A Formal Theory of Key Conjuring*

Véronique Cortier
LORIA, Projet Cassis
CNRS & INRIA
cortier@loria.fr

Stéphanie Delaune
LORIA, Projet Cassis
CNRS & INRIA
delaune@lsv.ens-cachan.fr

Graham Steel
School of Informatics
University of Edinburgh
graham.steel@ed.ac.uk

Abstract

Key conjuring *is the process by which an attacker obtains an unknown, encrypted key by repeatedly calling a cryptographic API function with random values in place of keys. We propose a formalism for detecting computationally feasible key conjuring operations, incorporated into a Dolev-Yao style model of the security API. We show that security in the presence of key conjuring operations is decidable for a particular class of APIs, which includes the key management API of IBM's Common Cryptographic Architecture (CCA).*

1 Introduction

Cryptographic security APIs are sets of functions designed to facilitate the secure generation, storage, use and destruction of cryptographic keys. Security APIs for tamper-resistant hardware devices typically manage keys by keeping a secret master key inside the device. This is used to encrypt all the working keys used for operational functions, so that they can be securely stored outside the device. One technique used by attackers attempting to breach security is to try calling API functions with random values in the place of encrypted keys, to see if they are allowed to pass, or whether the device signals an error. This process is known as *key conjuring* [1]. Learning the encrypted value of a key might not seem useful, but several attacks have been presented that leverage this trick in order to compromise the security of an API [1, 4, 3].

A promising approach to security API analysis involves adapting Dolev-Yao style protocol analysis techniques [10], where details of cryptographic algorithms used are abstracted away and a logical model is constructed, with rules describing the operations of the intruder and protocol. This can be adapted quite naturally to API analysis by considering the API to be a set of 2-party protocols, each describing an exchange between the secure hardware module and the host machine [12, 14, 8]. However, in previous work, the key conjuring trick was treated in an ad-hoc fashion, by adding a number of pre-chosen keys to the intruder's initial knowledge [12, 14, 8], or by adding a rule to allow particular keys to be conjured [11]. This raises doubts about completeness of the search for attacks, and hence the strength of any proofs of security.

The aim of the work in this paper is to address this problem by proposing a formal model that identifies all computationally feasible key conjuring operations, and allows these to be incorporated into a Dolev-Yao style model for security analysis of the API. We propose a transformation that automatically computes all the possible ways of performing key conjuring from the API rules. Our transformation takes as input a set of formal rules representing the behaviour of an API and outputs new formal rules representing key conjuring. In this manner, we eliminate the need for the user to generate key conjuring rules by hand. As far as we are aware, this paper presents the first formal treatment that allows an exhaustive set of key conjuring rules to be obtained.

Our second main contribution is to show decidability of the security of APIs (expressed as a reachability property), in the presence of key conjuring, for a class of APIs that includes for example the symmetric key management API of IBM's Common Cryptographic Architecture (CCA). In particular, it requires consideration of the algebraic properties of the Exclusive Or operation. Our decidability result holds for an unlimited number of sessions, though we do bound the number of times key conjuring operations are used. Indeed, it would not be realistic to allow the intruder to conjure as many keys as he wishes since it requires a significant amount of access to the API. This class is related to the class proposed in [8], with two main differences. First, we consider explicit decryption, since it was more appropriate for modelling key conjuring, and it reflects better the implementation. Second, we have to consider key conjuring

*This work has been partly supported by the RNTL project POSE, the ACI Jeunes Chercheurs JC9005, and EPSRC project *Automated Analysis of Security Critical Systems*, grant number GR/S98139/01.

rules which introduce fresh nonces. A precise comparison can be found in Section 6.5.

In the rest of the paper, we first explain the purpose and operation of security APIs, and define our formalism for describing them (Section 2). We then propose a transformation for key conjuring in Section 3. In Section 4, we explain the security problem we are interested in, and define a restricted class of APIs, arguing that these restrictions are quite natural. In Section 5 we show that certain classes of key conjuring operations are of no use to the intruder, and need not be considered in a formal model. We then show (Section 6) that security for our class of APIs is decidable in the presence of key conjuring operations. The class includes our motivating example, the key management API of the IBM 4758 Hardware Security Module, which was shown to be vulnerable to key conjuring attacks by Bond in [1]. We conclude, with a discussion of future work, in Section 7. Due to lack of space, some proofs are omitted and can be found in [7].

2 Background

In this section, we first explain what a security API is, before going on to define the concept more formally.

2.1 Security APIs

The purpose of a security application program interface (API) is to allow untrusted code to access sensitive resources in a secure way. Hardware security modules (HSMs), for example, have security APIs which control access to the cryptoprocessor and memory inside the module. This allows the API to manage access to cryptographic keys. HSMs are deployed in security critical environments such as the cash machine network, where they are used to protect customers PINs and other sensitive data. They typically consist of a cryptoprocessor and a small amount of memory inside a tamper-proof enclosure. They are designed so that should an intruder open the casing or insert probes, the memory will auto-erase in a matter of nanoseconds. In a typical ATM network application, all encryption, decryption and verification of PINs takes place inside the HSM. Many different cryptographic keys will be used for these operations. IBM's 4758 CCA[1] API [2] partitions keys into various types, such as data keys, PIN derivation keys, import keys and export keys. Each type has an associated public *control vector*. The HSM stores a *master key* in its tamper-proof memory. The keys the HSM uses for its various operations, called *working keys*, are stored outside the HSM encrypted under the master key XORed against the appropriate control vector for the key type. For example, a data key would be encrypted under $\mathsf{km} \oplus \mathsf{data}$.[2] Working keys can then only be used by sending them back into the HSM under an appropriate API command. Only particular types of keys will be accepted by the HSM for particular operations. For example, data keys can be used to encrypt arbitrary messages, but so-called *PIN Derivation Keys* (PDKs, with control vector pin) cannot. This is critical for security: a customer's PIN is just his account number encrypted under a PIN derivation key. In 2001, Bond discovered attacks in which the intruder uses API commands to change the type of a key, exploiting the algebraic properties of XOR [1]. The attack allows a PIN derivation key to be converted into a data key, which can then be used to encrypt data. Hence the attack allows the intruder to generate a PIN for any account number.

Formal work on the CCA first concentrated on rediscovering the attacks on the original version of the API [12, 14], and then on proving both Bond's proposed fixes [9], and the fixes IBM actually implemented [8], to be secure. However, these works made an informal approximation of the ability of the intruder to 'conjure' keys, a trick used several times in Bond's attacks. To explain precisely what key conjuring is, we first need to define some notation.

2.2 Definitions

We now define our (mostly standard) notation for reasoning about APIs, and then define the class of APIs considered in this paper.

Cryptographic primitives are represented by functional symbols. More specifically, we consider a *signature* Σ which consists of an infinite number of constants including a special constant 0 and three non constant symbols $\{\_\}_\_$ (encryption), dec (decryption) and $\oplus$ (XORing) of arity 2. We also assume an infinite set of variables $\mathcal{X}$. The set of *terms*, denoted by $\mathcal{T}(\Sigma, \mathcal{X})$, is defined inductively by

$$\begin{array}{lllll} T & ::= & & & \text{terms} \\ & & x & & \text{variable } x \\ & | & f(T_1, \ldots, T_n) & & \text{function application} \end{array}$$

where f ranges over the functions of Σ and n matches the arity of f. For instance, the term $\{m\}_k$ is intended to represent the message m encrypted with the key k (using symmetric encryption) whereas the term $m_1 \oplus m_2$ represents the message m_1 XORed with the message m_2. The constants may represent control vectors or keys for example.

We rely on a sort system for terms. Terms which respect this sort-system are said to be *well-typed*. It includes a set of base type Base and a set of ciphertext type Cipher. We have variables and constants of both types. Moreover we

[1]CCA stands for 'Common Cryptographic Architecture', while 4758 is the model number of the HSM. See `http://www-3.ibm.com/security/cryptocards/pcicc.shtml`

[2]$\oplus$ represents bitwise XOR.

assume that our function symbols have the following type:

$$\begin{array}{rcccccc} \oplus & : & \mathsf{Base} & \times & \mathsf{Base} & \rightarrow & \mathsf{Base} \\ \{\_\}_{\_} & : & \mathsf{Base} & \times & \mathsf{Base} & \rightarrow & \mathsf{Cipher} \\ \mathsf{dec} & : & \mathsf{Cipher} & \times & \mathsf{Base} & \rightarrow & \mathsf{Base} \end{array}$$

A pure term t is a well-typed term whose only encryption symbol (when such a symbol exists) is at its root position. We say that a term t is headed with f if its root symbol is f. The set of variables occurring in t is denoted $vars(t)$. We denote by $st(t)$ the set of subterms of t. This notation is extended as expected to set of terms. A term is *ground* if it has no variable. Substitutions are written $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with $dom(\sigma) = \{x_1, \ldots, x_n\}$. A substitution σ is *ground* if all of the t_i are ground. The application of a substitution σ to a term t is written $t\sigma$.

We equip the signature Σ with an equational theory $\mathsf{E_{API}}$ that models the algebraic properties of our operators:

$$\mathsf{E_{API}} := \left\{ \begin{array}{rclrcl} \{\mathsf{dec}(x,y)\}_y & = & x & x \oplus 0 & = & x \\ \mathsf{dec}(\{x\}_y, y) & = & x & x \oplus x & = & 0 \\ x \oplus (y \oplus z) & = & (x \oplus y) \oplus z & x \oplus y & = & y \oplus x \end{array} \right.$$

It defines an equivalence relation that is closed under substitutions of terms for variables and under application of contexts. In particular, we say that two terms t_1 and t_2 are equal, denoted by $t_1 =_{\mathsf{E_{API}}} t_2$ if they are equal modulo the equational theory $\mathsf{E_{API}}$. If two terms are equal using only the equations of the last line (resp. involving $\oplus$), we say that they are equal modulo Associativity and Commutativity (AC) (resp. modulo Xor).

In the CCA API, as in many others, symmetric keys are subject to parity checking. The 4758 uses the DES (and 3DES) algorithm for symmetric key encryption. A (single length) DES key consists of 64 bits in total, which is divided into eight groups, each consisting of seven key bits and one associated parity bit. For an odd parity key, each parity bit must be set so that the overall parity of its group is odd. For an even parity key, the parity bits must be set so that all groups are of even parity. If the groups have mixed parities, then the key is of undefined parity and considered invalid. The CCA API checks that all DES keys are of odd parity, and all control vectors are even, so that a key XORed against a control vector will give another odd parity key. These parity considerations are important for our analysis of key conjuring, and are represented in our formalism by occurrences of the predicate symbols chkEven and chkOdd, each having a term as argument. Intuitively, $\mathsf{chkOdd}(t)$ means that t has an odd parity. Among the constants in Σ, some have a parity. By default (no explicit parity given to a constant), we will assume that such a constant has no parity. Moreover, we have some rules to infer parity from known facts, which are:

$$\begin{array}{rcl} \mathsf{chkEven}(x_1),\ \mathsf{chkEven}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkOdd}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkEven}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkOdd}(x_1 \oplus x_2) \end{array}$$

Intruder capabilities and the protocol behaviour are described using *rules* as defined below.

