Deduction with XOR Constraints in Security API Modelling

Citation for published version:
Steel, G 2005, Deduction with XOR Constraints in Security API Modelling. in R Nieuwenhuis (ed.), *Automated Deduction - CADE-20: 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005. Proceedings.* Lecture Notes in Computer Science, vol. 3632, Springer-Verlag GmbH, pp. 322-336. https://doi.org/10.1007/11532231_24

Digital Object Identifier (DOI):
10.1007/11532231_24

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
Automated Deduction - CADE-20

Deduction with XOR Constraints in Security API Modelling

Graham Steel

School of Informatics,
University of Edinburgh,
Edinburgh, EH8 9LE, Scotland.
graham.steel@ed.ac.uk
http://homepages.inf.ed.ac.uk/gsteel

Abstract. We introduce XOR constraints, and show how they enable a theorem prover to reason effectively about security critical subsystems which employ bitwise XOR. Our primary case study is the API of the IBM 4758 hardware security module. We also show how our technique can be applied to standard security protocols.

1 Introduction

The application program interfaces (APIs) of several secure cryptographic hardware modules, such as are used in cash machines and electronic payment devices, have been shown to have subtle flaws which could be used to mount financially lucrative attacks, [5, 9]. These flaws were, however, only discovered after laborious hand analysis. This suggests them as promising candidates for formal analysis in a theorem prover. The APIs can be thought of as defining a set of two-party protocols, and modelled in a similar way to security protocols, which have attracted a large amount of attention from formal methods researchers in recent years. However, one key difference is that the specifications of the APIs are at the concrete bit by bit level, rather than at the abstract level of public or symmetric key cryptography. It has been by exploiting bit-level operations in the APIs that attacks have been found. One example of such an operation is bitwise exclusive-or (XOR). In the IBM 4758 Common Cryptographic Architecture API, XOR is used extensively. In an attack discovered by Bond, [5], the self-inverse property of XOR can be exploited, together with some other coincidences in the API transaction set, to reveal a customer's PIN. However, the combinatorial possibilities caused by the associative, commutative and self-inverse properties of XOR pose a significant challenge to formal analysis. It this challenge we address in the work described in this paper. We introduce XOR constraints, which state that two terms must be equal modulo XOR. We describe first-order theorem proving in a calculus modified to use these constraints, and show how this allows models of such APIs to be reasoned with effectively.

In the rest of this paper, we will first, in §2, give some more background in the field of secure hardware API analysis. In §3, we describe our approach using

XOR constraints. Our results using this method are given in §4. We conclude in §5, with some evaluation and comparison to related work.

2 Background

Hardware security modules (HSMs) are widely used in security critical systems such as electronic payment and automated teller machine (ATM) networks. 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 to try to read the memory, it will auto-erase in a matter of nanoseconds. In a typical application, this tamper-proof memory will be used to store the master keys for the device. Further keys will then be stored on the hard drive of a computer connected to the HSM, encrypted under the HSM's master key. These keys can then only be used by sending them back into the HSM, together with some other data, under the application program interface (API). The API will typically consist of a several dozen key management and PIN processing commands, and their prescribed responses. Although carefully designed, various subtle flaws in these APIs have been found, for example by Bond, [5], and Clulow, [9].

One can think of the API as defining a number of two-party protocols between a user and the HSM. This suggests that we might profitably employ methods developed for the analysis of security protocols, e.g. [23, 18, 3]. However, there are some important differences in the scenario to be modelled. We are concerned only with two 'agents', the HSM itself and a malicious intruder. The HSM itself is (usually) 'stateless', in the sense that no information inside the HSM changes during transactions. This means we need only concern ourselves with the knowledge of the intruder, and not with any other information about the state of the protocol participants. Readers familiar with conventional security protocol analysis will appreciate that this constitutes a significant simplification of the general problem. However, there are other aspects which add complexity not normally considered in standard protocol analysis. In order to look for attacks, we must analyse the composition of several dozen short two-party protocols, that may be combined in any order. This is a significant increase in complexity compared to analysing key exchange protocols, which typically entail half a dozen messages at most. Additionally, it is not sufficient to model APIs in a 'perfect encryption' or free algebra model, where algebraic properties of the crypto functions are ignored. Most of the attacks that have been found exploit properties of these functions. The example we look at in this paper is the XOR function, which is widely used in the Common Cryptographic Architecture (CCA) API of the IBM 4758[1].

