Technical Report

UCAM-CL-TR-644
ISSN 1476-2986

Number 644

Robbing the bank with a theorem prover

Paul Youn, Ben Adida, Mike Bond,
Jolyon Clulow, Jonathan Herzog,
Amerson Lin, Ronald L. Rivest, Ross Anderson

August 2005

ISSN 1476-2986

Robbing the Bank with a Theorem Prover

Paul Youn Ben Adida Mike Bond Jolyon Clulow Jonathan Herzog
Amerson Lin Ronald L. Rivest Ross Anderson

Abstract

We present the first methodology for analysis and automated detection of attacks on security application programming interfaces (security APIs) – the interfaces to hardware cryptographic services used by developers of critical security systems, such as banking applications. Taking a cue from previous work on the formal analysis of security protocols, we model APIs purely according to specifications, under the assumption of ideal encryption primitives. We use a theorem prover tool and adapt it to the security API context. We develop specific formalization and automation techniques that allow us to fully harness the power of a theorem prover. We show how, using these techniques, we were able to automatically re-discover all of the pure API attacks originally documented by Bond and Anderson against banking payment networks, since their discovery of this type of attack in 2000. We conclude with a note of encouragement: the complexity and unintuiveness of the modelled attacks make a very strong case for continued focus on automated formal analysis of cryptographic APIs.

1 Introduction

1.1 Security APIs and Attacks

A *security API* is an Application Programming Interface (API) that uses cryptography to provide specific security functionality and implements a policy that limits the interactions a user may perform. A *conventional* API exposes a set of calls to a hardware or software module to package functionality, and enable a form of abstraction: changes to the implementation remain localized if they do not alter the specified API behavior. This opaqueness encourages programmers to write modular code.

Just as conventional APIs give services, but abstract away the low-level implementation, security APIs give as much as they take away: certain features are offered, while others are purposely missing in order to guarantee well-defined security goals. For example, an SSL accelerator card might have an API that offers key-pair generation, fast encryption and decryption, and public key extraction; but it will never allow private key extraction.

Conventional APIs are ubiquitous: they include the Microsoft Win32 API [20] for Windows client programming and the Posix Interface [26] for Unix programming. Popular security APIs include RSA's PKCS#11 and Microsoft Cryptographic API (CAPI).

A *security API attack* is a breach of a target API's stated security goals that respects the API boundary. It is thus a *legal* sequence of calls, but which are chained in an unexpected way. The potential attack paths in an average API are numerous: usually, most API calls are available for invocation at each step in an attack search.

Our goal is the formal analysis of these security APIs and the detection of attacks against them. In this paper, we describe a new methodology for the specification of security APIs in the language of first-order logic, the input language of many theorem provers. This new methodology allows the application of many powerful search strategies. We developed our methodology during the analysis of APIs for financial ATM network processing, and we will use a particular API (the IBM 4758 CCA) as a running example. We first briefly describe the CCA API, and then summarize our modeling techniques.

1.2 Target APIs and their Vulnerabilities

The Common Cryptographic Architecture (CCA) is the security API for the IBM 4758, a hardware-based tamper-resistant device that enables secure banking operations. This cryptographic module is used in *Automatic Teller Machine (ATM)* networks for two important tasks: securing communications links in the international ATM network, and for the generation and verification of cusomter *Personal Identification Numbers (PINs)*.

The architecture of the CCA achieves these tasks through applying common cryptographic primitives and API concepts, which we describe briefly here:

- Fundamental commands are provided to generate, process and verify PINs. They work with clear user inputs, such as the customer account number, encrypted inputs, such as the trial PIN, and long-term secret keys.
- A *unique master key* is held in a special secure register in the device, and used as the root of a hierarchy for storage of keys.
- *Key wrapping* is used to increase available storage, and for secure communication between devices. For instance, a key can be "wrapped" by encryption under another *transport key*, storing all the metadata associated with the key alongside the value. When the CCA needs to work with a key k, the user must provide k wrapped under the master key (i.e., encrypted as $\{k\}_{KM}$), from some external storage, assumed to be insecure. The CCA implements policy on key usage: it will not simply unwrap the key and reveal it in the clear. Instead, it acts as a gate-keeper to the wrapped key: all operations with that key are performed by API requests, and all underlying computation is performed securely inside the hardware security module.
- Simple *Secret Sharing* [24] is used to bootstrap transmission of sensitive data from one CCA module to another. A top-level key-encrypting-key, KEK, is split into multiple shares (called *key parts*) and each share is transmitted independently. The parts are recombined using special CCA API calls. Both modules need to share at least one top-level transport key. Once received, each end stores KEK locally in external memory, wrapped under that device's master key. The complete KEK is never available in the clear outside the CCA's secure boundary: as soon as an operator imports the first KEK share, the resulting partial key is wrapped under the CCA's master key as usual. The second and third shares are added in similar ways, with the prior wrapped partial key as input and the newly recombined and wrapped key as output.
- *Key typing* provides a mechanism for specifying a key's precise usage domain. A PIN key, for example, can be used within a transaction to verify a customer PIN for a transaction. However, a key designed for this purpose should not be usable for generic encryption and decryption of data, otherwise the attacker could generate or verify PINs at will by issuing a simple decrypt call. The CCA API is designed to prevent this (among other things). Keys with distinctly different types can be used for other purposes: $DATA$ for communications, and *importer* and *exporter* keys for the wrapping of other keys.

1.3 Our Contribution

Our main contribution to the analysis of security APIs is three-fold:

1. We provide what we believe to be the first application of formal automated reasoning techniques to the problem of API security.

2. We define a modeling methodology which greatly speeds the analysis of APIs, thus making such analysis practical.

3. We reproduce, using our automated methods, all previously-known key-management attacks on the 4758, demonstrating the ability of the tool to reason at least as well as a skilled human (assuming the human can provide a correct API specification).