Definition 1 (API rule) *An API rule is a rule of the form* $\mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k), x_1, \ldots, x_n \rightarrow t$*, where*

- $x_1, \ldots, x_n$ *are variables,*
- t *is a term such that* $vars(t) \subseteq \{x_1, \ldots, x_n\}$*,*
- $u_1, \ldots, u_k$ *are terms of* Base *type not headed with* $\oplus$*,*
- $\mathsf{chk}_i \in \{\mathsf{chkOdd}, \mathsf{chkEven}\}$*,* $1 \leq i \leq k$*.*

We also assume that the rule only involves pure terms.

The third condition might seem restrictive. However, it merely requires that we check each component of a sum rather than the entire sum. For example, if the sum $v_1 \oplus \cdots \oplus v_k$ has some expected parity, each v_i should also have some expected parity, and we ask that their parity is checked separately.

Example 1 *The intruder capabilities are represented by the following set of three API rules:*

$$\begin{array}{rcll} x,\ y & \rightarrow & \{x\}_y & \textit{encryption} \\ x,\ y & \rightarrow & \mathsf{dec}(x,y) & \textit{decryption} \\ x,\ y & \rightarrow & x \oplus y & \textit{xoring} \end{array}$$

Example 2 *Commands may include several parity checks. In Figure 1, we give the symmetric key management subset of the IBM 4758 API, written in our notation. The terms* km, imp, exp, kp, data *and* pin *denote constant of* Base *type whereas* xtype, xk1, … *denote variables. Note that all the rules satisfies conditions stated in Definition 1. For instance, Key Import is used to make a new working key for an HSM. The new key is sent to the target HSM encrypted under a transport key. The command decrypts the imported package, and returns the key encrypted under the local master key XOR the appropriate control vector.*

3 A Formal Theory of Key Conjuring

We first introduce key conjuring informally, giving an example of a key conjuring attack. This will help to explain our transformation. We then formally define our transformation that takes a set of API rules, and extends it with rules that permit key conjuring.

$$
\begin{array}{lrcl}
\textit{Key Part Imp. 1}: & \mathsf{xk1},\ \mathsf{xtype} & \rightarrow & \{\mathsf{xk1}\}_{\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype}} \\
 & \mathsf{chkOdd(xk1)},\ \mathsf{chkEven(xtype)} & & \\
\textit{Key Part Imp. 2}: & \mathsf{chkEven(xtype)},\ \mathsf{y},\ \mathsf{xk2},\ \mathsf{xtype} & \rightarrow & \{\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})\oplus\mathsf{xk2}\}_{\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype}} \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})) & & \\
 & \mathsf{chkEven(xk2)} & & \\
\textit{Key Part Imp. 3}: & \mathsf{chkEven(xtype)},\ \mathsf{y},\ \mathsf{xk3},\ \mathsf{xtype} & \rightarrow & \{\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})\oplus\mathsf{xk3}\}_{\mathsf{km}\oplus\mathsf{xtype}} \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})) & & \\
 & \mathsf{chkEven(xk3)} & & \\
\textit{Key Import}: & \mathsf{chkEven(xtype)},\ \mathsf{y},\ \mathsf{xtype},\ \mathsf{z} & \rightarrow & \{\mathsf{dec}(\mathsf{y}, \mathsf{dec}(\mathsf{z}, \mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype})\}_{\mathsf{km}\oplus\mathsf{xtype}} \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{z}, \mathsf{km}\oplus\mathsf{imp})) & & \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{dec}(\mathsf{z}, \mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype})) & & \\
\textit{Key Export}: & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{z}, \mathsf{km}\oplus\mathsf{exp})),\ \mathsf{y},\ \mathsf{xtype},\ \mathsf{z} & \rightarrow & \{\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{xtype})\}_{\mathsf{dec}(\mathsf{z},\mathsf{km}\oplus\mathsf{exp})\oplus\mathsf{xtype}} \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{xtype})) & & \\
 & \mathsf{chkEven(xtype)} & & \\
\textit{Encrypt Data}: & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{data})),\ \mathsf{x},\ \mathsf{y} & \rightarrow & \{\mathsf{x}\}_{\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{data})} \\
\textit{Decrypt Data}: & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{data})),\ \mathsf{x},\ \mathsf{y} & \rightarrow & \mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{data})) \\
\textit{Translate Key}: & \mathsf{chkEven(xtype)},\ \mathsf{x},\ \mathsf{xtype},\ \mathsf{y1},\ \mathsf{y2} & \rightarrow & \{\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y1}, \mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype})\}_{\mathsf{dec}(\mathsf{y2},\mathsf{km}\oplus\mathsf{exp})\oplus\mathsf{xtype}} \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y1}, \mathsf{km}\oplus\mathsf{imp})) & & \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y2}, \mathsf{km}\oplus\mathsf{exp})) & & \\
 & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y1}, \mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype})) & &
\end{array}
$$

Figure 1. IBM CCA Symmetric Key Management Transaction Set

3.1 Key Conjuring

As we have seen, key management APIs like the CCA keep working keys outside the HSM, safely encrypted, so that they can only be used by sending them back into the HSM under the terms of the API. What happens when an intruder wants to use a particular command in an attack, but does not have access to an appropriate key? For example, suppose he has no data keys (terms of the form $\{\mathsf{d1}\}_{\mathsf{km}\oplus\mathsf{data}}$), but wants to use the *Encipher* command. In an implicit decryption formalism, the command is defined like this

$$\mathsf{x}, \{\mathsf{xkey}\}_{\mathsf{km}\oplus\mathsf{data}} \rightarrow \{\mathsf{x}\}_{\mathsf{xkey}}$$

This suggests that the command cannot be used if the intruder does not have a data key. However, in reality, an intruder could just guess a 64 bit value and use that in place of the data key. The HSM will decrypt the guessed value under $\mathsf{km}\oplus\mathsf{data}$, and check the parity of the resulting 64 bit term to see if it is a valid key before, enciphering the data. Usually, the check will fail and the HSM will refuse to process the command, but if the intruder guesses randomly, he can expect that 1 in every 256 guessed values will result in a valid key. This notion is captured by our formalism, in which we write the *Encipher* command like this:

$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km}\oplus\mathsf{data})),\ \mathsf{x},\ \mathsf{y} \rightarrow \{\mathsf{x}\}_{\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{data})}$$

It may seem useless for the intruder to simply guess values, since the result is a term he knows enciphered under an unknown key, but used cleverly, this technique can result in serious attacks. For example, Bond's so called import-export attack [1], uses key conjuring to convert a PIN derivation key into an encryption key, allowing an intruder to generate the PIN for any given account number.

Description of Bond's attack. We give Bond's attack in Figure 2, written in our formalism, with explicit decryption and parity checking. We assume that the attacker initial knowledge contains $\{\mathsf{pdk}\}_{\mathsf{km}\oplus\mathsf{pin}}$ (a PIN key encrypted for transfer), the control vectors pin, data, imp, exp, kp, and the constant 0. Moreover, we model the fact that control vectors are of even parity and secret keys km and pdk are of odd parity by considering the corresponding facts (*e.g.* $\mathsf{chkEven(pin)}$). We will show how the PIN derivation key pdk can be converted into a data key, which then can be used to encrypt data. Hence the attack allows a criminal to generate a PIN for any account number. For this, we show that the attacker is able to derive $\{\mathsf{pdk}\}_{\mathsf{km}\oplus\mathsf{data}}$.

Step 1 is a key conjuring step. The attacker is using the *Key Part Import 3* command, using the control vector imp (for xtype) and the key part 0 (for $\mathsf{xk3}$) but without a term of the form $\{\mathsf{m}\}_{\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{imp}}$. Instead, he repeatedly tries random values until some value n_1 decrypts under $\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{imp}$ to give a valid key, *i.e.* a term of

Step	Input		Output
1. Key Part Imp. 3	$?,\ 0,\ \mathsf{imp}$ $\mathsf{chkEven}(0),\ \mathsf{chkEven}(\mathsf{imp})$	$\xrightarrow{\text{new } \mathsf{n}_1}$	$\{\mathsf{dec}(\mathsf{n}_1, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{imp})\}_{\mathsf{km} \oplus \mathsf{imp}},\ \mathsf{n}_1$ $\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_1, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{imp}))$
Let $\mathsf{r} = \mathsf{dec}(\mathsf{n}_1, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{imp})$			
2. Key Part Imp. 3	$\mathsf{n}_1,\ \mathsf{imp} \oplus \mathsf{exp},\ \mathsf{imp}$ $\mathsf{chkEven}(\mathsf{imp}),\ \mathsf{chkEven}(\mathsf{imp} \oplus \mathsf{exp})$ $\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_1, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{imp}))$	$\rightarrow$	$\{\mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{exp}\}_{\mathsf{km} \oplus \mathsf{imp}}$
3. Key Import	$?,\ \mathsf{imp} \oplus \mathsf{kp}, \{\mathsf{r}\}_{\mathsf{km} \oplus \mathsf{imp}}$ $\mathsf{chkOdd}(\mathsf{r}),\ \mathsf{chkEven}(\mathsf{imp} \oplus \mathsf{kp})$	$\xrightarrow{\text{new } \mathsf{n}_2}$	$\{\mathsf{dec}(\mathsf{n}_2, \mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{kp}))\}_{\mathsf{km} \oplus \mathsf{imp} \oplus \mathsf{kp}},\ \mathsf{n}_2$ $\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_2, \mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{kp}))$
Let $\mathsf{r}' = \mathsf{dec}(\mathsf{n}_2, \mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{kp})$			
4. Key Import	$\mathsf{n}_2,\ \mathsf{exp} \oplus \mathsf{kp},\ \{\mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{exp}\}_{\mathsf{km} \oplus \mathsf{imp}}$ $\mathsf{chkEven}(\mathsf{exp} \oplus \mathsf{kp}),\ \mathsf{chkOdd}(\mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{exp})$ $\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_2, \mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{kp}))$	$\rightarrow$	$\{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{exp} \oplus \mathsf{kp}}$
5. Key Part Imp. 3	$\{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{exp} \oplus \mathsf{kp}},\ 0,\ \mathsf{exp}$ $\mathsf{chkEven}(0),\ \mathsf{chkEven}(\mathsf{exp}),\ \mathsf{chkOdd}(\mathsf{r}')$	$\rightarrow$	$\{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{exp}}$
6. Key Part Imp. 3	$\{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{imp} \oplus \mathsf{kp}},\ \mathsf{pin} \oplus \mathsf{data},\ \mathsf{imp}$ $\mathsf{chkEven}(\mathsf{data} \oplus \mathsf{pin}), \mathsf{chkEven}(\mathsf{imp}), \mathsf{chkOdd}(\mathsf{r}')$	$\rightarrow$	$\{\mathsf{r}' \oplus \mathsf{data} \oplus \mathsf{pin}\}_{\mathsf{km} \oplus \mathsf{imp}}$
7. Key Export	$\{\mathsf{pdk}\}_{\mathsf{km} \oplus \mathsf{pin}},\ \mathsf{pin},\ \{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{exp}},$ $\mathsf{chkOdd}(\mathsf{pdk}), \mathsf{chkEven}(\mathsf{pin})$	$\rightarrow$	$\{\mathsf{pdk}\}_{\mathsf{r}' \oplus \mathsf{pin}}$
8. Key Import	$\{\mathsf{pdk}\}_{\mathsf{r}' \oplus \mathsf{imp}},\ \mathsf{data},\ \{\mathsf{r}' \oplus \mathsf{data} \oplus \mathsf{pin}\}_{\mathsf{km} \oplus \mathsf{imp}}$ $\mathsf{chkOdd}(\mathsf{pdk}),\ \mathsf{chkOdd}(\mathsf{r}' \oplus \mathsf{data} \oplus \mathsf{pin}),\ \mathsf{chkEven}(\mathsf{data})$	$\rightarrow$	$\{\mathsf{pdk}\}_{\mathsf{km} \oplus \mathsf{data}}$