The attacks on the 4758 CCA were first discovered by Bond and Anderson, [5]. They exploit the way the 4758 constructs keys of different types. The CCA supports various key types, such as data keys, key encrypting keys, import keys

[1] http://www-3.ibm.com/security/cryptocards/pcicc.shtml

and export keys. Each type has an associated ‘control vector’ (a public value), which is XORed against the master key, km, to produce the required key type. So if $data$ is the control vector for data keys, then all data keys will be stored outside the 4758 box encrypted under $km \oplus data$[2]. Under the API, only particular types of keys will be accepted by the HSM for particular operations. For example, data keys may be used to encrypt given plaintext, but PIN derivation keys may not. Bond discovered an attack whereby the attacker changes the type of a key he has by exploiting the properties of XOR. In Bond’s attack, the API’s Key Part Import command is used to generate keys with a known difference. The Key Import command can then be used to change a key of one type to any other type. This allows a PIN derivation key to be converted to a data key, and then used to encrypt data, which is a critical flaw: a customer’s PIN is just his account number encrypted under a PIN derivation key. So, the attack allows a criminal to generate a PIN for any card number. Recent versions of the CCA manual (2.41 onwards) give procedural restrictions on the key part import commands to prevent the attack.

After discovering the attack, Bond made an effort to formalise the problem and rediscover the attack, using the theorem prover SPASS, [26]. He produced a first-order model with one predicate, P, to indicate that a term was known outside the box (and therefore available to an intruder). The XOR operation was modelled by equational axioms giving its associative, commutative and self-inverse properties. However, SPASS was unable to rediscover the attack, even after days of run time, and with only the three relevant commands from the API included in the model. Bond was able to prove that the attack was in the search space modelled, by inserting various hints and proving intermediate conjectures, but the combinatorial blow-up caused by the AC and self-inverse properties, and the fact that the spy can XOR together any two terms he knows to create a new term, prevented a genuine rediscovery.

Ganapathy et al. have recently attacked the same rediscovery problem using bounded model checking, [12]. They successfully rediscovered the part of the attack where a key’s type is changed. However, they included only two of the APIs commands in their model, so there was very little search involved. Again, associativity, commutativity and self-inverse were modelled by explicit axioms in the model.

The aim of the work described in this paper was to develop a framework for handling XOR that allows the whole of an API to be modelled and reasoned with effectively. Our approach involved adapting a theorem prover supporting AC unification (daTac, [25]) to handle constraints specifying equality between terms modulo XOR. Constraints have been used before in first-order theorem proving, for a variety of purposes. They provide a natural way of implementing ordering restrictions, [20], and the basicness restriction, [2], handling AC unification, [21, 25], and incorporating the axioms of Abelian groups, [13]. As has been remarked e.g. in [22], they are an attractive solution for bringing domain knowledge into deduction systems, because they provide a clean separation between

[2] We use $\oplus$ as the infix XOR operator.

the general purpose logic, and the special purpose constraints. Our results using the XOR constraint framework include a rediscovery of Bond's attack in a model formalising the whole of the CCA key-management transaction set. Additionally, we rediscovered a second, stronger, and more complex attack which, although already discovered by Bond, had not been previously modelled. We have also used our XOR constraint technique to rediscover an attack on a variant of the Needham-Schroeder-Lowe protocol, using XOR, given in [7]. As far as we are aware, this attack had also not been rediscovered automatically before.

3 XOR Constraints

We will first explain how our work on API analysis suggested the development of XOR constraints. We then define deduction, redundancy checking and simplification with XOR constraints. Finally, we discuss some implementation details.

3.1 The Need for XOR Constraints

The attacks we are concerned with on the 4758 CCA all employ the Key Import command. This command is designed to be used, for example, to import a key from another 4758 unit. Keys are transported between 4758s under 'key encrypting keys', or *KEK*s. Here is the command as a two-message protocol:

Key Import:

1. User $\rightarrow$ HSM : $\{\!|\ \text{KEY1}\ |\!\}_{\text{KEK}\oplus\text{TYPE}}$, TYPE, $\{\!|\ \text{KEK}\ |\!\}_{\text{km}\oplus\text{imp}}$
2. HSM $\rightarrow$ User : $\{\!|\ \text{KEY1}\ |\!\}_{\text{km}\oplus\text{TYPE}}$

We write the master key *km* and the import control vector *imp* in lowercase to show these are values specific to a particular 4758, and other values in caps to show these are in effect variables; for these, any values (modulo a certain amount of error checking) can be used. In the above protocol, the HSM does not know in advance the key required to decrypt the first packet, $\{\!|\ \text{KEY1}\ |\!\}_{\text{KEK}\oplus\text{TYPE}}$. It must first decrypt the final packet to obtain *KEK*, and then XOR this against *TYPE*, which is given in the clear. The result can then be used as a key to decrypt the first packet. It is this XORing together of parts that is exploited in Bond's attacks, the first variant of which runs like this, [4, §7.3.4]:

> Suppose some corrupt bank insider has seen $\{\!|\ \text{pp}\ |\!\}_{\text{kek}\oplus\text{pin}}$, a PIN derivation key (*pp*), encrypted under some key-encrypting key, *kek*, XORed against the appropriate control vector, *pin*. This is the form in which it would be sent to a 4758 in a bank. Additionally, we suppose that an attacker has access to, and can tamper with, a 'key part' for the key. Keys are divided into parts to allow two (or more) separate people to physically transfer key information to another 4758 module. The idea of dividing the key into parts is that attacks requiring collusion between multiple employees are considered infeasible. The key parts for the 4758 are combined using XOR. Because of the properties of XOR, no single

officer, in possession of a single key part, has any information about the value of the final key. Here is the sequence of commands required to import a two-part key $k1 = k1a \oplus k1b$:

Key Part Import(1):

1. Host $\rightarrow$ HSM : k1, TYPE
2. HSM $\rightarrow$ Host : $\{\!|\ \mathrm{k1a}\ |\!\}_{\mathrm{km}\oplus\mathrm{kp}\oplus\mathrm{TYPE}}$

Key Part Import(2):

1. Host $\rightarrow$ HSM : $\{\!|\ \mathrm{k1a}\ |\!\}_{\mathrm{km}\oplus\mathrm{kp}\oplus\mathrm{TYPE}}$, k1b, TYPE
2. HSM $\rightarrow$ Host : $\{\!|\ \mathrm{k1a} \oplus \mathrm{k1b}|\!\}_{\mathrm{km}\oplus\mathrm{TYPE}}$

The *kp* control vector indicates that a key is still only partially complete. For Bond's attack, we suppose the attacker is a single corrupt insider responsible for adding the final key part. So, he has a partial key which looks like $\{\!|\ \mathrm{kek} \oplus \mathrm{k2}\ |\!\}_{\mathrm{km}\oplus\mathrm{imp}\oplus\mathrm{kp}}$, where *k2* is his own key part. Before addng his own key part, he XORs it against *data* and *pin*. The final key part import command will then yield $\{\!|\ \mathrm{kek} \oplus \mathrm{data} \oplus \mathrm{pin}|\!\}_{\mathrm{km}\oplus\mathrm{imp}}$, which he can use in the Key Import command in the following way:

1. User $\rightarrow$ HSM : $\{\!|\ \mathrm{pp}\ |\!\}_{kek\oplus pin}$, data, $\{\!|\ \mathrm{kek} \oplus \mathrm{data} \oplus \mathrm{pin}\ |\!\}_{km\oplus imp}$
2. HSM $\rightarrow$ User : $\{\!|\ \mathrm{pp}\ |\!\}_{km\oplus data}$

On receiving message 1, the HSM forms $kek \oplus data \oplus pin \oplus data$, but with the self-inverse cancellation of bitwise XOR, this is the same as $kek \oplus pin$. So the encrypted key *pp* is output successfully, but instead of being under its correct control vector, as a PIN derivation key, it is now returned as a data key, allowing the attacker to generate customer's PINs using the Encrypt Data command[3]:

1. User $\rightarrow$ HSM : $\{\!|\ \mathrm{pp}\ |\!\}_{km\oplus data}$, PAN
2. HSM $\rightarrow$ User : $\{\!|\ \mathrm{PAN}\ |\!\}_{pp}$

In order to capture these kinds of attacks, we must model the command in the way it is implemented, i.e. we must model the HSM constructing the decryption key for packet 1, $KEK \oplus TYPE$, from the other two packets. This seemingly requires us to abandon the so-called 'implicit decryption' assumption usually used in modelling protocols, [19]. Instead, we must model the decryption of packet 1 explicitly, like this[4]:

$$
\begin{array}{l}
P(crypt(xkek1 \oplus xtype1, xk1)) \wedge \\
P(xtype2) \wedge P(crypt(km \oplus imp, xkek2)) \\
\rightarrow P(crypt(km \oplus xtype2, \\
\quad decrypt(xkek2 \oplus xtype2, \\
\quad\quad crypt(xkek1 \oplus xtype1, xk1))))
\end{array}
$$

[3] PAN stands for 'personal account number'. Recall that a PIN is just a customer's account number encrypted under a PIN derivation key.