Modeling & Special Techniques Our methodology models security APIs with a theorem prover. Specifically, we use OTTER [17], an automated reasoning tool based on first-order logic. At the most basic level, the model proceeds as follows:

- User knowledge of a message or value is represented via a predicate which is true only for known values. That is, user knowledge of a value `x` is represented via the truth of predicate `U(x)`.
- Compound terms are represented by functions. For example, the encryption of `x` with key `y` is represented by `e(x,y)`, while the XOR of these two values is represented as `xor(x,y)`.
- Initial user knowledge is represented as axioms stating this predicate to be true for given messages. For example, the user may know the *primary account number* (`PAN`) of a given customer. Thus, the model includes `U(PAN)` as an axiom.
- The user may be able to make basic deductions without the use of the API. For example, the user can XOR together two known values. Therefore, the model includes these possible deductions as inference rules.
- The user can also make function calls to the API, therefore the function calls are modeled as inference rules that map the input to the output.
- Properties of the underlying cryptographic primitives can be represented via equations. For example one of the properties of the XOR operation can be represented as `xor(x,y) = xor(y,x)`.
- Lastly, we ask the tool to find attacks by asking it to violate the security goals of the API. That is, we ask it to prove that the security goals are false. If we wish to find an attack that recovers a user's PIN, we might provide as a goal `-U(PIN)`. [1]

This approach is not especially novel. Unfortunately, it is not especially fast, either. Analysis of this simple specification will prove to be intractable, for two basic reasons: Firstly, the inference rules will allow the user to deduce an infinite number of terms, most of which are ill-formed. That is, the deduction system allows the user to apply the XOR and encryption operators in meaningless ways. Secondly, certain necessary optimizations may cause OTTER to miss certain attacks. Specifically, an optimization called *demodulation* that collapses all "equivalent" forms of a predicate to one canonical form. Using demodulation alone may cause the tool to miss attacks that use non-canonical forms; whilst enabling re-writing using *paramodulation* fixes this, it also returns us to the state-space explosion we are trying to avoid. However, demodulation is a powerful tool, and it seems infeasible to analyze the API without it.

Our contributions provide a way of modeling an API that avoids both of these problems. We impose typing on the deduction system[2] which restricts the possible deductions to meaningful, well-formed forms. We also propose a way of representing inference rules that simultaneously allows demodulation but captures otherwise-overlooked attacks. Indeed, this technique can be thought of as making demodulation work "for us" instead of "against us."

Verification of Known Attacks With these two techniques, it was possible to re-discover in an automated way all previously-known attacks [3, 6, 12], all of which were originally found "by hand." These attacks are similar in that they exploit the CCA's weakness of using the XOR function both to combine type information with keys, and to build the keys themselves from *key parts*. This weakness allows the type of a key to be changed by using a cleverly chosen value for the key part. The type-casted keys can then be used to either recover an unknown key or establish a known key in the system in one of the following ways:

- the *export-and-decrypt* attack to recover a key (of types $EXP$and $DATA$),

[1]The actual form of this goal is slightly more complex, for reasons explained in Section 2.1. However, the basic intuition is the same.

[2]Not to be confused with the typing system of the API, which is being modeled in the deduction system.

- the *encrypt-and-import* attack to import a known value as a key (of type $DATA$), and
- the *export-import loop* followed by one of the previous two methods to recover an unknown key or establish a known key (of any type).

The implications of either recovering an unknown key or establishing a known key depend on the associated type of the key and the permissible API calls. For example, a known or recovered exporter key can be used to export and recover other keys from the system.

We note that our model was developed without knowledge of some of these attacks. However, all these documented attacks were found in a few minutes of computation.

1.4 Previous & Related Work

Security protocols, a close friend of security APIs, have been the subject of formal analysis since 1989, when the seminal paper by Burrows, Abadi and Needham introduced BAN logic [9]. Automated verification began soon thereafter with Meadows's work on the NRL protocol analyzer [18].

In 1995, Lowe used the CSP/FDR model checker to find and provably fix [14, 15] a serious flaw in the Needham-Schroeder mutual authentication public-key protocol [21]. A theorem prover tool was first used for protocol verification by Paulson [22]. The field has since expanded [19], including complete formal modeling efforts such as Strand Spaces [25].

In 2000, Anderson introduced the first security API attack, against the Visa Security Module (VSM). This attack was extended by [3] into a suite of attacks against the key management of IBM's CCA [2]. [6, 10, 12, 5, 7] extended this work with additional attacks and addressed the issue of secure API design. The scope of API attacks was further expanded to include PIN processing for financial and banking systems in [12, 5, 8] and popular standards such as PKCS#11 [1] in [11].

1.5 Motivation

Security APIs are inherently similar to security protocols, and much work has been done on security protocol analysis since the introduction of BAN logic in 1989. However, there are important differences between APIs and protocols that make Security API analysis more interesting and challenging, for example:

- **Attacker Ability:** The current approach in modeling security protocols use some variant of the Dolev-Yao model, where the underlying cryptographic primitives and algorithms are given ideal properties and the adversary is limited to read, modify and delete traffic. Specifically, the ability to modify is limited to actions like encryption, decryption, concatenation and xor [19].

  In a security API, the same assumption model is used except the attacker's ability to modify messages is augmented to also include making any API call using the contents of the message. Ultimately, this exposes a much larger attack surface.

- **Shared State:** We can also view a security API as a large set of short-chain protocols, where most chains are a single exchange of messages. However, the execution of one short protocol may result in a change in the state of the system. Since the state of the system is shared across any other API call, it represents a different mode of attack - *inter-chain* state attacks. In security protocol analysis, the interest is only in *intra-chain* state attacks.

Therefore, it seems that security APIs have a much more complex adversary threat model and therefore require different formal analysis techniques.

1.6 This Paper

We first present an overview of the IBM Common Cryptographic Architecture (Section 2). We use this API as an example with which to discuss the theorem-prover based modeling and analytical techniques we developed and used (Section 3 and 4). We conclude with a number of ideas for future work in turning these initial results into a full-blown, generic toolkit for discovering security API attacks (Section 6).

2 Targets

2.1 Overview of the 4758

The IBM 4758 is a hardware cryptographic co-processor [2], used in ATM networks and bank computer systems, which implements IBM's Common Cryptographic Architecture (CCA) to do so. Its purpose is two-fold:

1. To enable the ATM network to cryptographically authenticate the customer, and
2. To prevent anyone – including bank insiders – from gaining access to customers' accounts.

The bank issues customers with Personal Identification Numbers (PINs) with which to authenticate themselves to ATMs. Each PIN is *cryptographically* derived from the customer's Primary Account Number (PAN)[3] That is, when the bank first starts issuing PINs, the 4758 randomly generates a *PIN-derivation key* P, which will be used to generate all PINs for all customers. An individual customer's PIN is generated by encrypting that customer's PAN with this key[4]. We denote the encryption operation as:

$$PIN := \{PAN\}_P \tag{1}$$

Thus, to verify a PIN, the 4758 module must compute the true PIN by encrypting the PAN of the cusotmer's card under the PIN-derivation key, and then compare this true PIN with the number punched in by the customer himself.