"?" represents inputs that are replaced by random values by the attacker.

Figure 2. Bond's Import/Export Attack in our formalism

odd parity. Note that we have written this by labelling the arrow to show the conjuring of a new term n_1, and the odd parity check is now on the right hand side of the rule, indicating that the intruder has learnt the fact $\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_1, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{imp}))$. In the rest of the attack we write r in place of $\mathsf{dec}(\mathsf{n}_1, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{imp})$.

Having succeeded in finding a suitable value n_1, he uses the command again with $\mathsf{imp} \oplus \mathsf{exp}$ as the key part to be added to the key, in *Step 2*. This yields two unknown key encrypting keys, r and $\mathsf{r} \oplus \mathsf{imp} \oplus \mathsf{exp}$, with a known difference.

In *Step 3*, the intruder uses key conjuring again, this time with the *Key Import* command, using random values in place of $\{\mathsf{xkey}\}_{\mathsf{xkek} \oplus \mathsf{xtype}}$, and using $\{\mathsf{r}\}_{\mathsf{km} \oplus \mathsf{imp}}$ as the key encrypting key. Again, we write this as the generation of a new term n_2, and the intruder learns the fact $\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_2, \mathsf{r} \oplus \mathsf{imp}))$. In the rest of the attack we write r' in place of $\mathsf{dec}(\mathsf{n}_2, \mathsf{r} \oplus \mathsf{imp})$. In *Step 4*, the intruder uses the conjured value again to obtain an export version of the key.

The partial keys obtained by these two operations can then be completed using *Key Part Import 3*. The exporter is completed to give $\{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{exp}}$, in *Step 5*, whilst the importer is set to change the type of a key from pin to data, in *Step 6*. A PIN derivation key $\{\mathsf{pdk}\}_{\mathsf{km} \oplus \mathsf{pin}}$ can then be turned into a data key by first exporting it under $\{\mathsf{r}'\}_{\mathsf{km} \oplus \mathsf{exp}}$ using *Key Export* in *Step 7*, and then changing the type by re-importing it using $\{\mathsf{r}' \oplus \mathsf{data} \oplus \mathsf{pin}\}_{\mathsf{km} \oplus \mathsf{imp}}$ as the importer, in *Step 8*. Having obtained a PIN derivation key as a data key, the intruder can now encrypt account numbers to obtain customer PINs.

In 2003, as a result of work by Youn et. al [14], it came to light that this attack was impossible in practice, as an undocumented check in the CCA's implementation prevents key parts being passed to *Key Import*. This would mean steps 3 and 4 of the attack couldn't be executed. However, further attacks using key conjuring had been discovered by then, [3, 4], on both the CCA API and other APIs. Clulow notes in [4] that key conjuring can be prevented by using a hash or MAC to test the authenticity of keys, but many designs do not include such measures, which increase the key management overhead.

Our example attack shows the potential of key conjuring to mount attacks. It also demonstrates the features of our formalism which allow us to detect realistic key conjuring operations. A straightforward 'explicit decryption' model is not sufficient for a key conjuring analysis, since though this allows an attack like Bond's be discovered, it doesn't take into account parity checks. This means that the model cannot distinguish between feasible and non-feasible key conjuring steps, leading to false attacks. For example, for a command like *Key Import* (see Example 2), an explicit decryption model without parity checking would allow an intruder to conjure values for both y and z, which in practice is highly unlikely: only 1 in every 2^{16} pairs of values will pass. Our transform ensures that the intruder has to guess values for at most one parity check.

3.2 Transformation on the API rules

We propose a transformation allowing us to model key conjuring. This transformation is generic enough to deal with any API made up of rules satisfying the conditions given in Definition 1.

We first introduce a set of *nonces*, denoted by $\mathcal{N}$, a subset of the set of constants that does not contain the special constant 0. We assume an infinite number of nonces of both types. A nonce represents a fresh value that has been never used before. Rules obtained after transformation are called key conjuring rules and have the following form:

$$\begin{array}{ccc} x_1, \ldots, x_n & \overset{\mathsf{new\ n}}{\rightarrow} & t, \mathsf{n} \\ \mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k) & & \mathsf{chk}'_1(v_1), [\mathsf{chk}'_2(\mathsf{n})] \end{array}$$

The notation $[\mathsf{chk}'_2(\mathsf{n})]$ is used to express the fact that $\mathsf{chk}'_2(\mathsf{n})$ is optional.

Let $\mathsf{R}_l \rightarrow \mathsf{R}_r = \mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k), x_1, \ldots, x_n \rightarrow t$ be an API rule. For each i such that $1 \leq i \leq k$, since u_i is a term of Base type not headed with $\oplus$ and which contains no encryption symbol, we have that u_i is either a constant, a variable or a term of the form $\mathsf{dec}(z, t)$. In this last case, we compute the key conjuring rules associated to $\mathsf{R}_l \rightarrow \mathsf{R}_r$ as follows:

1. Let $\sigma = \{z \mapsto n\}$, we consider the new rule
$$(\mathsf{R}_l \smallsetminus \{z, \mathsf{chk}_j(u_j)\} \overset{\mathsf{new\ n}}{\rightarrow} \mathsf{R}_r \cup \{z, \mathsf{chk}_j(u_j)\})\sigma$$
2. Moreover, we have that
$$t = \bigoplus_{i=1}^{p} y_i \oplus \bigoplus_{i=1}^{\ell} c_i \oplus \bigoplus_{i=1}^{q} \mathsf{dec}(z_i, t_i).$$
for some variables y_i, z_i, some constants c_i and some terms t_i. For each j such that $1 \leq j \leq p$, we let $\sigma = \{y_j \mapsto n\}$ and we consider the new rule
$$(\mathsf{R}_l \smallsetminus \{y_j, \mathsf{chk}_j(u_j)\} \overset{\mathsf{new\ n}}{\rightarrow} \mathsf{R}_r \cup \{y_j, \mathsf{chk}_j(u_j)\})\sigma$$
Moreover, we push also on the right hand-side the check performed on y_j if such a check exists.

Given an API rule R, we denote by $\mathsf{KeyCj}(\mathsf{R})$ the set of rules obtained after applying the transformation described above. This notation is extended as expected to sets of API rules.

Example 3 *Consider the rule* R*, namely Key Part Import 3 described below.*

$$\begin{array}{l} \mathsf{y}, \mathsf{xk3}, \mathsf{xtype} \rightarrow \{\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype}) \oplus \mathsf{xk3}\}_{\mathsf{km} \oplus \mathsf{xtype}} \\ \mathsf{chkEven}(\mathsf{xtype}) \\ \mathsf{chkEven}(\mathsf{xk3}) \\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype})) \end{array}$$

The purpose of this rule is to allow a user to add a final key part xk3 *to a partial key* y *with control vector* xtype*. After applying our transformation, the set* $\mathsf{KeyCj}(\mathsf{R})$ *contains the two rules described below:*

$$\begin{array}{ll} \mathsf{xk3}, \mathsf{xtype} \overset{\mathsf{new\ n}}{\rightarrow} \{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype}) \oplus \mathsf{xk3}\}_{\mathsf{km} \oplus \mathsf{xtype}} & \\ \mathsf{chkEven}(\mathsf{xtype}) & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype})) \\ \mathsf{chkEven}(\mathsf{xk3}) & \end{array}$$

$$\begin{array}{lcr} \mathsf{y},\ \mathsf{xk3} & \overset{\mathsf{new\ n}}{\rightarrow} & \{\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n}) \oplus \mathsf{xk3}\}_{\mathsf{km} \oplus \mathsf{n}} \\ \mathsf{chkEven}(\mathsf{xk3}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n})) \\ & & \mathsf{chkEven}(\mathsf{n}) \end{array}$$

This represents the two ways the intruder can use the rule for key conjuring. In the first, he conjures a partially completed key (this is the rule used in step 1 of the Bond attack in Figure 2). In the second, for a fixed constant y*, he conjures a control vector that will allow* y *to be decrypted to form a valid partial key. Note that the conjured control vector is of even parity, so the intruder learns two parity facts in this case. Our transform allows this kind of conjuring because it is assumed the intruder can set the parity of the terms he uses as guesses. The value that is checked for even parity is under his control. Hence the probability of success is the same as for the first conjuring variant.*

The rules obtained by applying our key conjuring transformation on the IBM CCA Symmetric Key Management Transaction Set is fully described in Appendix (Figure 3).