[4] We follow Bond's original model, with the *P* predicate signifying a term known outside the HSM.

and introduce the cancellation rule:

$$\rightarrow \mathit{decrypt}(x1, \mathit{crypt}(x1, x2)) = x2$$

This brings the attacks into the search space, but only allows them to be found by forward search through the model. This is because the result of the key import command will not unify with the result required in the attack (which is $P(\mathit{crypt}(\mathit{km} * \mathit{data}, \mathit{pp})))$ until after the equations specifying the self-cancelling properties of XOR, and the encrypt/decrypt cancellation rule above, have been applied. To allow this to happen in backwards search, we would have to allow the cancellation rules to be applied in reverse, which would cause an enormous blow-up in the search space. When searching forwards, the branching rate is still very large, since for any terms x and y, if we have $P(x)$ and $P(y)$, we can always combine these with XOR to produce $P(x \oplus y)$. We have to search a large space of these possible combinations in order to derive the terms required to make attacks.

Our solution is to introduce XOR constraints to state that in the key import command, the key used to encrypt the first packet, and the key derived by the HSM from the other packets, must be equal modulo XOR. This allows us to use implicit encryption. The clause to model the command now looks like this:

$$\begin{array}{l} P(\mathit{crypt}(x1, \mathit{xk}1)), P(\mathit{xtype}), P(\mathit{crypt}(\mathit{km} \oplus \mathit{imp}, \mathit{xkek}2)) \\ \rightarrow P(\mathit{crypt}(\mathit{km} \oplus \mathit{xtype}, \mathit{xk}1)) \\ \text{IF } \mathit{xkek}2 \oplus \mathit{xtype} =_{XOR} x1 \end{array}$$

Semantically, the constraints are a restriction on the universally quantified variables in the clauses. Rather than generating instantiations to solve the constraints directly, we allow the search to proceed normally, but check after each deduction that the constraints remain soluble. Note that the output of the new form of the command will now unify with terms in a backwards search from the goal. To encourage this, we employ the 'basic strategy', [2], which seems highly effective for these problems.

3.2 Deduction with XOR Constraints

We employ the constrained resolution/paramodulation calculus of [25], with the addition of XOR constraints. The XOR constraints of two resolving clauses are combined in the resolvent using logical AND, just like the pre-existing ordering constraints and substitution constraints. When generating new inferences we apply the substitution required for the inference to the constraints before checking for solubility. Because of the self-cancelling property of XOR, this amounts to checking whether the constraint is ground and has been solved, or whether there are still variables at positions which could solve the constraint. More formally, solubility of an XOR constraint $s_1 \oplus \ldots \oplus s_m =_{XOR} t_1 \oplus \ldots \oplus t_n$ is checked like this:

1. If any of $s_1, \ldots, s_m, t_1, \ldots, t_n$ contain variables, the constraint is regarded as soluble.
2. If all $s_1, \ldots, s_m, t_1, \ldots, t_n$ are ground, then first discard zeros, and then count up the number of occurrences of each term in the set $\{s_1, \ldots, s_m, t_1, \ldots, t_n\}$. If all terms occur an even number of times, the constraint is soluble. If not, it is insoluble.

Only inferences producing clauses with soluble constraints are permitted. These simple syntactic checks can be made very quickly. Note that condition 1 may allow us to keep some clauses with constraints which cannot be solved. We do not attempt to check that, in the theory in question, the variable may eventually be instantiated to a value which satisfies the constraint. However, we will certainly only discard genuinely insoluble constraints, thereby preserving completeness. We can eliminate some further insoluble sets of XOR constraints by a simple pairwise check for inconsistency. If two constraints, both attached to the same clause, equate the same variables to different ground terms, then they are inconsistent and the clause can be pruned away. Again, this check may allow some insoluble constraints through, but it is quick to perform, preserves completeness and seems useful in practice.

3.3 Subsumption with XOR Constraints

Simplification and checking for redundancy are of paramount importance in practical theorem proving. Modifications to the resolution/paramodulation calculus which prevent the use of subsumption checking rules are usually useless in practice, whatever their apparent theoretical advantages. The use of XOR constraints allows subsumption checking, though with the following restriction: the solutions of the constraints in a clause that is subsumed must be a subset of the solutions of the clause we retain. For XOR constraints, this occurs just when:

1. The more general clause has no XOR constraint, or
2. The XOR constraints of the subsumed clause and the subsuming clause are identical (modulo AC), *after* any substitution required to make the subsumption has been applied to the XOR constraint.

This is similar to the standard rule for subsumption in the constrained calculus, [25, §5.1]. To see why condition 2 is not only sufficient but necessary to ensure that all solutions of some constraint, T_1, are solutions of another, T_2, the reader is invited to write down two identical constraints, and then add a single variable or ground term to either. Immediately it is possible to construct solutions of the first which are not solutions of the second, and vice versa.

In practice, many checks for subsumption are ruled out because of incompatible XOR constraints. Fortunately, the check can be made quite quickly (see §3.5, below).