If these two values match, the 4758 will allow the transaction to proceed. To protect the customer and to limit fraud, the 4758 must prevent the customer's true PIN and the PIN-derivation key from ever leaving the device. Note that loss of the first endangers the assets of one customer, while loss of the second puts at risk the accounts of *all* customers.

In addition to PIN processing functionality, the 4758 CCA needs a secure mechanism to exchange PIN-derivation keys, and to set up keys to protect transactions in transit from ATM to bank. For these purposes, the CCA API defines functionality that allows one 4758 (e.g. in the ATM network) to establish and use a secure channel with another 4758 (e.g., at the bank headquarters)

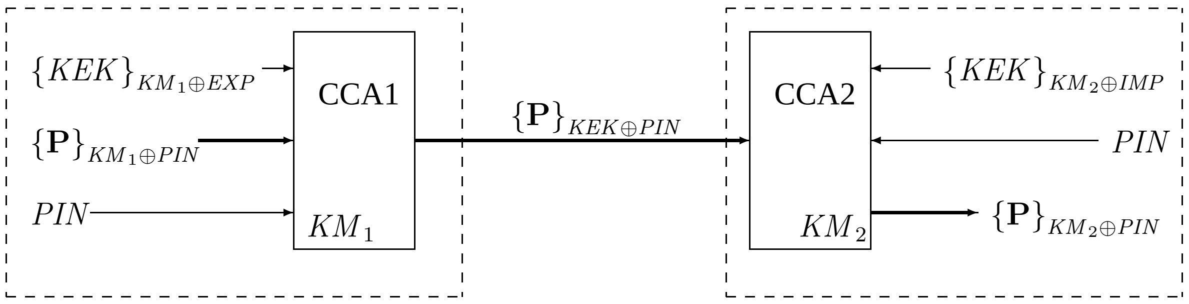
To establish the first connection between two 4758 modules, one module creates a fresh random key KEK (a *key-encrypting key*). KEK is not directly exported from the module . Instead, an elementary secret sharing algorithm is employed to produce three shares, called *key parts*. Each key part is sent by individual courier and fed into the other 4758. The parts are then XORed together in the module, and KEK is recovered.

It is unwieldy to send three couriers for every key to be shared between two 4758 modules. Therefore, once KEK is shared between two 4758 modules, other keys (including the PIN-derivation key P) can be encrypted with it and sent from one to the other. This process of encrypting one key with another is called *key wrapping*. These new keys (and KEK itself) can then be used for:

- deriving PINs (i.e., acting as a PIN-derivation key),
- encrypting arbitrary data, such as customer transactions,

[3]There are other PIN generation/verification techniques such as the VISA PIN Verification Value (PVV) method, but we will not go into them here.

[4]Technically, the PIN is only the first few digits of this encryption, but this will make no difference to our model.



$\{KEK\}_{KM_1 \oplus EXP}$
$\{\mathbf{P}\}_{KM_1 \oplus PIN}$
PIN
CCA1
KM_1
$\{\mathbf{P}\}_{KEK \oplus PIN}$
CCA2
KM_2
$\{KEK\}_{KM_2 \oplus IMP}$
PIN
$\{\mathbf{P}\}_{KM_2 \oplus PIN}$

Figure 1: Transfer of P from CCA-1 to CCA-2 via KEK

- acting as an *importer* key, and
- acting as an *exporter* key.

The last two require a brief explanation. *Importer* keys are those keys trusted to secure other keys arriving from some external source. *Exporter* keys are trusted to secure keys being sent to some external destination. In both cases, the importer and exporter keys wrap the key to be secured. For example, KEK will be generated by one 4758 and split into three parts. The generating 4758 designates KEK an exporter key. After this key is reconstructed from the three parts, the receiving 4758 will designate KEK an importer key. Thus, the first 4758 can generate a new key $K1$ and export it by wrapping it with the exporter key KEK. Likewise, the second 4758 can import it by decrypting (unwrapping) the so-generated ciphertext with the importer key KEK. Once again, once both modules have a shared key (this time, $K1$) they may use it for a joint cryptographic operation.

A simple observation exposes the need for strict control over wrapped keys: if a key $K1$ can be used both to import a wrapped key – $\{K2\}_{K1}$ – and to decrypt sensitive data, then it is trivial for an attacker to simply decrypt the wrapped key and obtain $K2$ in the clear. The CCA provides a mechanism for *key typing* which is meant to ensure that certain keys can only be used for certain actions.

Because much of the 4758 CCA transaction set concerns this dedication of keys to their appropriate uses, we describe it in further detail. Each individual 4758 card contains a *master key* KM. The on-board storage of a 4758 card is limited; with this master key, protected storage can be implemented using key wrapping and normal external memory. Instead of storing the pin-derivation key P in the 4758, for example, the ciphertext $\{P\}_{KM}$ is stored externally. Of course, this ciphertext must be fed to the 4758 by the operating system when the PIN-derivation key is required. In order to denote key type information, the CCA specification fixes a set of *control vectors*: binary strings that are public and unique for a given type across all implementations of the CCA. The CCA control vector system has many thousands of variant types, but we need only consider the major distinctions:

- PIN: used for PIN-derivation keys
- IMP: used for importer keys,
- EXP: used for exporting keys, and
- $DATA$: used DATA keys.

To dedicate a key to a particular purpose, it is *wrapped and typed* to that control vector. That is, it is encrypted not with the master key KM, but with the XOR of KM and the relevant type. Hence, the PIN-derivation key will actually be stored as $\{P\}_{KM \oplus PIN}$. For the rest of this paper, we will speak of a key as *being* the type used in the wrap-and-type operation. That is, when we say that a key K is a "$DATA$ key," we mean that the ciphertext $\{K\}_{KM \oplus DATA}$ is stored externally to the 4758. Similarly, if the ciphertext $\{K\}_{KM \oplus IMP}$ is stored, then K can be used as an importer key. Note that it is possible (and sometimes also correct) to obtain a given key wrapped and typed under more than one control vector.

Note that the method for recombining key parts also uses the XOR operation. This dual use of XOR is the basis of the attacks modeled in the next section.

3 Modeling Security APIs

A properly-defined security API provides:

1. **A set of function calls**. Generally, there is no required order as to which function call is performed when, though there can be constraints on the formats of inputs to these calls.

2. **A security policy**. This policy comprises security invariants meant to be guaranteed by the API. An API without a security policy cannot be studied, as there would be no security policy to attempt breaking.