Note that our transformation will sometimes produce rules which the intruder cannot use. This happens when the fresh nonce appears in a parity check on the left, as in the first rule for *Key Import* in Figure 3. The intruder cannot use this rule, since he does not know any parity information about the new nonce before the command is used. This corresponds to a case where the intruder would have to guess a value that decrypts to give a valid key, k, such that k also decrypts some other value to give a valid key. For single length DES keys, this gives the intruder a 1 in 2^{16} chance of success, which we consider unrealistic. However, if the intruder has extended access to a live HSM running the API, we believe our transformation could be quite naturally extended to these more costly operations (see Section 7).

3.3 Intruder rules

We denote by $\mathcal{I}$ the three API rules representing the capabilities of the intruder (see Example 1). We observe that the intruder does not have to follow any parity checks when encrypting or decrypting, but that he can also check the parity of terms he produces. Recall that parity is defined only on terms of Base type. If an intruder makes a new term by XORing, he can already predict the parity of the outcome following the rules in Section 2.2. However, when decrypting, the intruder may learn new parity information by decrypting a known constant with a random key, or by decrypting a random constant with a known key. We refer to this as offline key conjuring. The rules corresponding to this are described below:

- by decrypting a random constant with a known key

$$\mathsf{y} \overset{\mathsf{new\ n}}{\rightarrow} \begin{array}{l}\mathsf{dec(n,y),\ n} \\ \mathsf{chkX(dec(n,y))}\end{array} \quad \text{with } \mathsf{X} \in \{\mathsf{Odd}, \mathsf{Even}\}$$

Let $\mathcal{I}_1^+$ be the set of these two rules.

- decrypting a known constant with a random key

$$\mathsf{x} \overset{\mathsf{new\ n}}{\rightarrow} \begin{array}{l}\mathsf{dec(x,n),\ n,\ chkX(n)} \\ \mathsf{chkY(dec(x,n))}\end{array} \quad \text{with } \mathsf{X},\mathsf{Y} \in \{\mathsf{Odd}, \mathsf{Even}\}$$

Let $\mathcal{I}_2^+$ be the set of these four rules.

In Section 5, we will see that for a certain class of APIs, the class considered in this paper, the offline key conjuring rules can be safely ignored. Our final set of intruder rules, including offline key conjuring, is denoted by $\mathcal{I}^+ = \mathcal{I} \cup \mathcal{I}_1^+ \cup \mathcal{I}_2^+$.

4 A New Decidable Class

In this section, we define the semantics of our API-rules and we introduce the class of rules for which we prove our decidability result.

4.1 Security Problem

The problem we consider is the problem of deciding whether a particular term, for example a PIN derivation key, can be learnt by an attacker. The intruder starts with a fixed set of terms that constitute his *initial knowledge*. He can then use the rules of the API and also the key conjuring variants of the rules in any order to extend his knowledge.

We first need to make sure that parity checks are performed consistently.

Definition 2 (consistent) *Let $S = \{\mathsf{chk_1}(u_1), \ldots, \mathsf{chk_i}(u_i)\} \cup T$ where $u_1, \ldots, u_i$ are ground terms of* Base *type and T is a set of terms. We denote by $\mathsf{SatChk}(S)$ the smallest set which contains S and that is closed by application of the following rules modulo* Xor.

$$\begin{array}{rcl}\mathsf{chkEven}(x_1),\ \mathsf{chkEven}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkOdd}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkEven}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkOdd}(x_1 \oplus x_2)\end{array}$$

We say that S is consistent *if for any term t, $\mathsf{chkOdd}(t)$ and $\mathsf{chkEven}(t)$ are not both in $\mathsf{SatChk}(S)$.*

A *fact* is either a term t or a parity check, *i.e.* $\mathsf{chkX}(t)$. A fact is *ground* if the term t is ground and it is said pure if the term t is pure and of Base type inside a parity check.

Example 4 *Let S be the following set:*

$$S = \{\mathsf{chkEven}(a \oplus b),\ \mathsf{chkEven}(b \oplus c),\ \mathsf{chkOdd}(a \oplus c)\}$$

S is not consistent. Indeed, since $(a \oplus b) \oplus (b \oplus c) =_{\mathsf{Xor}} a \oplus c$, we have that $\mathsf{chkEven}(a \oplus c) \in \mathsf{SatChk}(S)$ and also that $\mathsf{chkOdd}(a \oplus c) \in \mathsf{SatChk}(S)$.

Definition 3 (one-step deducible, deducible) *Let $\mathcal{A}$ be a set of rules of the form $\mathsf{R}_l \overset{[\mathsf{new\ n}]}{\rightarrow} \mathsf{R}_r$ and E be an equational theory. Let S be a set of pure ground facts that is consistent. The set of facts F is* one-step deducible *from S if there exists a rule $\mathsf{R}_l \overset{[\mathsf{new\ n}]}{\rightarrow} \mathsf{R}_r \in \mathcal{A}$ and a ground substitution θ such that*

- *$\mathsf{R}_l\theta \subseteq \mathsf{SatChk}(S)$ (modulo E),*
- *$F = \mathsf{R}_r\theta$ (modulo E), and*
- *n is fresh, i.e. n does not occur in S.*

A term u is deducible *from S by using the set of rules $\mathcal{A}$ modulo the equational theory E, denoted by $S \vdash_{\mathcal{A},\mathsf{E}} u$ if $u \in S$ (modulo E) or there exists some sets of facts $F_1, \ldots, F_n$ such that $u \in F_n$ and F_i is one-step deducible from $S \cup F_1 \cup \ldots \cup F_{i-1}$. The sequence $F_1, \ldots, F_n$ is a* proof *that $S \vdash_{\mathcal{A},\mathsf{E}} u$.*

Of course, at each step of the proof the set of ground facts obtained has to be consistent with respect to the parity checking predicates. However, this will be the case by construction, since the only rules which add parity facts are the key conjuring ones, which always introduce something fresh in the parity facts.

Example 5 *Let $S = \{\{s\}_a,\ a \oplus b,\ b\}$. We have that s is deducible from S by using the rules $\mathcal{I}$ modulo $\mathsf{E_{API}}$. Indeed, we have that $S, \{a\}, \{s\}$ is a proof of $S \vdash_{\mathcal{I},\mathsf{E_{API}}} s$.*

Example 6 (Bond's Import/Export attack) *Let $\mathcal{A}$ be the rules described in Figure 1, $\mathsf{V} = \{\mathsf{pin},\ \mathsf{data},\ \mathsf{exp},\ \mathsf{imp},\ \mathsf{kp}\}$. Let S be a set which contains:*

- *$\{\mathsf{pdk}\}_{\mathsf{km} \oplus \mathsf{pin}}$, $\mathsf{chkOdd}(\mathsf{pdk})$, $\mathsf{chkOdd}(\mathsf{km})$,*
- *t and $\mathsf{chkEven}(t)$ for any $t \in \mathsf{V}$.*

We have that $\{\mathsf{pdk}\}_{\mathsf{km} \oplus \mathsf{data}}$ is deducible from S by using the rules in $\mathcal{A} \cup \mathsf{KeyCj}(\mathcal{A}) \cup \mathcal{I}$ modulo $\mathsf{E_{API}}$. The proof witnessing this fact can be easily extracted from Figure 2.

Note that this attack involves two online key conjuring steps. Each key conjuring attempt has a 1 in 256 chance of success, due to the parity checks. Each time the adversary wants to conjure a key, it requires a significant amount of access to the API. We assume in what follows that the use of these rules by the adversary is limited. This is modelled by introducing a parameter k that bounds the maximum number of applications of the key conjuring rules induced by the protocol. The value of k could be set based on the amount of time an attacker may have access to a live HSM, based on physical security measures, auditing procedures in place, etc. Note however that we do not bound the number of offline key conjuring since it is much easier for an adversary to try numerous values off-line.

Formally, we write $S \vdash_{\mathcal{A}_1,\mathsf{E_{API}}}^{\mathcal{A}_2 \leq k} u$ if u is deducible from S by using the rules in $\mathcal{A}_1$ and at most k instances of the rule in $\mathcal{A}_2$ (modulo $\mathsf{E_{API}}$). In this paper we rely on a fixed equational theory, denoted by $\mathsf{E_{API}}$ (see Section 2.2) and a fixed set of intruder rules denoted by $\mathcal{I}^+$. Hence our problem is the following one:

Security Problem

Entries: A finite set $\mathcal{A}$ of API rules, a set S of pure ground facts that is consistent (the initial knowledge of the attacker), a pure ground term s (the secret) and a bound $k \in \mathbb{N}$ (number of key conjuring steps).

Question: Is the secret s deducible from S by using the rules in $\mathcal{A} \cup \mathcal{I}^+$ and at most k instances of rules in $\mathsf{KeyCj}(\mathcal{A})$ (modulo $\mathsf{E_{API}}$), *i.e.* does $S \vdash_{\mathcal{A}\cup\mathcal{I}^+,\mathsf{E_{API}}}^{\mathsf{KeyCj}(\mathcal{A}) \leq k} s$?

4.2 Well-formed API

API-rules as defined in Definition 1 are slightly too general for our decidability result. Hence we introduce further assumptions, that we believe are very reasonable in practice. Note that these hypotheses are checked on the API rules before performing the key conjuring transformation.

Definition 4 *Let S_0 be a set of pure ground fact that is consistent. Let $\mathsf{R} = \mathsf{R}_l \xrightarrow{\mathsf{new}\ \mathsf{n}} \mathsf{R}_r$ be a rule and t be a term of* Base *type. We say that t* is checked in R *w.r.t.* S_0 *if* $\mathsf{chkX}(t) \in \mathsf{SatChk}(S_0 \cup \mathsf{R}_l \cup \mathsf{R}_r)$.

Definition 5 *Let* R *be a rule.* $\mathsf{KeyTerm}(\mathsf{R})$ *are the subterms of* R *which appear at a key position. More formally,* $\mathsf{KeyTerm}(\mathsf{R}) = \{\mathsf{KeyTerm}(t) \mid t \in \mathsf{R} \text{ or } \mathsf{chkX}(t) \in \mathsf{R}\}$ *where* $\mathsf{KeyTerm}(t)$ *is defined as follows:*

$$\begin{array}{rcl}\mathsf{KeyTerm}(t) & = & \{u_2 \mid \mathsf{dec}(u_1,u_2) \in st(t) \textit{ for some } u_1\} \\ & & \cup\ \{u_2 \mid \{u_1\}_{u_2} \in st(t) \textit{ for some } u_1\}.\end{array}$$

We will restrict our attention to APIs such that a term which appears at a key position has to be parity checked. This hypothesis is natural, since it corresponds to the API designer being consistent about checking the parity of keys before they are used.

