unfortunately, as the knowledge is expressed as a role, the intruder is limited in the number of times that he can use it, since the search is bounded on the number of role instances. Although this does not stop the IBM attack being found, 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. The bound does not limit the number of overall steps in an attack, but rather the maximum number of times that any particular command can be used in it.\textsuperscript{6} This causes a significant jump in the size of the search space when the bound increases.

\textsuperscript{6}This means that the set of possible command interleavings to be checked is \(\frac{(b+c)!}{bc}\), where \(b\) is the bound, and \(c\) is the number of commands.
Table II. 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.

<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>Youn et al.</td>
<td>Breadth-First</td>
<td>200</td>
<td>31</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>114</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>Full</td>
<td>Breadth-First</td>
<td>70</td>
<td>70</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>20</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

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 given in §2.3, a version was used that is equivalent to Youn et al.’s model given in [Youn 2004] and [Youn et al. 2005]. 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 II shows the results of running CL-AtSe on both models.

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 results in Table II, 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. Such restrictions have been proven to be valid for IBM’s CCA API, and similar command sets [Cortier and Steel 2006]. 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 Trans- late commands as well as the ‘add’ and ‘last’ Key Part Import commands. This pair is referred to as the ‘Standard Commands’ model in Table III, and is termed as such because, with the exception of Key Translate, these are the commands which have traditionally been included in models of the API, [Steel 2005; Youn 2004; Youn et al. 2005]. 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, in order to gauge how it would scale up to larger
Table III. 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>Rqld. Cmds.</td>
<td>Breadth-First</td>
<td>88</td>
<td>88</td>
<td>3.40</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>70</td>
<td>70</td>
<td>5.52</td>
</tr>
<tr>
<td>Rqld. Cmds.</td>
<td>Breadth-First</td>
<td>460</td>
<td>460</td>
<td>13.67</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</td>
<td>299</td>
<td>299</td>
<td>132.64</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>Breadth-First</td>
<td>895</td>
<td>895</td>
<td>13.67</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>-</td>
<td>-</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>Breadth-First</td>
<td>5857</td>
<td>5857</td>
<td>176.43</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</td>
<td>-</td>
<td>-</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>All Cmds.</td>
<td>Breadth-First</td>
<td>161384</td>
<td>5753</td>
<td>700.16</td>
</tr>
<tr>
<td></td>
<td>Depth-First</td>
<td>-</td>
<td>-</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>All Cmds.</td>
<td>Breadth-First</td>
<td>415673</td>
<td>19142</td>
<td>2030.65</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</td>
<td>-</td>
<td>-</td>
<td>&gt;3600</td>
</tr>
</tbody>
</table>

command sets. Table III 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 a data key to the intruder’s initial knowledge has a significant effect on the run time, due 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 ‘unnecessary’ keys are kept to a minimum. That is, keys which are of no real use to the attacker. As noted in [Cortier et al. 2007], giving the intruder two unrelated keys of the same type does not allow him to do anything more than if he had just the one. The reasoning is based upon the fact that none of the API commands check for disequality between keys.

In [Steel 2005], Steel analysed 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 IV. The size of the search space caused the breadth-first strategy to run out of memory on all of the models, and the depth-first strategy to time-out on the other, more complex, models. 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.
Table IV. Results for the rediscovery of the second variant of Bond’s Import/Export Loop Attack using CL-AtSe with the initial simplification steps enabled.

<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>Depth-First</td>
<td>5637</td>
<td>-</td>
<td>2958</td>
</tr>
</tbody>
</table>

Table V. 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.

<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>454</td>
<td>355</td>
<td>805.61</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</td>
<td>454</td>
<td>355</td>
<td>806.82</td>
</tr>
<tr>
<td>Rqd. Cmds.</td>
<td>Breadth-First</td>
<td>293</td>
<td>220</td>
<td>37.51</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</td>
<td>293</td>
<td>220</td>
<td>37.30</td>
</tr>
<tr>
<td>Rqd. Cmds.</td>
<td>Breadth-First</td>
<td>293</td>
<td>225</td>
<td>68.00</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</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>+ Data Key</td>
<td>Depth-First</td>
<td>664</td>
<td>338</td>
<td>49.94</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>Breadth-First</td>
<td>1114</td>
<td>688</td>
<td>120.05</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</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>2252</td>
<td>1192.56</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>Depth-First</td>
<td>60358</td>
<td>2252</td>
<td>1184.53</td>
</tr>
</tbody>
</table>

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 explicitly given as zero. Table V gives the results for the first variant of the attack, where the intruder is provided with both sets of conjured keys.