3.4 Simplification with XOR Constraints

Reduction of newly produced clauses by demodulation, clausal simplification, etc. is also permissible for XOR-constrained clauses. In our implementation, we only allow demodulation by clauses with no XOR constraint, since in our experiments, it seems to be only these clauses we would like to use.

3.5 Implementation

Our prototype implementation is in daTac, [25], which uses the constrained basic calculus we require. It also supports AC-unification, implemented via constraints. We store XOR constraints in a normal form, $v_1 \oplus \ldots \oplus v_m =_{XOR} t_1 \oplus \ldots \oplus t_n$, where the v_i are pure variables, and the t_j are atoms or complex terms. After an inference, we apply the required substitution to the XOR constraint, and then re-normalise it, by first moving any non-variables from the v_i to the right hand side of the equality, and then by removing from the t_j any occurrences of the identity element for XOR, and any pairs of identical ground terms. We also order the v_i and t_j according to some arbitrary total ordering. After the simplification, our checks for solubility described in §3.2 can be readily performed. Additionally, keeping the constraints in a normal form speeds up the check for identical constraints required in §3.3.

4 Results

We present first our results on the command set of the 4758 API. Then, we present results from experiments using XOR constraints in the modelling of a standard key-exchange protocol, which has an attack that can only be found when the properties of XOR are taken into account. All the model files and results are available via http at `http://homepages.inf.ed.ac.uk/gsteel/xor/`

4.1 The 4758 CCA Attacks

Our model for the first variant of Bond's attack includes all the symmetric key management commands in the 4758 CCA API, except those which do not output keys (MAC Generate, MAC Verify and Key Test). We employ XOR constraints to model any command in which the HSM must decrypt a packet using a key formed by XOR. In the 4758 CCA, the Key Import command and the Key Translate command require XOR constraints. The XOR operation is modelled by an operator (*) which is declared to be AC. We add the following two equations to the model to define an identity element for the XOR operation:

$$x1 * id = x1$$
$$x1 * x1 = id$$

These properties of the XOR function are also implicitly modelled by the way we check the solubility of XOR constraints, as described in §3.2. However, the XOR constraints will only account for these properties when the XOR cancellation happens during a command being executed by the HSM. We have to add these equations to allow for the possibility of the attacker doing his own manipulations outside the HSM.

The intruder in our model is given the initial knowledge specified by Bond (see §3.1), along with some other public terms such as the values of all the control vectors. We chose a precedence ordering which set the P predicate largest, the XOR operator smallest, and put the key and control vector identifiers in a set of symbols of equal precedence in between. The conjecture is that a term of the form $\{\!|\ \mathrm{PAN}|\!\}_{\mathrm{PDK}}$, i.e. a PIN, may become public. daTac finds the attack in just under 1 second[5], generating 162 new clauses for a model with 21 initial clauses.

We also modelled the second, stronger variant of Bond's attack, the key import/export attack, described in [4, §7.3.5]. This attack does not require the intruder to have seen a PIN derivation key being sent to the 4758 under attack, nor does it require him to have access to a key part for that PDK. Instead, he converts a PDK already imported for use on the box into a data key by first creating a pair of import/export keys with a known difference. The cleartext value of these keys is unknown – they are generated by a process known as 'key conjuring', [4, §7.2.3], where random values are tried until one is accepted by the HSM. We approximate this by explicitly adding the conjured parts to the model. The idea is to export the PDK under the exporter key, then to re-import it under the importer key to turn it into a data key, using the same XOR trick as in the previous attack. In fact, in this attack, the 'trick' must be used twice: once to create the related import export pair from conjured key parts, and once to use it to change the type of the PDK. In our model for this experiment, in addition to the conjured key parts, we include in the intruder's initial knowledge a PIN derivation key encrypted under $km \oplus pin$ (instead of $kek \oplus pin$, as before). The attack is found after 1.47 seconds, and the generation of 601 clauses in a model with 23 initial clauses.

As a further experiment, we modelled the attack exhibited by IBM engineers in their response to Bond's discovery, described in [9, p. 54]. This attack requires that we take into account the fact that the true value of the 'data' control vector is 0. This we model with the additional rule:

$$P(crypt(xk * data, x)) \rightarrow P(crypt(xk, x))$$

1 The attack is quite simple, and is found in 0.2 seconds after generating 229 clauses from an initial theory of 21 clauses.