Example 7 *Let* $\mathsf{V} = \{\mathsf{imp}, \mathsf{kp}, \mathsf{exp}, \mathsf{pin}, \mathsf{data}\}$. *and S_0 be a set that is consistent and which contains at least* $\mathsf{chkEven}(t)$ *for any $t \in \mathsf{V}$ and* $\mathsf{chkOdd}(\mathsf{km})$. *The rules given in Figure 1 are such that each term which appears at a key position is checked w.r.t.* S_0.

Definition 6 (dec-property) *Let T be a set of terms. We say that T has the* dec*-property if*

$$\mathsf{dec}(x, v_1), \mathsf{dec}(x, v_2) \in st(T) \ \Rightarrow\ v_1 = v_2.$$

We say that a rule R *has the* dec*-property if the set of terms* $T = \{t \mid t \in \mathsf{R} \text{ or } \mathsf{chkX}(t) \in \mathsf{R}\}$ *satisfies the* dec*-property.*

In the API we consider, we will assume that all the rules satisfy the dec-property. This hypothesis is natural, since it only forbids the API from decrypting the same input under two different keys. Note that the dec-property is clearly satisfied by the rules given in Figure 1.

Definition 7 (well-formed API rule) *Let S_0 be a set of pure ground fact that is consistent. Let* R *be an API rule.*

$$\mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k), x_1, \ldots, x_n \to t$$

We say that R *is* well-formed *w.r.t.* S_0 *if:*

- *for all i such that $1 \leq i \leq k$, we have that $u_i \in st(t)$,*
- R *satisfies the* dec*-property,*
- *for all* $v \in \mathsf{KeyTerm}(\mathsf{R})$, *$v$ is checked in* R *w.r.t.* S_0.

An API rule satisfying only the two first points is said to be weakly well-formed.

The first point requires that the API only checks the parity of objects that are to be used in generating the output. Since the form of our rules has only variables on the left, and all decryption explicitly stated on the right, this is quite natural. We would not expect an API to check the parity of a term that is subsequently discarded. For instance, the API rules given in Figure 1 are well-formed. However, the rules describing the capabilities of the attacker (see Example 1) are not well-formed, but only weakly well-formed.

4.3 Decidability

Theorem 1 (Main result) *Let $\mathcal{P}$ be an instance of the security problem (as stated at the end of Section 4.1) where*

- *the set $\mathcal{A}$ of API rules is well-formed w.r.t. the set S*

- $0 \in S$,
- *the terms in* $S \cup \{s\}$ *do not contain any symbol* dec.

We can decide whether $\mathcal{P}$ *is a positive or a negative instance of the security problem.*

The remainder of the paper is devoted to the proof of this result. We proceed in several steps:

1. *From* $\mathcal{I}^+$ *to* $\mathcal{I}$. In Section 5, we establish some reduction results allowing us to get rid of the offline key conjuring rules. These results are obtained for any set of API rules as defined in Definition 1, and not only the well-formed ones introduced in Definition 7.

2. *From* $\mathsf{E_{API}}$ *to* AC. In Section 6.1, we show that we can get rid of some axioms of the equational theory by using the fact that $\mathsf{E_{API}}$ satisfies the finite variant property introduced in [6]. This can be done safely by considering some new rules, namely the variants denoted $\mathcal{V}ar(\mathcal{A})$, which are obtained from the rules $\mathcal{A}$ we have at the beginning by instantiating them.

3. *Controlling the form of the rules.* In Section 6.2, we show that the variants computed at the previous step satisfy some properties. Given a set $\mathcal{A}$ of (weakly) well-formed API rules, we have that $\mathcal{V}ar(\mathcal{A} \cup \mathsf{KeyCj}(\mathcal{A}))$, rules obtained after our both transformations, are (weakly) well-adapted (see Definition 9).

4. *Existence of a pure attack.* In Section 6.3, we show that for a set of weakly well-adapted rules, if there exists an attack then there is one which only involves pure terms (see Proposition 5).

5. *Bounded the number of subterms headed with* dec. Now, to obtain our decidability result it is sufficient to bound the number of terms headed with dec in an attack (see Section 6.4). This allows us to consider only a finite number of terms.

5 Off-line key conjuring is useless

The adversary can perform as many off-line key conjuring as he wishes, since it is very easy for him to try numerous values off-line, until the decryption algorithm yields a bitstring of the desired parity. We show now that in fact, off-line key conjuring does not provide any extra possibilities for the adversary to mount an attack. Thus there is no need to consider these rules.

We first show that the rules of $\mathcal{I}_1^+$ are useless as soon as the adversary knows a fixed constant of each parity.

Proposition 1 *Let* $\mathcal{A}$ *be a set of API rules and* S *be a set of pure ground facts. We have that*

$$S \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I}^+, \mathsf{E_{API}}} u \quad \Leftrightarrow \quad S' \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I} \cup \mathcal{I}_2^+, \mathsf{E_{API}}} u$$

where $S' = S \cup \{c_1, c_2, \mathsf{chkOdd}(\mathsf{c_1}), \mathsf{chkEven}(\mathsf{c_2})\}$ *and* c_1, c_2 *are constants of* Base *type that do not appear in* $\mathcal{A}$, S *and* u.

Then, we show that there is no need to consider rules of $\mathcal{I}_2^+$ if the intruder already knows terms of the form $\mathsf{dec}(1, c_i)$ of each parity. Intuitively, the intruder knows an instance of each of the four rules.

Proposition 2 *Let* $\mathcal{A}$ *be a set of API rules and* S *be a set of pure ground facts. We have that*

$$S \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I} \cup \mathcal{I}_2^+, \mathsf{E_{API}}} u \quad \Leftrightarrow \quad S' \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I}, \mathsf{E_{API}}} u$$

where S' *is the set obtained from* S *by adding*

- *the constants* 1 *(*Cipher*) and* c_1, c_2, c_3, c_4 *(*Base*),*
- $\mathsf{chkOdd}(\mathsf{dec}(1, c_1)), \mathsf{chkOdd}(c_1)$
- $\mathsf{chkOdd}(\mathsf{dec}(1, c_2)), \mathsf{chkEven}(c_2)$
- $\mathsf{chkEven}(\mathsf{dec}(1, c_3)), \mathsf{chkOdd}(c_3)$
- $\mathsf{chkEven}(\mathsf{dec}(1, c_4)), \mathsf{chkEven}(c_4)$

and c_1, c_2, c_3, c_4 *do not appear in* $\mathcal{A}$, S *and* u.

The idea of the proof is to replace each application of a rule

$$\mathsf{x} \xrightarrow{\mathsf{new\ n}} \mathsf{dec}(\mathsf{n}, \mathsf{x}), \mathsf{chkX}(\mathsf{dec}(\mathsf{x}, \mathsf{n})),\ \mathsf{chkY}(\mathsf{n})$$

in $\mathcal{I}_2^+$ by its corresponding instance. In particular, x is always replaced by the same constant 1. We can show that we still obtain a proof. Intuitively, if it was not the case, it would mean that it was important for x to be an encryption or a decryption. This would be the case only if there was nested encryption on the right hand side of the rule, which is not the case for API rules.

6 Decidability for Well-Formed APIs

In the remainder of this section we describe a decision procedure to deal with any set of well-formed API rules.

6.1 Getting rid of some equations

The goal of this section is to get rid of all the axioms of the equational theory but associativity and commutativity, decomposing the theory into a convergent rewriting system modulo AC equations. The idea is to pre-compute variants of the rules so that there is no need to apply the full equational theory anymore.

Let $\mathcal{R}$ be a term rewriting system (TRS) and E' be an equational theory, we write $u \rightarrow_{\mathcal{R},\mathsf{E}'} v$ when v can be written into v modulo E'. A decomposition of an equational theory E is a pair $(\mathcal{R}, \mathsf{E}')$ such that $\mathcal{R}$ is an E'-convergent system for E, *i.e.* $u =_{\mathsf{E}_{\mathsf{API}}} v$ if and only if $u{\downarrow} = v{\downarrow}$ where $u{\downarrow}$ denotes the normalised form of u w.r.t. $\rightarrow_{\mathcal{R},\mathsf{E}'}$.

For instance, for the equational theory $\mathsf{E}_{\mathsf{API}}$, we can show that $(\mathcal{R}_\oplus, \mathsf{AC})$ is a decomposition of $\mathsf{E}_{\mathsf{API}}$ where

$$\mathsf{R}_\oplus = \begin{cases} \mathsf{dec}(\{x\}_y, y) \rightarrow x & x \oplus x \rightarrow 0 \\ \{\mathsf{dec}(x,y)\}_y \rightarrow x & x \oplus 0 \rightarrow 0 \\ x \oplus (x \oplus y) \rightarrow y & \end{cases}$$

Definition 8 (finite variant property) *A decomposition* $(\mathcal{R}, \mathsf{E}')$ *of a given theory* E *has the* finite variant property *if for every term t, there is a finite set of substitutions $\Sigma(t)$ such that*

$$\forall \sigma \exists \theta \in \Sigma(t), \exists \tau \text{ such that } \sigma{\downarrow} =_{\mathsf{E}'} \theta\tau \wedge (t\sigma){\downarrow} =_{\mathsf{E}'} (t\theta){\downarrow}\tau.$$

In other words, all possible reductions in an instance of t can be computed in advance. Given a term t, we denote by $\mathcal{V}ar(t)$ the set of its variants, *i.e.* $\mathcal{V}ar(t) = \{(t\theta){\downarrow} \mid \theta \in \Sigma(t)\}$. In [6], the authors give sufficient condition to establish that a given presentation satisfies the finite variant property. Moreover they give an algorithm allowing us to compute the variants associated to a given term. By using their result, it is easy to establish that $(\mathcal{R}_\oplus, \mathsf{AC})$ is a decomposition of $\mathsf{E}_{\mathsf{API}}$ which satisfies the finite variant property. The so-called variants of a rule R are obtained by performing narrowing with $\mathcal{R}_\oplus$ modulo AC.

Narrowing. The subterm of t at position $p \in \mathcal{O}(t)$ is written $t|_p$. The term obtained by replacing $t|_p$ with u is denoted $t[u]_p$. We denote by $\bar{\mathcal{O}}(t)$ the set of non-variable position of t. Given a TRS $\mathcal{R}$, we say that a term t *narrows to t' with the substitution σ*, at $p \in \bar{\mathcal{O}}(t)$, by $l \rightarrow r \in \mathcal{R}$ if there exists a renaming $l' \rightarrow r'$ of $l \rightarrow r \in \mathcal{R}$ such that σ is a unifier of $t|_p$ and l' and $t' = (t[r]_p)\sigma$. In this case, we write $t \rightsquigarrow_\sigma t'$. We write $t \overset{*}{\rightsquigarrow}_\sigma t'$ if there exists a narrowing derivation $t = t_1 \rightsquigarrow_{\sigma_1} t_2 \ldots \rightsquigarrow_{\sigma_{n-1}} t_n = t'$ such that $\sigma = \sigma_1 \ldots \sigma_{n-1}$. If E' is a set of equations such that an E'-unification algorithm exists, we define E'-narrowing as expected (σ is an E'-unifier of $t|_p$ and l). In particular, this allows us to define AC-narrowing.