A security API vulnerability consists of a sequence of legal function calls (and, potentially, offline computation) which leads to the violation of the security policy. In this section, we show how to model simple attacks where the number of functions is small and the vulnerability sequence is short.

3.1 A Large Search Tree Problem

Consider a typical API which may contain fifty function calls, and an attacker that can perform offline generic cryptographic functions – e.g. encryption and XOR. As an approximation, say each function expects two input terms and returns a single output term. We define the depth of an attack as the number of consecutive API and offline function calls required. Thus, searching through all attacks up to a depth of n takes time and space slightly worse than exponential in n. At level $k < n$ of an arbitrary search, any of the previous terms from the search can be used as input to any of the available b functions. As an upper bound, there are at most n^2b function calls available. The search has a total of n levels, each of which has an upper bound of n^2b possible function calls available. Thus, a complete search up to a depth of n will take $O(c^{2n\log(n)})$ time.

Though this simplified model ignores possible constraints on function data types, it illustrates the combinatorial explosion at the heart of verifying security APIs. Our modeling techniques must provide significant optimization techniques to make finding attacks possible.

3.2 Choice of Tool: the Theorem Prover OTTER

A notable property of many security APIs is their *effective statelessness*. Any knowledge gained by the adversary can be forever retained for use at a later time, and all API calls are accessible at all times – provided the operator can supply correctly-formed inputs. This statelessness is particularly true for systems like the Visa Security Module [4] and the 4758 (described in section 2), where external storage of wrapped keys is a prominent mechanism. The adversary becomes the external storage manager for these wrapped keys, and we assume the adversary has unlimited storage capacity.

Thus, in a series of interactions between an adversary and a security API, the knowledge of the adversary increases monotonically. Any knowledge gained and any goal achieved at time t remain known and achieved for all time after t.

These observations point naturally to theorem provers as a modeling tool. Truth statements expressed in first-order logic can represent knowledge and security invariants, both of which, once deduced, remain forever true. Logical inferences represent API calls and offline computation, with new, deduced statements representing acquired knowledge.

The theorem prover OTTER takes input written in first-order logic. OTTER was written by William McCune [17] and implements many of the innovative ideas of Larry Wos [29, 30, 28, 16]. OTTER is extremely well documented by McCune, Wos [27], and others. This paper assumes the reader is familiar with the common first-order logic terms: *predicate*, *literal*, *clause*, *completeness*, and *resolution* [23]. Informal definitions will be given as needed. For more detailed definitions, see [27].

Though there exist newer and potentially more advanced theorem provers based on First-Order Logic and beyond, we chose OTTER for its rich documentation and maturity. We suspect that our methods can be adapted to other first-order logic tools, and we intend to explore these for future work.

3.3 Target Simplification

A number of security API attack approaches exploit inherent cryptographic mistakes: a weak key, a bad cipher, a poor cipher chaining mode (e.g. ECB mode for a block cipher), etc .... These attacks are valid, but they do not fall within the specific goal of our study: pure API attacks that result from a logical flaw in the conception of the API.

In order to focus on these pure API attacks, we simplify the target model using concepts of the Dolev-Yao model [13] and a few bits of common sense:

1. **Perfect cryptographic primitives**. Ciphertexts can only be decrypted with knowledge of the key, and no information whatsoever about the plaintext or the key is leaked by the ciphertext itself.
2. **Atomic operations**. Encryptions of long plaintexts are considered atomic. Block chaining modes and the vulnerabilities they engender are ignored.
3. **Long, strong keys**. No key can be found by brute force search.
4. **Full Capability APIs**. While some security APIs restrict access to certain calls using role-based policies, we ignore these constraints. There may be false positives in our security analysis, but the simplicity gained is well worth it.

3.4 Basic Techniques

Security API modeling begins with the definition of the parties involved and the interactions between them. Specifically, the API user is, in effect, the adversary. He attempts to use the API to extract information which the API's security policy ostensibly disallows. In order to properly model this system, we need to consider:

1. User/adversary knowledge representation
2. The effect of API calls and offline computation on user/adversary knowledge
3. Security invariants
4. Computational properties of certain primitives like `XOR`

User Knowledge The fundamental unit of the model is the *user knowledge predicate* `U`. That is, user knowledge of a value `X` would be represented by setting to true the predicate

```
U(X).
```

Merely placing the above string in the model will suffice to specify that `U(X)` is true.[5]

User knowledge includes initial knowledge atoms, new deductions over time, and potential breaches of security policy. Thus, a secrecy goal is easily represented by negated user knowledge:

```
-U(MASTER_KEY).
```

API Calls and Offline Computation The inferences of this deduction system will be of two types: those deductions which the user can make offline, and API function calls. Inferences of the first type are simple. For example, the adversary can always XOR two known values:

```
-U(x) | -U(y) | U(xor(x,y)).
```

The above statement reads:

- User does not know x, OR

[5]Note that all OTTER statements must be terminated with a period.

- User does not know y, OR
- User knows the XOR of x and y.

Thus, if the user knows x AND the user knows y, the first two statements are false and the last one must be true. For technical reasons, OTTER prefers this disjunctive form to the equivalent `U(x) && U(y) => U(e(x,y))`.

In accordance with the previously mentioned Dolev-Yao concepts, we denote a ciphertext as a function of a plaintext and key. We express the user's ability to perform offline encryption and decryption using inferences similar to the above:

```
-U(x) | -U(y) | U(e(x,y)).
-U(e(x,y)) | -U(y) | U(x).
```

Because we state no further properties on ciphertexts, OTTER will be unable to reason out any relationship between ciphertexts using different plaintexts or keys. This is precisely the *ideal encryption* setting we outlined.

The API functions can be also simply represented as inferences: if the user knows some given input to a function call, he can learn the output. For example, the "Encrypt with Data Key" function call of the 4758 can be represented as:

```
-U(x) | -U(e(K, xor(KM,DATA))) | U(e(x,K)).
```

Because the user does not know `K` in the clear, the above rule is the only inference path by which the user can obtain the ciphertext of `x` under `K`. Thus, this inference correctly represents the *capability* offered by the security API: if the user knows `x` and the key `K` wrapped as a $DATA$ key under the master key KM, the user can obtain the ciphertext of any plaintext it knows under this key `K`.