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:

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

This is not discussed in [Clulow 2003], 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. This demonstrates quite clearly that the manner in which CL-AtSe handles the search bound is not ideally suited to the analysis of security APIs. The longer attack only requires one more command call, but an extra $n$ possible calls are considered (where $n$ is the total number of commands in the model).

For the ‘standard’ and ‘all’ command sets, it is the shorter attack which is found. Table VI shows the results for the second variant of IBM’s attack, where just...
Table VI. Results for the rediscovery of the second 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.

<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. 2</td>
<td>breadth-first</td>
<td>7111</td>
<td>582</td>
<td>3465</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>depth-first</td>
<td>551</td>
<td>63</td>
<td>181</td>
</tr>
<tr>
<td>Std. Cmds.</td>
<td>breadth-first</td>
<td>8081</td>
<td>582</td>
<td>3465</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>depth-first</td>
<td>613</td>
<td>63</td>
<td>207</td>
</tr>
<tr>
<td>All Cmds.</td>
<td>breadth-first</td>
<td>22712</td>
<td>1772</td>
<td>5386</td>
</tr>
<tr>
<td>+ Data Key</td>
<td>depth-first</td>
<td>7339</td>
<td>337</td>
<td>763</td>
</tr>
</tbody>
</table>

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.

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 search bound of two, and as was shown with the Import/Export Loop Attack, this can cause a severe blow-up in the resources required.

If the search bound was tied to the number of command calls in the attack, rather than the maximum number of times that each command could be called, then it is likely that the attack could have been found. Although the key conjuring roles cause a big increase in the search space, the full attack only requires one more command call than the second variant.

5. VERIFICATION OF IBM RECOMMENDATIONS

In response to Bond’s discovery of the Key Import Attack, IBM released a set of three recommendations [IBM 2001] 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 can be carried out by one person, therefore the set of recommendations only attempt to prevent attacks by lone individuals. Note that all verification runs used breadth-first search, since this strategy is faster when the entire search space has to be explored, provided sufficient memory is available.

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.
Using the public key approach, the KEK to be transferred is encrypted under the public key of the security module that it is being sent to. Once decrypted, all other keys are transferred as before, encrypted under this shared KEK. Although the IBM paper, [IBM 2001], 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.

5.1.2 Checking the Recommendation. In order to check IBM’s first recommendation, the **PKA Symmetric Key Import** command had to be modelled. It has the following semantics (where $pk$ is the public key of the security module):

\[
\text{PKA Symmetric Key Import} \\
\text{User} \to \text{HSM} : \{\text{key, type}\}_{pk} \\
\text{HSM} \to \text{User} : \{\text{key}\}_{\text{KM} \oplus \text{type}}
\]

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. Given our model of 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.

It should be noted that strict procedural controls, such as those suggested in IBM’s third recommendation, may prevent the intruder from having the opportunity to carry out this attack. However, such reasoning could be applied to any attack where the intruder has access to all the required commands, and this attack therefore serves to highlight the vulnerability at the API level.

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 (see step 3), 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 **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:
   - use the **Key Translate** command to convert the key being transferred to encryption under the known export-type KEK
   - use the **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 as described in §4.4 (except for the key conjuring versions). 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.
Table VII. 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.

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>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>MEM</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Initially, we thought that this attack would be prevented if encrypted key blocks for export-type KEKs were not accepted by the PKA Symmetric Key Import command. However, an attack still exists, as the intruder can introduce a known import-type KEK, then use the 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 Key Import and PKA Symmetric Key Import commands. We created two models — one with the Key Import command disabled, and one with the PKA Symmetric Key Import command disabled — and discovered no further attacks up to bounds of 4 and 10 respectively. The results are shown in Table VII.

5.1.4 Analysis of Results. The results for the model where the intruder can use 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 PKA Symmetric Key Import command is available to the intruder, 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 ‘last’ Key Part Import command and the Key Import command, 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 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.

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. 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 imported 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.
Table VIII. 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.

<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>&gt;3600</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>&gt;3600</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>

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

Although these restrictions are weaker than those suggested by IBM — 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 VIII, with no attack being found up to the bounds checked.

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 number of reachable states only increase 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:

\[ \{ kp \}_{KM \oplus type \oplus KP} \quad \{ kp \oplus KEK \oplus KP1 \}_{KM \oplus IMP} \]

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 KP1 is the first key part given to the intruder.

Neither of these formats correspond to input parameters of any of the available commands, and 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 type. Since the intruder cannot generate terms which can be used, he will never be able to mount an attack of any form. This kind of analysis should be amenable to automation, and we foresee that such techniques will be employed by future security API analysis tools (see §6.2).