Computation of the variants. Let R be an API rule and k be the number of occurrences of $\{\_\}_\_$, dec and $\oplus$. According to [6], we have that

$$\mathcal{V}ar(\mathsf{R}) = \{\mathsf{R}' \mid \mathsf{R} \overset{*}{\rightsquigarrow}_\sigma \mathsf{R}' \text{ by a derivation of length at most } k\}$$

Now the proposition below is an easy consequence of the fact that $\mathsf{E}_{\mathsf{API}}$ satisfies the finite variant property.

Proposition 3 *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be two sets of rules, S be a set of ground facts and s be a ground term (in normal form).*

$$S \vdash^{\mathsf{KeyCj}(\mathcal{A}_1)\ \le k}_{\mathcal{A}_1 \cup \mathcal{A}_2, \mathsf{E}_{\mathsf{API}}} u \text{ if and only if } S \vdash^{\mathcal{V}ar(\mathsf{KeyCj}(\mathcal{A}_1))\ \le k}_{\mathcal{V}ar(\mathcal{A}_1 \cup \mathcal{A}_2), \mathsf{AC}} u$$

Moreover, we only need to consider instances of the rules which involve terms in normal form.

Example 8 *For instance, consider the following rule* $\mathsf{R} = x, y \rightarrow \mathsf{dec}(x,y)$. *We have that* $\mathcal{V}ar(\mathsf{R}) = \{\mathsf{R}, \mathsf{R}'\}$ *where* $\mathsf{R}' = \{z\}_y, y \rightarrow z$. *Note that* R' *is a normalised instance of* R. *Indeed* $\mathsf{R}' = \mathsf{R}\theta{\downarrow}$ *where* $\theta = \{x \mapsto \{z\}_y\}$.

6.2 Controlling the form of the rules

We need to control the form of the rules after computation of the key conjuring transformation and computation of the variants. We show that the set $\mathcal{V}ar(\mathcal{A} \cup \mathsf{KeyCj}(\mathcal{A}))$ obtained from a set $\mathcal{A}$ which only contains (weakly) well-formed rules w.r.t. S is (weakly) well-adapted w.r.t. S.

Definition 9 (well-adapted) *Let S_0 be a set of pure ground fact that is consistent. Let* $\mathsf{R} = \mathsf{R}_l \xrightarrow{[\mathsf{new}\ \mathsf{n}]} \mathsf{R}_r$. *We say that* R *is* well-adapted *w.r.t. S_0 if*

1. R *is well-typed and* $vars(\mathsf{R}_r) \subseteq vars(\mathsf{R}_l)$,
2. *a term of type* Cipher *appearing as a strict subterm position in* R *is either a nonce or a variable,*
3. *for all* $t \in \mathsf{KeyTerm}(\mathsf{R})$, *$t$ is checked in* R *w.r.t. S_0,*
4. *there is at most one term u in a check in* R_r *not equal to* n *and we are in one of the following cases:*
   - $u = \mathsf{dec}(y, \mathsf{n} \oplus u')$,
   - $u = \mathsf{dec}(\mathsf{n}, u')$, *or*
   - n *occurs in* R_l *and hence the rule* R *is useless.*

A set of rules which satisfies the two first points is said to be *weakly well-adapted*.

Proposition 4 *Let S_0 be a set of pure ground fact that is consistent. Let $\mathcal{A}$ be a set of (weakly) well-formed API rules w.r.t. S_0. Let* $\mathcal{A}' = \mathcal{V}ar(\mathcal{A} \cup \mathsf{KeyCj}(\mathcal{A}))$. *We have that $\mathcal{A}'$ is a set of (weakly) well-adapted rules w.r.t. S_0.*

The notion of well-adapted relies on four conditions (see Definition 9). The conditions 1, 3 and 4 are established by using the fact that a variant R' is just a normalised instance of well-formed API rule R, that is $\mathsf{R}' = \mathsf{R}\theta{\downarrow}$ for some θ.

Proving condition 2 is more involved. As shown in the example below, Condition 2 is not stable by AC-narrowing, *i.e.* by computation of the variants, thus we had to first enforce it.

Example 9 *Let* $\mathsf{R} = x \to \{\mathsf{dec}(x, k_1)\}_{\mathsf{dec}(x,k_2)}$. *The condition 2 is satisfied by* R. *Now, consider the rule*

$$\mathsf{R}' = \{y\}_{k_1} \to \{y\}_{\mathsf{dec}(\{y\}_{k_1},k_2)}$$

We have that $\mathsf{R}' \in \mathcal{V}ar(\mathsf{R})$. *However,* R' *does not satisfy the condition. This problem comes from the fact that there is a variable of type* Cipher *which involved in two different subterms headed with* dec. *Here we have that* x *is involved in* $\mathsf{dec}(x, k_1)$ *and also in* $\mathsf{dec}(x, k_2)$. *Since* $k_1 \neq k_2$, *the rule* R *does not satisfy the* dec*-property and hence is not a well-formed rule.*

6.3 Existence of a pure attack

We show in this section that we can restrict our attention to proofs which only involve pure terms. The following result holds for any set of weakly well-adapted rules. The conditions 3 and 4 of Definition 9 are only used for the last part of our decision procedure (see Section 6.4).

A position in a term is *impure* if the subterm at that position is not of the expected type and form. By convention the root position is always an impure position. Note that in a pure term t the only impure position is Λ.

Example 10 *Let* $t = \mathsf{dec}(a \oplus b, c)$ *where* a, b *and* c *are constant of* Base *type. The position* p *in* t *such that* $t|_p = a \oplus b$ *is impure. Let* $t = \mathsf{dec}(\{a\}_b, c)$ *where* a, b *and* c *are constant of* Base *type. The position* p *in* t *such that* $t|_p = \{a\}_b$ *is impure.*

We first prove that whenever an impure term occurs in a deducible term t at a position p, the term $t|_p$ is itself deducible.

Lemma 1 *Let* $\mathcal{A}$ *be a set of weakly well-adapted rules and* S *be a set of pure ground facts that is consistent and which contains* 0. *Let* u *be a ground term deducible from* S *and* $F_1, \ldots, F_n$ *be a proof that* $S \vdash_{\mathcal{A},\mathsf{AC}} u$. *Let* p *be an impure position of* u. *We have that* $u|_p \in S \cup F_1 \cup \ldots \cup F_n$.

We are now ready to state our result which says that only pure terms need to be considered when checking for deducibility.

Proposition 5 *Let* $\mathcal{A}$ *be a set of weakly well-adapted rules and* S *be a set of pure ground facts that is consistent and which contains* 0. *Let* u *be a pure ground term. If* $S \vdash_{\mathcal{A},\mathsf{AC}} u$ *then there is a proof of* $S \vdash_{\mathcal{A},\mathsf{AC}} u$ *which only involve pure terms.*

To establish this result, we assume we are given a proof P of $S \vdash_{\mathcal{A},\mathsf{AC}} u$, and we show how to compute a proof P' from P which only involves pure terms. The proof P' uses exactly the same rule at each step but not the same instance. In particular any term appearing at an impure position will be replaced to obtain a pure term. From this, we easily deduce the following corollary.

Corollary 1 *Let* $\mathcal{A}_1$, $\mathcal{A}_2$ *be two set of weakly well-adapted rules and* S *be a set of pure ground facts that is consistent and which contains* 0. *Let* u *be a pure ground term. If* $S \vdash^{\mathcal{A}_1 \leq k}_{\mathcal{A}_2,\mathsf{AC}} u$ *then there is a proof witnessing this fact which only involves pure terms.*

6.4 A bound on the number of dec terms

From Corollary 1 we know that if there is an attack, there is an attack that involves only pure terms. Pure terms are well-typed and contain at most one encryption symbol. However, the dec symbols might be arbitrarily nested. Our goal is to bound the size of an attack by limiting the use of dec symbols.

A dec-term is a term of the form $\mathsf{dec}(u, v)$. Given a proof $F_1, \ldots, F_n$ of some deduction fact $S \vdash_{\mathcal{R},\mathsf{AC}} w$, we say that a dec-term t is *legal* if it is checked in S, that is $\mathsf{chkX}(t) \in \mathsf{SatChk}(S)$ or it has been produced by a key-conjuring rule, that is $\mathsf{chkX}(t) \in F_j$ for some $1 \leq j \leq n$. The term t is said to be *illegal* otherwise. Let k' be the number of legal dec-terms occurring in S. Since there are at most k applications of the key-conjuring rules and since each key-conjuring rule introduces at most one term that is not a name, there are at most $k + k'$ legal dec-terms occurring as subterm in a proof $F_1, \ldots, F_n$. We wish to show that, besides the legal dec-terms, no decryption symbol can occur under a key position. This ensures that illegal dec-terms can only occur as plaintext thus can not be nested.
We first show that illegal dec-term cannot occur in checks.

Lemma 2 (No illegal dec-term in checks) *Let* $\mathcal{A}$ *be a set of well-adapted rules and* S *be a set of pure ground facts such that no* dec *terms occurs in* $\mathsf{KeyTerm}(S)$. *Let* w *be a pure ground term deducible from* S *and* $F_1, \ldots, F_n$ *be a proof that* $S \vdash_{\mathcal{A} \cup \mathcal{V}ar(\mathcal{I}),\mathsf{AC}} w$ *that involves only pure facts. We assume that there is no* dec*-term subterm of* w. *For any term* t *such that* $\mathsf{chk}(t) \in \mathsf{SatChk}(S \cup F_1 \cup \ldots \cup F_n)$, *for any* $\mathsf{dec}(u, v)$ *subterm of* t, *the* dec*-term* $\mathsf{dec}(u, v)$ *is legal.*

The intuitive idea for proving this lemma is that new checks can only be introduced by the key-conjuring rules, which are limited. In addition, when a $\mathsf{chkX}(t)$ is introduced, illegal dec-terms cannot occur since the rules are well-adapted.

We then prove that illegal dec-terms cannot occur in key position or they can be replaced by 0. Let N and N' be two terms. For any term M, we denote by $M\delta_{N,N'}$ the term M where any occurrence of N in *key position* is replaced by N'.