Security Invariants When OTTER is run, it attempts to prove an inconsistency in its input. As such, theorems to be proven are entered in a negated form: if an inconsistency is discovered, the original theorem is true. In the case of Cryptographic APIs, we add the security goals to the input because "proofs" are actually attacks on an API: steps that derive a clause that contradicts a security goal. For example, if one security goal was that the master key `KM` is never learned by the user, one would include in the specification the statement:

```
-U(KM).
```

If OTTER derives the clause `U(KM)`, then its proof will show how that security goal has been violated.

Modeling Computational Properties Because encryption is considered ideal, we provide OTTER with inference rules limited to opaque encryption and decryption operations, assuming proper user knowledge. Certain other operations, however, do have important computational properties that OTTER needs to know about for the model to make any sense at all.

Specifically, in a number of the security APIs we have studied, the `XOR` operator plays a major role in numerous API calls. This operator has important computational properties crucial to the successful function of the security API. As it turns out, these same properties are crucial to the attacks, too.

We represent these properties using OTTER's ability to operate over equalities:

```
xor(x,y) = xor(y,x).
xor(x,x) = ID.
xor(x,ID) = x.
xor(xor(x,y),z) = xor(x,xor(y,z)).
```

The strings `x`, `y` and `z` are automatically assumed to be variables. `ID` is a literal, and OTTER will not automatically assume that `ID` represents the identity element for the `xor` operator. Thus, the behavior of `ID` needs to axiomatized.

One should note how the simple property of commutativity of `XOR` needs to be axiomatized in OTTER. This should help clarify how we model ideal operation of other primitives like encryption: where we do not specify a property, OTTER simply will not assume it. OTTER reasons only on inferences that are explicitly stated.

4 Finding Complex Attacks

4.1 Complications

A model of the 4758 created using the methods in section 3 will be perfectly valid. However, it will also take far too long for OTTER to reason about it, and it may even fail altogether. In this section, we describe how to achieve two major goals:

1. Optimize a model using various theorem prover tricks.
2. Ensure that such optimization does not impede completeness of the search.

4.2 Partitioning Adversary Knowledge

The above naive specification of the user's offline encryption capabilities allows for terms to be derived that are unlikely to be of use to an attacker. For example, given that the attacker initially knows two control vectors `U(DATA)` and `U(IMP)`, he could derive:

```
            U(e(DATA,IMP).
        U(e(DATA,e(DATA,IMP))).
    U(e(DATA,e(DATA,e(DATA,IMP)))).
       U(xor(e(DATA,IMP),DATA)).
                  ⋮
```

To limit an attacker's ability to perform encryption, we only allow the attacker to encrypt a message that is not the result of a previous encryption. Though this approach may fail to model certain classes of attacks, we believe these attacks are unlikely to succeed under the assumption of ideal cryptographic primitives. Likewise, the key used for encryption cannot be the result of a previous encryption. To enforce this standard, we must introduce types. Data that is the result of an encryption will be of one type, while data that is not the result of an encryption will be another type. We use the predicate `UE(x)` to represent that the attacker knows x, and x is the result of an encryption. `UN(x)` is the predicate that represents an attacker knows x and that x is *not* the result of a previous encryption. Together, these two predicates replace the previous "attacker knowledge predicate" `U(x)`. Now, the attacker ability to encrypt a message x under a key k can be represented by the clause:

```
-UN(x) | -UN(k) | UE(e(x,k)).
```

In addition, we decided that it is unlikely that an attack would require calculating the XOR of any known constant with a result of a previous encryption. We model the attackers ability to XOR two known messages x and y as:

```
-UN(x) | -UN(y) | UN(xor(x,y)).
```

These simplifications greatly reduce the size of our search space. Specifically, it can be easily shown that the number of literals that the attacker can learn is finite under this model of XOR and encryption. Theoretical work is currently underway to exactly specify what attacks might be missed by using this data typing simplification. The above simplifications were necessary before OTTER was able to successfully able to recreate known attacks originally found by Mike Bond and Ross Anderson [6] on the 4758.

4.3 Intermediate Steps

The search optimization techniques built into OTTER can be counter-productive under certain circumstances. *Demodulation* [30] is a rewrite rule that acts to simplify clauses. When a clause is demodulated, it is replaced with the simplest form of the clause. OTTER defines the simplest form to be the form with the fewest symbols, and, if the number of symbols is equal, the simpler clause has a lower lexicographical ordering. OTTER's default lexicographical ordering of symbols is an alphabetical ordering. Because the weighting and lexicographical ordering of terms is a total ordering on equivalent clauses, if clause $C1$ is equivalent to clause $C2$, they will both demodulate to the same clause $C3$. Demodulation is initially applied to every input clause, and also applied to every clause that is derived.

We use demodulation mainly to deal with the function XOR. Because XOR is associative and commutative, there can be many equivalent ways to rewrite a given term. For example, without demodulation OTTER may derive all of the following clauses:

```
UN(xor(B,xor(C,D))).
UN(xor(xor(B,C),D)).
UN(xor(D,xor(B,C))).
UN(xor(xor(C,B),D)).
         ⋮
```

although they are all equivalent. However, if we declare the following equalities as *demodulators*:

```
      xor(x,y) = xor(y,x).
         xor(x,x) = ID.
         xor(x,ID) = x.
xor(xor(x,y),z) = xor(x,xor(y,z)).
```

OTTER will only retain the first clause `U(xor(B,xor(C,D)))`. However, using demodulation has its costs. In particular, when OTTER uses existing knowledge to perform one of the transactions it must *unify* variables to knowledge we already know. In some cases, it may be *impossible* to *unify* the variables in the transaction to knowledge we already know *even if* we know equivalent information. For example, the clauses:

```
-UN(x)|-UN(y)|-UN(xor(x,y))|GOAL(x).
          UN(xor(B,C)).
          UN(xor(D,E)).
    UN(xor(B,xor(C,xor(D,E)))).
         -GOAL(xor(B,C)).
```

will not unify and the initial transaction will not be able to proceed. To resolve these clauses, OTTER may attempt to unify the variable `x` with `xor(B,C)`, and `y` with `xor(D,E)`, but then the term `xor(x,y)` will need to unify with a term of the form `xor(xor(B,C),xor(D,E))`. Because the form required for unification is not the same as the simplest form, `GOAL(xor(B,C))` cannot be derived.