Together, these results represent a qualitative improvement over the previous efforts to reason with XOR described in §2. We have found stronger, and more complex attacks on the whole command set, while Bond's original model, and the Ganapathy et al. model, both required hints and/or simplifications to discover

[5] Running under Linux 2.4.20 on a 2GHz Pentium 4 machine.

even part of the original, shorter attack. We believe it is the XOR constraints that make the difference, and to investigate this, we tried a version of the model without them, modelling the key import command with explicit encryption and decryption, as described in §3.1. If we model only the commands required for the attack, daTac finds the attack after about 5 minutes, generating more than 10 000 clauses. If the whole command set is used, the attack is not found after more than 24 hours run-time. This supports the hypothesis that it is the XOR constraints that are making the difference.

4.2 Attacking a Variant of the Needham–Schroeder–Lowe Protocol

In [7], Chevalier et al. exhibit a variant of the well known Needham–Schroeder–Lowe protocol, [17], which uses XOR to bind Bob's identifier to Alice's nonce in message 2. They show that this protocol can be attacked, but only when the AC and self-inverse properties of XOR are taken into account. We were keen to see if our XOR constraints could be more generally applied to security problems, so we formalised this protocol. Our model for the protocol is based on the first-order model devised by Jacquemard, Rusinowitch and Vigneron, [14]. They also used daTac, since the model uses AC unification to deal with the set of intruder knowledge, and to model the set of states of agents. The protocol in question runs like this:

1. $A \rightarrow B : \{\!|\, N_A, A |\!\}_{pubK_B}$
2. $B \rightarrow A : \{\!|\, N_A \oplus B, N_B |\!\}_{pubK_A}$
3. $A \rightarrow B : \{\!|\, N_B |\!\}_{pubK_B}$

To attack the protocol, we assume Alice starts a session with our intruder, I, who is accepted as an honest agent. He forwards on the nonce and identifier to the victim, Bob, *after* XORing the nonce against B and I. He can now carry out Lowe's classic man-in-the middle attack:

$$\begin{array}{lll}
1. & A \rightarrow I & : \{\!|\, N_A, A |\!\}_{pubK_I} \\
1'. & I_A \rightarrow B & : \{\!|\, N_A \oplus B \oplus I, A |\!\}_{pubK_B} \\
2'. & B \rightarrow I_A & : \{\!|\, N_A \oplus B \oplus I \oplus B, N_B |\!\}_{pubK_A} \\
2. & I \rightarrow A & : \{\!|\, N_A \oplus B \oplus I \oplus B, N_B |\!\}_{pubK_A} \\
3. & A \rightarrow I & : \{\!|\, N_B |\!\}_{pubK_I} \\
3.' & I_A \rightarrow B & : \{\!|\, N_B |\!\}_{pubK_B}
\end{array}$$

In our model, we use an XOR constraint to model the fact that A will accept message 2 from B only if, after XORing it against the identifier for B, it is equal to the nonce she sent to B in message 1. In the Jacquemard et. al model, agents move into a state of waiting for an appropriate response at the same time as they send a message. So in this protocol, a single clause models A sending a message 1 containing nonce N_A, and moving into a state of waiting for a response containing $N_A \oplus B$. We simply change the clause so that A is waiting for a response containing some fresh variable X, and add the constraint

$X \oplus B =_{XOR} N_A$. We also allow the intruder to XOR arbitrary terms he knows against each other to generate new ones, and allow him to send these terms in messages.

These modifications bring the attack into the search space, but we are faced with the usual XOR problem: there are simply too many ways the spy can combine terms together. However, we can fix this problem using the same trick Jacquemard et al. used for modelling intruder knowledge: use AC unification. In all protocols, it will only pay the intruder to consider the self-inverse properties of XOR when he is trying to fool other agents. This we model with XOR constraints, which take into account self-inverse. Now XOR is just an AC combinator, equivalent to the set combinator used in the model to collect together terms. When faking a message, the spy simply picks a term from his knowledge by unification, and we allow the XOR constraints mechanism to check whether this term is satisfactory when the combinator is considered to be XOR.

Using this model, daTac finds the attack presented in [7] in 12.35 seconds, deriving 1652 clauses. This is in the middle of the range of times reported in [8], where daTac was used to find flaws in a number of protocols in a free algebra model. One should bear in mind that the first-order model had by then evolved a little from the one reported in [14], and hardware used was probably slower (details are not given in the paper). Even so, we have succeeded in automatically discovering a protocol attack in a model using XOR. We compare this to related work in the next section.

5 Conclusions

In the experiments we have performed, the use of XOR constraints has greatly improved the performance of the theorem prover on the problems examined. In the analysis of the 4758 CCA API, the greatest benefit seems to come from the fact that XOR constraints allow us to use a goal-directed search, via the basic strategy, in a model where explicit encryption and decryption would otherwise prevent this. Experiments with using a positive strategy on the same models yielded no proofs after 24 hours run-time. For the protocol considered in §4.2, the benefit is that by considering the self-inverse properties of XOR only in the constraints, we can treat XOR as a standard AC combinator, and so use AC unification to build the terms we need to satisfy the constraints. Again, running daTac on a model where XOR terms must be built up explicitly yields no proofs after hours of run-time.