Lemma 3 (Replacement of dec-terms in key position) *Let $\mathcal{A}$ be a set of well-adapted rules and S be a set of pure ground facts such that no* dec *terms occurs in* $\mathsf{KeyTerm}(S)$. *Let w be a pure ground term deducible from S and $F_1, \ldots, F_n$ be a proof that $S \vdash_{\mathcal{A} \cup \mathcal{V}ar(\mathcal{I}), \mathsf{AC}} w$ that involves only pure facts. We assume that there is no* dec*-term subterm of w. Let t be a term such that $t \in F_j$ for some $1 \leq j \leq n$ and let p be some key position of t such that $t|_p = \mathsf{dec}(u, v) \oplus t'$ (t' being possibly empty in which case by convention, $t|_p = \mathsf{dec}(u, v)$).*

- *Either the term $\mathsf{dec}(u, v)$ is legal.*
- *Or $F_1\delta_{(\mathsf{dec}(u,v)\oplus t',0)}, \ldots, F_j\delta_{(\mathsf{dec}(u,v)\oplus t',0)}$ is a pure proof of $S \vdash_{\mathcal{R} \cup \mathcal{V}ar(\mathcal{I}), \mathsf{AC}} t\delta_{(\mathsf{dec}(u,v)\oplus t',0)}$.*

The lemma is proved by induction.

Now, we are able to prove our main result (Theorem 1).
Proof. Let P be an instance of the security problem where the set $\mathcal{A}$ of API rules is well-formed w.r.t. S and $0 \in S$. Let S' be the set of facts obtained from S by adding

- 1 (constant of type Cipher),
- $c_1, c_2, c'_1, c'_2, c'_3, c'_4$ constants of Base type,
- $\mathsf{chkOdd}(c_1)$, $\mathsf{chkOdd}(c'_1)$, $\mathsf{chkOdd}(c'_3)$,
- $\mathsf{chkEven}(c_2)$, $\mathsf{chkEven}(c'_2)$, $\mathsf{chkEven}(c'_4)$,
- $\mathsf{chkOdd}(\mathsf{dec}(1, c'_1))$, $\mathsf{chkOdd}(\mathsf{dec}(1, c'_2))$,
- $\mathsf{chkOdd}(\mathsf{dec}(1, c'_3))$, $\mathsf{chkOdd}(\mathsf{dec}(1, c'_4))$.

Note that no dec terms occurs in $\mathsf{KeyTerm}(S')$.

Thanks to Propositions 1 and 2, we easily deduce that

$$S \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I}^+, \mathsf{E_{API}}} u \quad \Leftrightarrow \quad S' \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I}, \mathsf{E_{API}}} u$$

Proposition 3 gives us

$$S' \vdash^{\mathsf{KeyCj}(\mathcal{A}) \leq k}_{\mathcal{A} \cup \mathcal{I}, \mathsf{E_{API}}} u \quad \Leftrightarrow \quad S' \vdash^{\mathcal{V}ar(\mathsf{KeyCj}(\mathcal{A})) \leq k}_{\mathcal{V}ar(\mathcal{A} \cup \mathcal{I}), \mathsf{AC}} u$$

Thanks to the well-formedness of the rules in $\mathcal{A}$, we deduce (Proposition 4) that

- the rules in $\mathcal{V}ar(\mathsf{KeyCj}(\mathcal{A}))$ are well-adapted,
- the rules in $\mathcal{V}ar(\mathcal{A})$ are well-adapted,
- the rules in $\mathcal{V}ar(\mathcal{I})$ are weakly well-adapted.

Note also that $\mathcal{V}ar(\mathcal{A} \cup \mathcal{I}) = \mathcal{V}ar(\mathcal{A}) \cup \mathcal{V}ar(\mathcal{I})$.

Now, we apply Corollary 1 and we deduce that if $S' \vdash^{\mathcal{V}ar(\mathsf{KeyCj}(\mathcal{A})) \leq k}_{\mathcal{V}ar(\mathcal{A} \cup \mathcal{I}), \mathsf{AC}} u$ then there exists a proof witnessing this fact which involves only pure terms. Lastly, Lemmas 2 and 3 allow us to bound the number of dec-terms which can appear in such a proof. This allows us to consider only a finite number of terms: we have a finite number of constants and nonces which can only be combined to produce pure terms involving some precise dec-terms. $\square$

Complexity. Our decision procedure works as follows. We first guess the k legal terms that are produced by key conjuring rules and then saturate the set S' with all deducible terms that are pure terms with no illegal dec terms under key position. Let n by the number of constants occurring in S' plus k. Illegal dec terms cannot occur nested thus it is easy to see that there are at most $n \times 2^n$ illegal dec terms. These dec terms can be arbitrarily XORed in plaintext position but cannot occur under key position. Thus we have to consider at most $2^{2^{\mathcal{O}(n)}}$ terms. Thus our procedure terminates after at most $2^{2^{\mathcal{O}(n)}}$ steps. Altogether, we can show that our algorithm is non-deterministic 2-EXPTIME.

6.5 Related work

The class of well-formed API rules is related to the class proposed in [8]. There it is shown that secrecy preservation of protocols is decidable for an unbounded number of sessions for protocols with XOR, provided they can be expressed with rules in the WFX-class, that is, a set of rules of the form $t_1, \ldots, t_n \to t_{n+1}$ where each t_j is

- either a *xor term* that is $t_j = \bigoplus_{i=1}^{n} u_i$, $n \geq 1$ where each u_i is a variable or a constant.
- or $t_j = \{u\}_v$ where u and v are xor terms.

This is intuitively related to our notion of *well-typed terms* that ensures in particular that at most one encryption symbol can appear in a term. However, there are two main differences between the class of well-formed API rules introduced in this paper and the WFX-class.

1. We consider here an equational theory with explicit decryption. This is necessary for modelling key-conjuring. Adding the two equations for encryption and decryption requires a much more careful treatment when proving that whenever there is an attack, there is an attack that involves only pure terms.
2. In the work presented here, it is not sufficient to bound the number of encryption symbols, as in [8]. Indeed, there are an infinite number of well-typed terms since the number of nested decryption symbols is not

bounded by typing. Thus we had to show that it is not necessary to consider nested decryption symbols except for a finite number of terms, coming from the application of key-conjuring rules (the *legal* dec terms).

To conclude, the two classes are formally incomparable. While well-formed API rules enable explicit decryption, thus potentially more attacks, there are no equality checks between components of the received messages. For example, the following rule

$$\{x\}_{k_1}, \{x\}_{k_2} \rightarrow k_3$$

belongs to the WFX-class but cannot be expressed as a well-formed API rule. We would need to extend API rules with equality checks.

To the best of our knowledge, there exist only two other decidable classes [5, 13] for secrecy preservation for protocols with XOR, for an unbounded number of sessions. In both cases, the main difference with our class is that we make restrictions on the combination of functional symbols rather than on the occurrences of variables. As a consequence, our class is incomparable to the two existing ones. In particular, the IBM CCA protocol cannot be modelled in either of these two other classes.

7 Conclusion

We have presented a formalism for key conjuring, obtained by applying a transformation to a model of a security API with explicit parity checks. We have shown that the security problem is decidable for a general class of APIs (well-formed).

In this paper, we have concentrated on the example of the IBM CCA API, which uses parity checks to validate DES keys. However, we believe our approach can be applied in general to security API analysis, where other functions may be used to check validity of keys. In particular, our language for defining API commands, with variables on the left hand side and all decryption made explicit on the right, seems more natural than the use of an Alice-Bob style implicit decryption formalism. It would enable the detection of the so-called 'trojan key' attacks of the type described by Clulow, [4]. We could also extend our transformation to allow more computationally expensive key conjuring operations, by allowing multiple fresh terms to be generated in a single rule.

We plan to extend our results to a larger class of APIs, incorporating pairing and further cryptographic primitives, and to implement our model in an analysis tool. There remains a significant class of known API attacks that has not been dealt with formally: so-called parallel key search attacks. Formalising key conjuring is an important first step towards this, since many of these attacks rely on building up a set of conjured keys.

References

[1] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris (France), 2001. Springer-Verlag.

[2] *CCA Basic Services Reference and Guide*, Oct. 2006. Available online.

[3] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded System (CHES'02)*, volume 2523 of *LNCS*, pages 579–592, Redwood Shores (CA, USA), 2003. Springer.

[4] J. Clulow. On the security of PKCS#11. In *Proceedings of the 5th International Worshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425, Cologne (Germany), 2003. Springer-Verlag.

[5] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *LNCS*, pages 148–164, Valencia, Spain, 2003. Springer-Verlag.

[6] H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *LNCS*, pages 294–307, Nara (Japan), 2005. Springer.

[7] V. Cortier, S. Delaune, and G. Steel. A formal theory of key conjuring. Research Report 6134, INRIA, Feb. 2007. 41 pages.

[8] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of xor-based key management schemes. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, LNCS, pages 538–552, Braga (Portugal), 2007. Springer-Verlag.

[9] J. Courant and J.-F. Monin. Defending the bank with a proof assistant. In *Proceedings of Workshop on Issues in the Theory of Security (WITS '06)*, Vienna (Austria), March 2006.

[10] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[11] G. Keighren. Model checking IBM's common cryptographic architecture API. Informatics Research Report EDI-INF-RR-0862, University of Edinburgh, 2006.

[12] G. Steel. Deduction with XOR constraints in security API modelling. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *LNCS*, pages 322–336, Tallinn (Estonia), 2005. Springer-Verlag.

[13] K. N. Verma, H. Seidl, and T. Schwentick. On the complexity of equational Horn clauses. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *LNCS*, pages 337–352, Tallinn (Estonia), 2005. Springer-Verlag.

[14] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.

A Existence of a pure attack

Lemma 1 *Let $\mathcal{A}$ be a set of weakly well-adapted rules and S be a set of pure ground facts that is consistent and which contains 0. Let u be a ground term deducible from S and $F_1, \ldots, F_n$ be a proof that $S \vdash_{\mathcal{A},\mathsf{AC}} u$. Let p be an impure position of u. We have that $u|_p \in S \cup F_1 \cup \ldots \cup F_n$.*

Proof. The proof is by induction on the number of steps needed to obtain u. The base case, *i.e.* $u \in S$, is trivial. For the induction step, we have that there exists a weakly well-adapted rule $\mathsf{R}_l \overset{\mathsf{new}\ \mathsf{n}}{\rightarrow} \mathsf{R}_r$ and a ground substitution θ such that $R_l\theta \subseteq \mathsf{SatChk}(S \cup F_1 \cup \ldots \cup F_{n-1})$ and $u \in F_n = \mathsf{R}_r\theta$ (modulo AC). Let p be an impure position in u.