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, respectively:

\[ \{ kp \oplus KEK \oplus KP3 \}_{KM \oplus IMP \oplus KP} \quad \{ kp \oplus KEK \oplus KP3 \}_{KM \oplus IMP} \]

Similarly to before, \( kp \) is the clear key part provided as input to the commands, and KP3 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 IX.

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 an event has
Table IX. Results for the verification of IBM’s third 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.

<table>
<thead>
<tr>
<th>Bound</th>
<th>States Analysed</th>
<th>Reachable States</th>
<th>Run-Time (s)</th>
<th>Run-Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46</td>
<td>8</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>2</td>
<td>687</td>
<td>136</td>
<td>5.13</td>
<td>5.17</td>
</tr>
<tr>
<td>3</td>
<td>13133</td>
<td>2625</td>
<td>2827.35</td>
<td>8407.05</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>MEM</td>
<td>MEM</td>
</tr>
</tbody>
</table>

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

Our experiments were primarily considered with 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 in certain circumstances these procedural controls are necessary, and it is not enough to rely solely on the separation of duty provided by the API.

The first recommendation — to use public key techniques — potentially allows a more serious attack. However, logging when the PKA Symmetric Key Import command is called 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. The first of these conditions is enforced by more recent versions of the API, and provided that the latter condition is also 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 section, do not take into consideration the possibility of an attacker using brute-force key-breaking techniques such as parallel key search (see [Bond 2004, §3.3.1]).

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 middle four (points three through six) are already known, they are significant enough to be collectively presented again here.

—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 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 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.

—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 an existing model checking tool performs on a real-world example. This section presents a discussion of these topics, based on our experiences over the course of our work, covering various points from 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 which should be considered when analysing the former:

— 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.
— There is no enforced ordering on the interactions between the intruder and the security module.

The non-determinism resulting from the last of these causes the search space to be far larger, and is the primary reason for security API analysis being so difficult.

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 a smaller command set has been verified.

This can be determined by looking at the format of the input and output parameters of the commands. Only commands which output terms that can be used as input to other commands, or that can be used by the intruder to generate such terms, need to be considered. 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. As noted in [Cortier et al. 2007], the intruder does not gain anything from having multiple versions of essentially equivalent terms (e.g. many data keys with unknown, and unrelated, values). This is why our models did not include all variants of the Key Generate command.

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

---

7The intruder’s ability to generate new terms is discussed in §6.4.

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 if each command was a separate transition within one big role, but that causes CL-AtSe to spend a huge amount of time trying to determine optimisations based on all possible interleavings of the various commands.

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. Therefore, at each step of the search, it will be possible to carry out the exact same command call(s) as in the previous step. Thus, for the search to be efficient, only the new command calls should be considered.

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:

\[
\text{even}(X) \oplus \text{even}(Y) = \text{even}(X \oplus Y) \\
\text{even}(X) \oplus \text{odd}(Y) = \text{odd}(X \oplus Y) \\
\text{odd}(X) \oplus \text{odd}(Y) = \text{even}(X \oplus Y)
\]

One of the other back-ends in the AVISPA tool set, OFMC, 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 its unification process was implemented, and it failed to find any of the attacks.

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 exaggerates this problem further.

However, Cortier and Steel [2006] have recently shown that only a very small subset of all these possible terms need to be considered — namely, singly-encrypted terms and the exclusive-or of unencrypted terms. That is, the terms \(X \oplus Y\) and \(|X|_Y\), where 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
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 Key Import command as an example again, the intruder should not be able to conjure the import-type key encryption key, since it is a double length DES key and thus there is a $1 \text{ in } 2^{112}$ 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 also increase the accuracy of the model.

7. CONCLUSIONS

We have demonstrated the application of a protocol analysis tool (CL-AtSe) to the problem of security API analysis, and shown 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 the IBM Attack which we believe had not previously been known. Part of the reason for this is that the Key Translate command, which was required for the attack, has not traditionally been modelled. This 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 by IBM in response to Bond’s Key Import Attack. 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 where the clear value of a key could be discovered. However, stringent procedural controls would prevent the attacks in all but one case, where additional separation of duty was necessary. By revising our assumptions, we were able to show that no attacks existed for the bounds checked. 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, with the models that simulated the intruder using the available commands to conjure keys, we were unable to verify the recommendations beyond a bound of 1. This was in line with the overall perception that commands which add a large number of keys to the intruder’s knowledge cause the search space to be too big.

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 previous methods, the experiments highlighted a number of issues that we feel need to be considered when analysing security APIs.

Overall, while our results show that progress is being made, it is clear that there is plenty to be done before API command sets can be verified in their entirety without enforcing potentially unsound restrictions upon the intruder or set of commands.
REFERENCES
CL-AtSe. Available from http://www.loria.fr/equipes/cassis/softwares/AtSe/.