We introduce an *intermediate clause* to solve this problem, a method developed by author Youn [31]. We do not know if this approach has been used before. We split the implication clause into two separate clauses:

```
-A(x)|-A(y)|INTERMEDIATE(xor(x,y),x).
 -INTERMEDIATE(z,x) | A(z) | GOAL(x).
```

In this case, we can take advantage of demodulation. Once

```
INTERMEDIATE(xor(xor(B,C),xor(D,E)),xor(B,C)).
```

is derived, OTTER will demodulate it into the simplest form:

```
INTERMEDIATE(xor(B,xor(C,xor(D,E))),xor(B,C)).
```

Then, in the second clause, the variable `z` can be unified with the term `xor(A,xor(B,xor(C,D)))`, `x` can be unified with `xor(B,C)`, and `GOAL(xor(B,C))` can be derived. Notice that the intermediate clause contains the variable `x` by itself, which is necessary to eventually derive `GOAL(x)` because `x` is not recoverable from `xor(x,y)`.

Unfortunately, by using intermediate steps, we may derive extra clauses. These extra clauses are undesirable, but because the use of intermediate clauses can be limited to cases where we know unification will be difficult, it is much preferable to running OTTER without demodulation which will blindly retain numerous equivalent versions of arbitrary clauses.

Other simple barriers to unification involve term cancellation and pattern matching, and are solved by keeping several versions of implication clauses that explicitly deal with these barriers. For example, the clause

```
-A(xor(x,B))|GOAL(x)
```

would be expanded into:

```
-A(xor(x,B))|GOAL(x)
-A(xor(B,x))|GOAL(x)
-A(x)|GOAL(xor(x,B))
```

4.4 Forwards/Backwards Search

Our approach to finding an API-level attack on an API specification is akin to exploring a search graph until the goal is found: the start node represents the initial knowledge of the adversary and edges represent legal API calls resulting in more adversary knowledge.

In the process, however, many clauses were derived that did not contribute in finding the goal. We call these *chaff clauses* since they distract the reasoning tool from finding the goal. While deriving *chaff clauses* is unavoidable in general, we can avoid deriving too many by having the theorem-prover work more efficiently. One way is to have the theorem prover reason from both directions:

- reasoning forwards from the adversary's initial set of knowledge, determining additional knowledge with each step
- reasoning backwards from the adversary's goals, determining what the adversary has to know in order to reach his goal

OTTER provides a variant of hyper-resolution – negative hyper resolution – that we use to accomplish backwards search. This can be turned on by setting the flag `neg_hyper_res` in OTTER. Negative hyper-resolution works in exactly the same way as positive hyper-resolution except that unification is done on negative literals: each positive literal in the nucleus has to be unified with a negative literal in the satellites.

$Nucleus$	:	`-A(x)	-B(x)	C(x)`
$Satellite$	:	`-C(GOAL)`
$Negative\ Hyperresolvent$	:	`-A(GOAL)	-B(GOAL)`

Now, instead of proving *C(GOAL)* to derive a contradiction, OTTER can now prove both *A(GOAL)* and *B(GOAL)*.

However, this approach does not always speed up OTTER's search. When a reasoning tool is given more deductive power, it also ends up producing more *chaff clauses*. These *chaff clauses* are either *units* (e.g. `-A(x)`) or *non-units* (e.g. `-A(x) | -B(x) | -C(x)`). *Non-units* tend to cause clause explosion when used with negative hyper-resolution since they resolve to form larger non-unit clauses with more variables. Ultimately, it is the production ratio of *chaff clauses* versus useful clauses that will determine whether negative hyper-resolution is useful. This ratio can be tuned by setting other OTTER parameters such as `pick_given_ratio`, which picks clauses with lower weight first (thereby delaying reasoning about huge clauses) and `max_distinct_vars`, which discards clauses if they contain more than a certain number of variables.

In section 5, we show results demonstrating how turning on negative hyper-resolution may improve or deprove OTTER's search speed.

5 Recreation of Attacks & Results

5.1 VSM Attacks

The two following attacks on the Visa Security Module were modeled and recreated efficiently with OTTER's first-order logic language.

XOR to null key attack To distribute key shares and combine them successfully at the destination ATM to form the *terminal master key*, the VSM security API provides 2 commands:

1. "Generate Key Share": returns a key share wrapped under a master key.

```
U(e(Share,KM))
```

2. "Combine Key Share": takes in two key shares and returns the XOR of the shares wrapped under the same master key.

```
-U(e(Share1,KM)) | -U(e(Share2,KM)) | U(e(xor(Share1,Share2),KM))
```

It can be seen that providing the same key share twice will result in a NULL terminal master key.

Key type attack The VSM security system uses a key typing system, where key types are differentiated by encryption under separate master keys. However, the VSM allows the re-use of key types, enabling an attacker to calculate any user's PIN number. The following API calls constitute the attack:

1. "Insert *Terminal Communications Key*": This allows a clear key to be entered as a TCK.

```
-U(x) | U(e(x,TCK))
```

2. "Rewrap with *Terminal Master Key*": This allows a TCK to be rewrapped under a TMK such that the TCK can be transferred to an ATM.

```
-U(e(x,TCK)) | -U(e(y,TMK)) | U(e(x,y))
```

The attack involves passing in a *primary account number* as a TCK and rewrapping that using the PIN derivation key. This represents the simplest attack that requires more than a single API call. The attack was found by OTTER in a fraction of a section, with only 215 generated clauses.

5.2 IBM 4758 Attacks

The attacks on the IBM 4758 are considerably more complex than those on the VSM. We briefly describe two attacks with the required API calls and demonstrate how our modeling techniques helped in the recreation of these attacks using OTTER.

PIN derivation attack This attack allows the attacker to obtain the PIN of any *primary account number*. First, key-type confusion is introduced with ‘Key Part Import’, yielding a bogus importer key. Next, the PIN derivation key is imported and typed as a ‘DATA’ key. Finally, the PIN derivation key is used as a $DATA$ key to encrypt any *primary account number*. The following API calls represent the *skeletal* model for the attack:

- **Encrypt with DATA key** takes an arbitrary message m and a $DATA$ key and produces the corresponding ciphertext:

$$m, \{K\}_{KM \oplus DATA} \rightarrow \{m\}_K \tag{2}$$

```
-U(x) | -UE(e(y,xor(KM,DATA))) | UE(e(x,y)).
```

- **Key Import** takes a specified type (`x`) and the target key (`y`) encrypted under an importer key (`z`), as well as the importer key blob and ‘imports’ the target key into the system by rewrapping the key under the 4758’s master key, typed with `x`:

$$T, \{K1\}_{KM \oplus IMP}, \{K2\}_{K1 \oplus T} \rightarrow \{K2\}_{KM \oplus T} \tag{3}$$

```
-UN(x) | -UE(e(z,xor(IMP,KM))) | -UE(e(y,xor(z,x))) |
            UE(e(y,xor(x,KM))).
```