In the early nineties, Longley and Rigby were the first to look at the problem of automatically analysing the key-management schemes presented by HSM APIs, [16]. They searched for attacks by querying a PROLOG database of API rules with terms that an attacker might try to obtain. They did not consider any algebraic properties of the cryptofunctions used, however. In §2, we mentioned more recent related work by Ganapathy et al., which did consider the properties of XOR. In §4.1, we showed how our results have improved on theirs, by allowing attack discovery even when the whole symmetric-key management command set is modelled, and by discovering the more complex key import/export attack, which had not been modelled before.

In the field of security protocol analysis, several researchers have recently started to consider algebraic properties of crypto functions, such as XOR. The decidability of insecurity for protocols using XOR, provided the number of sessions is bounded, has been shown in [7] and [10]. Their decision procedures for insecurity are, however, purely theoretical. We can relate decidability of insecurity for APIs, specified in the way we have shown in this paper, to these decidability results: first, observe that if we bind the number of times each API command can be executed, there are a bound number of ways these commands can be chosen and ordered. Consider each command as a two message protocol, as we have suggested in this paper. Consider each choice and ordering of the commands as a complete protocol. Then a bound API specifies a set of bound protocols. We can now augment an NP decision procedure for XOR protocols by first guessing a protocol from our set. Unfortunately, this is not quite enough to settle the question of decidability, since we also have to change the formalism for protocols to allow for steps like the Key Import command, where a key used for decryption must be generated using XOR in the same step. Delaune and Jacquemard have proposed a decidability procedure for protocols with explicit encryption and decryption, which we could adapt to such API commands, [11]. Unfortunately, they have so far been unable to show whether the XOR operation can be covered by their results. We suspect that one of these works could indeed be adapted to show decidability of insecurity for bound API transaction sets. However, for the moment, it remains an open question.

In further work, we plan to improve the performance of the prover when handling XOR constraints. We have observed that in our proof searches, often very early a clause is derived containing just $P(x_1) \wedge \ldots \wedge P(x_m) \Rightarrow$, and an XOR constraint $x_1 \oplus \ldots \oplus x_m =_{XOR} t_1 \oplus \ldots \oplus t_n$, with all the terms t_i needed to solve the constraint available in the domain of P. However, it still takes a long time to finally resolve this to the empty clause, since there are still many ways the XOR combination rule, $P(x) \wedge P(y) \Rightarrow P(x \oplus y)$, can be applied to the clause. We discovered this by using *viz*, our tool for visualising first-order proof search, [24]. The output from *viz* for the first 4758 attack is given in Figure 1. Note that *viz* produces output in scalable vector graphics (SVG) format, which is designed to be viewed interactively in a viewer supporting zooming and panning. However, for small proofs, even a straight printout of the search space can be revealing, as in this case. The nodes on the graph are clauses, and the edges mark logical dependency. Nodes on the 'critical path', i.e. ones used in the proof, are diamond-shaped. The nodes are coloured to indicate when they were generated in the search. Darker nodes were generated first, and lighter nodes generated later. The circle we have added to the diagram surrounds Clause 50, a clause of the form we have just described, which appears like this in the daTac output:

```
Clause 50: P(x1), P(x2) =>  \
      IF   x1 * x2 =x data * pin * k2
```

Clause 50 is eventually solved to give the proof, but we can see from Figure 1 that most of the work in the proof search is just to find the terms requires to solve Clause 50's XOR constraint (this is what is happening in the large collection of nodes in the bottom portion of the diagram). Finding the combination of API commands required to effect the attack takes a relatively small amount of search (in the top of the diagram). To address this problem, we could perhaps employ our AC unification trick again somehow, or use a specialised 'terminator' tactic, [1], to try to finish off these clauses.

There are many other electronic payment and banking APIs waiting to be formalised. They pose more challenges, such as reasoning about the relative complexity of breaking different crypto functions. We will be continuing our research in this area.

Acknowledgements

The API of the 4758 CCA is specified in a 500 page manual. Mike Bond's unpublished 4 page 'summary sheet' for the key management transaction set was therefore invaluable, and we are grateful to him for providing us with a copy. We are also grateful to Laurent Vigneron for providing the source code for daTac, and to the anonymous referees for their considered criticisms and suggestions for improvements.