- either $p = \Lambda$ and in such a case we have that $u|_p \in S \cup F_1 \cup \ldots \cup F_n$,
- or $u|_p$ is a strict subterm of u. Since $\mathsf{R}_l \overset{\mathsf{new}\ \mathsf{n}}{\rightarrow} \mathsf{R}_r$ is a weakly well-adapted rule, $u|_p$ must be a subterm of $x\theta$ for some variable $x \in \mathsf{R}_r$. Since $vars(\mathsf{R}_r) \subseteq vars(\mathsf{R}_l)$, we have that there exists $t \in \mathsf{R}_l$ such that $t\theta \in S \cup F_1 \cup \ldots \cup F_{n-1}$ and $u|_p \in st(t\theta)$. Moreover, we can easily check that $u|_p$ appears at an impure position in $t\theta$. By induction hypothesis, we deduce that $u|_p \in S \cup F_1 \cup \ldots \cup F_{n-1}$ and thus $u|_p \in S \cup F_1 \cup \ldots \cup F_n$. □

Proposition 5 *Let $\mathcal{A}$ be a set of weakly well-adapted rules and S be a set of pure ground facts that is consistent and which contains 0. Let u be a pure ground term. If $S \vdash_{\mathcal{A},\mathsf{AC}} u$ then there is a proof of $S \vdash_{\mathcal{A},\mathsf{AC}} u$ which only involve pure terms.*

We define the function $\overline{\cdot}$ over ground terms that replaces any term at an impure position by 0 (neutral element of $\oplus$) or 1 (constant of type Cipher). More formally $\overline{\cdot}$ is inductively defined as follows:

$$\begin{array}{rcll}
\overline{u} & = & u & \text{if } u \text{ is a variable or a constant} \\
\overline{u_1 \oplus u_2} & = & \overline{u_1}^0 \oplus \overline{u_2}^0 & \\
\overline{\mathsf{dec}(u_1, u_2)} & = & \mathsf{dec}(u_1, \overline{u_2}^0) & \text{if } u_1 \in \mathcal{N} \text{ of type Cipher} \\
\overline{\mathsf{dec}(u_1, u_2)} & = & \mathsf{dec}(1, \overline{u_2}^0) & \text{otherwise} \\
\overline{\{u_1\}_{u_2}} & = & \{\overline{u_1}^0\}_{\overline{u_2}^0} &
\end{array}$$

where $\overline{\cdot}^0$ are defined by:

$$\begin{array}{rcll}
\overline{u}^0 & = & u & \text{if } u \text{ is a variable or a constant of base type} \\
\overline{u_1 \oplus u_2}^0 & = & \overline{u_1}^0 \oplus \overline{u_2}^0 & \\
\overline{\mathsf{dec}(u_1, u_2)}^0 & = & \mathsf{dec}(u_1, \overline{u_2}^0) & \text{if } u_1 \in \mathcal{N} \text{ of type Cipher} \\
\overline{\mathsf{dec}(u_1, u_2)}^0 & = & \mathsf{dec}(1, \overline{u_2}^0) & \text{otherwise} \\
\overline{u}^0 & = & 0 & \text{otherwise}
\end{array}$$

The functions $\overline{\cdot}^0$ and $\overline{\cdot}$ are extended to sets of facts as expected. Moreover, the function $\overline{\cdot}^0$ is also defined on checks as follows:

$$\overline{\mathsf{chk_X}(t)}^0 = \mathsf{chk_X}(\overline{t}^0).$$

Proof. Consider a proof $F_1, \ldots, F_n$ of $S \vdash u$. We show by induction on n that we can construct sets $G_1, \ldots, G_p$ which only involve pure facts such that

- $G_1, \ldots, G_p$ is a proof of $S \vdash \overline{t}$ for any $t \in S \cup F_1 \cup \ldots \cup F_n$,
- $\overline{\mathsf{chkX}(t)}^0 \in \mathsf{SatChk}(S \cup G_1 \ldots \cup G_p)$ for any $\mathsf{chkX}(t) \in \mathsf{SatChk}(S \cup F_1 \cup \ldots \cup F_n)$.

This would conclude the proof since $u \in F_n$ and $\overline{u} = u$.

The base case $u \in S$ is trivial. For the induction step, we assume that there are sets of pure ground facts $G_1, \ldots G_p$ such that

- $G_1, \ldots, G_p$ is a proof of $S \vdash \overline{t}$ for any $t \in S \cup F_1 \cup \ldots \cup F_i$,
- $\overline{\mathsf{chkX}(t)}^0 \in \mathsf{SatChk}(S \cup G_1 \cup \ldots \cup G_p)$ for any check $\mathsf{chkX}(t) \in \mathsf{SatChk}(S \cup F_1 \cup \ldots \cup F_i)$.

and we show that we can construct a set of pure ground facts G_{p+1} such that

- $G_1, \ldots, G_{p+1}$ is a proof of $S \vdash \overline{t}$ for any $t \in S \cup F_1 \cup \ldots \cup F_{i+1}$,
- $\overline{\mathsf{chkX}(t)}^0 \in \mathsf{SatChk}(S \cup G_1 \cup \ldots \cup G_{p+1})$ for any $\mathsf{chkX}(t) \in \mathsf{SatChk}(S \cup F_1 \cup \ldots \cup F_{i+1})$.

The set of ground facts F_{i+1} is one-step deducible from $S \cup F_1 \cup \ldots F_i$, thus there exists a weakly well-adapted rule $\mathsf{R}_l \overset{\mathsf{new}\ \mathsf{n}}{\rightarrow} \mathsf{R}_r \in \mathcal{A}$ and a ground substitution θ such that $\mathsf{R}_l\theta \subseteq \mathsf{SatChk}(S \cup F_1 \cup \ldots \cup F_i)$ and $F_{i+1} = \mathsf{R}_r\theta$ (modulo AC). Let θ' be the substitution defined by

- $x\theta' = \overline{x\theta}^0$ for any $x \in dom(\theta)$ of type Base,
- $x\theta' = x\theta$ when $x\theta$ is a constant or a nonce of type Cipher and 1 otherwise.

We can show that $G_{p+1} = \mathsf{R}_r\theta'$ satisfies the required conditions. □

Key Part Import 2:

$$\begin{array}{rcl}
\mathsf{xk2},\ \mathsf{xtype} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype}) \oplus \mathsf{xk2}\}_{\mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xk2}),\ \mathsf{chkEven}(\mathsf{xtype}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype})) \\[1ex]
\mathsf{y},\ \mathsf{xk2} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n}) \oplus \mathsf{xk2}\}_{\mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xk2}) & & \mathsf{chkEven}(\mathsf{n}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n}))
\end{array}$$

Key Part Import 3:

$$\begin{array}{rcl}
\mathsf{xk3},\ \mathsf{xtype} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype}) \oplus \mathsf{xk3}\}_{\mathsf{km} \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xk3}),\ \mathsf{chkEven}(\mathsf{xtype}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype})) \\[1ex]
\mathsf{y},\ \mathsf{xk3} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n}) \oplus \mathsf{xk3}\}_{\mathsf{km} \oplus \mathsf{n}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xk3}) & & \mathsf{chkEven}(\mathsf{n}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{n}))
\end{array}$$

Key Import:

$$\begin{array}{rcl}
\mathsf{y},\ \mathsf{xtype} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{y}, \mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})\}_{\mathsf{km} \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xtype}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{imp})) \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})) & & \\[1ex]
\mathsf{xtype},\ \mathsf{z} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{n}, \mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})\}_{\mathsf{km} \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{imp})) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})) \\[1ex]
\mathsf{y},\ \mathsf{z} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{y}, \mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{n})\}_{\mathsf{km} \oplus \mathsf{n}},\ \mathsf{n} \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{imp})) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{n})) \\
 & & \mathsf{chkEven}(\mathsf{n})
\end{array}$$

Key Export:

$$\begin{array}{rcl}
\mathsf{y},\ \mathsf{xtype} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{xtype})\}_{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xtype}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{exp})) \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{xtype})) & & \\[1ex]
\mathsf{xtype},\ \mathsf{z} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{xtype})\}_{\mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{exp})) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{xtype})) \\[1ex]
\mathsf{y},\ \mathsf{z} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{n})\}_{\mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{n}},\ \mathsf{n} \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{z}, \mathsf{km} \oplus \mathsf{exp})) & & \mathsf{chkEven}(\mathsf{n}) \\
 & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y}, \mathsf{km} \oplus \mathsf{n}))
\end{array}$$

Encrypt Data: $\mathsf{x} \xrightarrow{\mathsf{new}\ \mathsf{n}} \{\mathsf{x}\}_{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{data})},\ \mathsf{n},\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{data}))$

Decrypt Data: $\mathsf{x} \xrightarrow{\mathsf{new}\ \mathsf{n}} \mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{data})),\ \mathsf{n}$

$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{data}))$$

Translate Key:

$$\begin{array}{rcl}
\mathsf{x},\ \mathsf{xtype},\ \mathsf{y2} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})\}_{\mathsf{dec}(\mathsf{y2}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y2}, \mathsf{km} \oplus \mathsf{exp})),\ \mathsf{chkEven}(\mathsf{xtype}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{imp})) \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})) & & \\[1ex]
\mathsf{x},\ \mathsf{xtype},\ \mathsf{y1} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})\}_{\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp})) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{exp})) \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})) & & \\[1ex]
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y2}, \mathsf{km} \oplus \mathsf{exp})),\ \mathsf{xtype},\ \mathsf{y1},\ \mathsf{y2} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{n}, \mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})\}_{\mathsf{dec}(\mathsf{y2}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{xtype}},\ \mathsf{n} \\
\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp})) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{xtype})) \\[1ex]
\mathsf{x},\ \mathsf{y1},\ \mathsf{y2} & \xrightarrow{\mathsf{new}\ \mathsf{n}} & \{\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{n})\}_{\mathsf{dec}(\mathsf{y2}, \mathsf{km} \oplus \mathsf{exp}) \oplus \mathsf{n}},\ \mathsf{n} \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y2}, \mathsf{km} \oplus \mathsf{exp})) & & \mathsf{chkEven}(\mathsf{n}),\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{x}, \mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp}) \oplus \mathsf{n})) \\
\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y1}, \mathsf{km} \oplus \mathsf{imp})) & &
\end{array}$$

Note that no key conjuring variant can be obtained from the *Key Part Import 1* rule.

Figure 3. Key Conjuring variants of the rules of the IBM CCA Key Management Transaction Set