We modeled this command using an *intermediate clause* to speed up OTTER’s reasoning.

```
-UN(x) | -UE(e(z,xor(IMP,KM))) | INT(e(y,xor(z,x)),xor(x,KP)).
 -INT(e(y,x),xor(w,KP)) | -UE(e(y,x)) | UE(e(y,xor(w,KM))).
```

- **Key Part Import** is used to merge key shares in a destination 4758. We modeled only the part of the command that imports the third key share.

$$K3, T, \{K1 \oplus K2\}_{KM \oplus kp(T)} \rightarrow \{K1 \oplus K2 \oplus K3\}_{KM \oplus T} \tag{4}$$

```
-UN(xor(x, KP)) | -UN(y) | -UE(e(z,xor(x,xor(KP,KM)))) |
          UE(e(xor(z,y), xor(x,KM))).
```

Discover PIN derivation key attack In modeling the previous attack, we added related API calls to make the model more complete and realistic - if an API call was present in the model, so would any of its logical counterparts. This model would be closest to one that would be created if the modeller was trying to search for a vulnerability and not attempting to recreate a known attack. Therefore, we added the following 3 commands to create what we term the *complete* model.

- **Decrypt with DATA Key** takes an arbitrary ciphertext and a $DATA$ key and returns the decryption:

$$\{m\}_K, \{K\}_{KM \oplus DATA} \to m \tag{5}$$

```
-UE(e(x,y)) | -UE(e(y,xor(KM,DATA))) | U(x)
```

- **Key Export** is simply the conceptual inverse of Key Import.

$$T, \{K1\}_{KM \oplus EXP}, \{K2\}_{KM \oplus T} \to \{K2\}_{K1 \oplus T} \tag{6}$$

```
-UN(x) | -UE(e(y,xor(x,KM))) | -UE(e(z,xor(EXP,KM))) |
                    UE(e(y, xor(x,z)))
```

  This encrypted blob is intended for another CCA which has `z` as an importer key.

- **Key Generate** This command generates a key randomly and returns it as two keys (a matching pair of import and export keys). This command was not modelled as an implication but as initial attacker knowledge.

$$\begin{gathered} K := RandomKey(), TYPE := IMP, EXP, DATA... \\ () \to \{K\}_{KM \oplus TYPE} \end{gathered} \tag{7}$$

```
UE(e(K,xor(KM,TYPE)))
```

The aim was to determine if OTTER was able to find the same PIN derivation attack despite these additional unimportant API calls. In the process, we made two exciting discoveries:

- OTTER was able to find the same attack albeit using more time and space (more implications and more generated clauses). Furthermore, our modeling techniques displayed the same effectiveness in cutting down the search space (refer to section 5.3 for actual numbers).

- OTTER was able to find a different attack in our model - one that we had not intended to find. This additional attack allows the attacker to discover the PIN derivation key in the clear. Although this attack had been found before by *M. Bond et al*, these findings attest to the usefulness of our techniques in modeling the 4758.

5.3 Attack Statistics

The tables below describes how OTTER performed in recreating the manually found PIN derivation attack [3, 6, 12] under various modeling techniques. Table 1 gives statistics taken from running OTTER on the skeletal API model (the minimum commands required to demonstrate the type-casting attack) while Table 2 gives statistics taken from running OTTER on the complete model.

The numbers in Table 1 demonstrate that a naive approach (1) did not find any attacks (OTTER fails to terminate as far as we can tell). By partitioning adversary knowledge (2), OTTER was able to find the attack but only after a very substantial amount of time. Turning on demodulation (2) gave OTTER the ability to canonicalise XORs and reduce the number of *chaff clauses* but in the process missing the attack. Subsequently, the addition of *intermediate clauses* (3) enabled OTTER to unify with demodulated terms, allowing OTTER to find the attack once again. Finally, we gave OTTER the backwards searching capability and it continued to find the attack but using significantly less time and space.

Table 1: Skeletal Model (3 API calls) - otter 3.3f running on a 3.2 Ghz P4 with 2GB RAM

No.	Strategies	Attack Found?	Clauses Generated	Clauses Kept	System CPU Time (sec)
1	Vanilla	No	n/a	n/a	n/a
2	Partitioned Knowledge	Yes	TBC	TBC	TBC
3	Partitioned Knowledge, Demodulation	No	33,154	16,652	0.09
4	Partitioned Knowledge, Demodulation, Intermediate Steps	Yes	49,666	33,165	0.12
5	Partitioned Knowledge, Demodulation, Intermediate Steps, Backwards Searching	Yes	27,832	15,860	0.01

Table 2: Complete Model (6 API calls): otter 3.3f running on a 3.2 Ghz P4 with 2GB RAM

No.	Strategies	Attack Found?	Clauses Generated	Clauses Kept	System CPU Time (sec)
1	Vanilla	No	n/a	n/a	n/a
2	Partitioned Knowledge	Yes	5,458,647	8,31	28
3	Partitioned Knowledge, Demodulation	No	892,019	265,805	5.93
4	Partitioned Knowledge, Demodulation, Intermediate Steps	Yes	1,663,954	389,424	3.46
5	Partitioned Knowledge, Demodulation, Intermediate Steps, Backwards Searching	Yes	301,986	103,739	0.2

The numbers in Table 2 reflect how OTTER performed in trying to recreate the same PIN derivation attack but with three additional unrelated API calls. The addition of these calls represent a more complete and realistic model – one that a security API modeller would have made without prior knowledge of an attack.

OTTER continued to find the attack when it should have but the search space was understandably larger in all corresponding experiments. Moreover, OTTER demonstrated the same trends with regards to the various modeling techniques used, suggesting that if an attack does exist in a set of API calls S, the same attack can still be deduced in a superset of API calls S' and any modeling improvements used for S will yield the same effects when used in S'.

6 Conclusion & Future Work

With this work, we have shown the first successful application of formal tools to the analysis of security APIs. Using these tools, we have re-discovered an entire family of attacks against a widespread hardware security module – the IBM 4758 CCA. We also found a novel variant of one of these attacks.

We accomplished these results through a series of techniques to optimize the attack search process while simultaneously ensuring that attacks would not be missed by these optimizations. We believe these techniques will prove quite useful in analyzing a range of security APIs. We suspect these techniques will soon help discover new attacks that might be too ununtuitive to find by hand.