References

1. G. Antoniou and H. J. Ohlbach. TERMINATOR. In *International Joint Conference on Artificial Intelligence*, pages 916–919, 1983.
2. L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *11th Conference on Automated Deduction*, number 607 in LNCS, pages 462–476, 1992.
3. D. Basin, S. Mödersheim, and L. Viganò. An on-the-fly model-checker for security protocol analysis. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, pages 253–270, 2003. Extended version available as Technical Report 404, ETH Zurich.
4. M. Bond. *Understanding Security APIs.* PhD thesis, University of Cambridge, 2004.
5. M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, pages 67–75, October 2001.
6. A. Bundy, editor. *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*. Springer, 1994.
7. Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An NP decision procedure for protocol insecurity with XOR. In Kolaitis [15], pages 261–270.
8. Y. Chevalier and L. Vigneron. Automated unbounded verification of security protocols. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification, 14th International Conference*, volume 2404 of *Lecture Notes in Computer Science*, pages 324–337, Copenhagen, Denmark, July 2002. Springer.

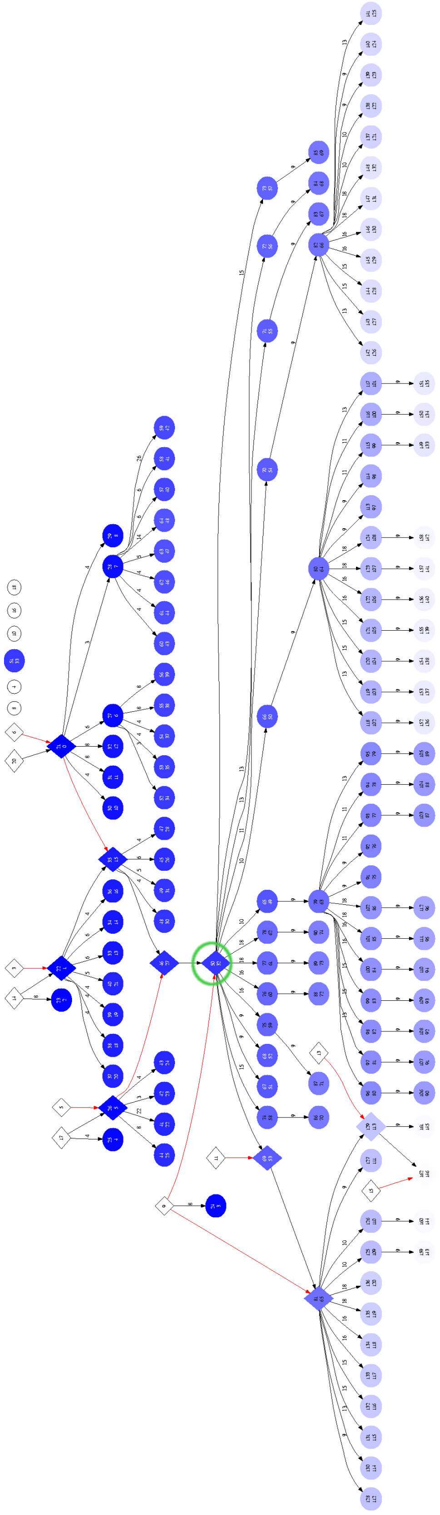

Fig. 1. The search for the first 4758 attack

9. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
10. H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In Kolaitis [15], pages 271–281.
11. S. Delaune and F. Jacquemard. A decision procedure for the verification of security protocols with explicit destructors. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 278–287. ACM Press, 2004.
12. V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of API-level exploits. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 312–321, New York, NY, USA, May 2005. ACM Press.
13. G. Godoy and R. Nieuwenhuis. Superposition with completely built-in abelian groups. *J. Symb. Comput.*, 37(1):1–33, 2004.
14. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 131–160. Springer, 2000.
15. P. G. Kolaitis, editor. *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings.* IEEE Computer Society, 2003.
16. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.
17. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.
18. G. Lowe. Breaking and fixing the Needham Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.
19. J. K. Millen. On the freedom of decryption. *Inf. Process. Lett.*, 86(6):329–333, 2003.
20. R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *11th Conference on Automated Deduction*, number 607 in LNCS, pages 477–491, 1992.
21. R. Nieuwenhuis and A. Rubio. AC-superposition with constraints: No AC-unifiers needed. In Bundy [6], pages 545–559.
22. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
23. L.C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.
24. G. Steel. Visualising first-order proof search. In *Workshop on User Interfaces for Theorem Provers (UITP '05)*, pages 179–189, Edinburgh, Scotland, April 2005.
25. L. Vigneron. Associative-commutative deduction with constraints. In Bundy [6], pages 530–544.
26. C. Weidenbach et al. System description: SPASS version 1.0.0. In H. Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 378–382, Trento, Italy, July 1999. Springer-Verlag.