In addition, our results emphasize the usefulness of theorem provers like OTTER in the verification of security APIs. Theorem provers have become very efficient (note the speed of attack discovery) and appear particularly talented at discovering non-intuitive attacks.

We expect this research to continue in at least four interesting directions:

1. **further optimization**: we are currently working to ensure that our use of the OTTER Theorem Prover provides a more focused search of the *real attack tree*, even with optimizations. The real attack tree is a pruned version of the naive attack tree: we believe theoretical results can rule out entire subtrees and help fight the computational explosion problem inherent to API verification.

2. **improved modeling tools**: even if we had a perfect theorem proving approach, modeling remains a very human — and thus very fallible — process. One likely research direction is the construction of pre-processing tools that might greatly simplify the modeling task in order to minimize human error. Another direction is the exploration of newer theorem provers and alternative tools that straddle the model-checker/theorem-prover divide.

3. **modeling information leakage**: it will be useful to move away from the ideal encryption assumption and find ways of reducing the computational requirements for brute-force key-breaking efforts. Modeling such information leakage attacks is a fascinating and challenging direction for this type of research.

4. **new attack discoveries**: our methodology is ready to take on new, more complex APIs. We fully expect the techniques presented here to be used in discovering new and powerful attacks against a number of API targets.

We hope these techniques and future improvements will help bridge the gap between the high-level security policies and low-level specifications of security APIs.

References

[1] : 2001, 'RSA Security Inc. Public-Key Cryptography Standards (PKCS)'. RSA Laboratories. Available at: `ftp://ftp.rsasecurity.com/pub/ pkcs/pkcs-11/v211/pkcs-11v2-11r1.pdf`.

[2] : 2003, ‘IBM PCI Cryptographic Coprocessor: CCA Basic Services Reference and Guide Release 2.41, Revised September 2003 for IBM 4758 Models 002 and 023’. International Business Machines Corporation.

[3] Bond, M.: 2001a, ‘Attacks on Cryptoprocessor Transaction Sets’. In: *CHES*. Berlin, pp. 220–234.

[4] Bond, M.: 2001b, ‘Attacks on Cryptoprocessor Transaction Sets’. Presented at the CHES 2001 Workshop in Paris.

[5] Bond, M.: 2004, ‘Understanding Security APIs’. Ph.D. thesis, University of Cambridge.

[6] Bond, M. and R. Anderson: 2001, ‘API-Level Attacks on Embedded Systems’. *IEEE Computer Magazine* **34**(10), 67–75.

[7] Bond, M. and R. Anderson: 2003, ‘Protocol Analysis, Composability and Computation’. *Computer Systems: Theory, Technology and Applications*.

[8] Bond, M. and P. Zielinkski: 2003, ‘Decimalisation Table Attacks for PIN Cracking’. Technical Report TR-560, University of Cambridge, Cambridge.

[9] Burrows, M., M. Abadi, and R. Needham: 1990, ‘A Logic of Authentication’. *ACM Transactions in Computer Systems* **8**(1), 18–36.

[10] Clayton, R. and M. Bond: 2003, ‘Experience Using a Low-Cost FPGA Design to Crack DES Keys’. In: *Cryptographic Hardware and Embedded System - CHES 2002*, Vol. 2523. pp. 579–592.

[11] Clulow, J.: 2003a, ‘On the Security of PKCS #11’. In: *CHES*. Berlin, pp. 411–425.

[12] Clulow, J. S.: 2003b, ‘The Design and Analysis of Cryptographic Application Programming Interfaces for Devices’. Master’s thesis, University of Natal, Durban.

[13] Dolev, D. and A. Yao: 1983, ‘On the Security of Public-Key Protocols’. *IEEE Transactions on Information Theory* **29**, 198–208.

[14] Lowe, G.: 1995, ‘An Attack on the Needham-Schroeder Public Key Authentication Protocol’. *Information Processing Letters* **56**(3), 131–136.

[15] Lowe, G.: 1996, ‘Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR’. In: *Proceeedings of* TACAS, Vol. 1055 of *Lecture Notes in Computer Science*. pp. 147–166.

[16] McCharen, J., R. Overbeek, and L. Wos: 1976, ‘Complexity and Related Enhancements for Automated Theorem- Proving Programs’. *Computers and Mathematics with Applications* **2**, 1–16.

[17] McCune, W.: 2003, ‘Otter 3.3 Reference Manual’. Aragonne National Laboratory, Argonne, Illinois.

[18] Meadows, C.: 1994, ‘A Model of Computation for the NRL Protocol Analyzer’. In: *Proceedings of the Computer Security Foundations Workshop VII*. pp. 84–89.

[19] Meadows, C.: 2003, ‘Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends’. *IEEE Journal on Selected Areas in Communication* **21**(1), 44–54.

[20] Miles, B., ‘Win32 API Tutorial’. `http://www.winprog.org/tutorial/`.

[21] Needham, R. and M. Schroeder: 1978, ‘Using Encryption for Authentication in Large Networks of Computers’. *Communications of the ACM* **21**(12).

[22] Paulson, L. C.: 1997, ‘Proving Properties of Security Protocols by Induction’. In: *10th IEEE Computer Security Foundations Workshop*. pp. 70–83.

[23] Robinson, J. A.: 1965, ‘A Machine-Oriented Logic Based on the Resoultion Principle’. *Journal of the Association for Computing Machinery* **12**(1), 23–41.

[24] Shamir, A.: 1979, ‘How to Share a Secret’. *Communications of the ACM* **22**, 612–613.

[25] THAYER Fábrega, F. J., J. C. Herzog, and J. D. Guttman: 1999, ‘Strand Spaces: Proving Security Protocols Correct’. *Journal of Computer Security* **7**(2/3), 191–230.

[26] Walli, S. R.: 1995, ‘The POSIX family of standards’. *StandardView* **3**(1), 11–17.

[27] Wos, L. and G. Pieper: 1999, *A Fascinating Country in the World of Computing, Your Guide to Automated Reasoning*. New Jersey: World Scientific Publishing.

[28] Wos, L. and G. Robinson: 1968, ‘Paramodulation and Set of Support’. In: *IRIA Symposium on Automatic Demonstration*. Versailles.

[29] Wos, L., G. Robinson, and D. Carson: 1965, ‘Efficiency and Completeness of the Set of Support Strategy in Theorem Proving’. *Journal of the Association for Computing Machinery* **12**(4), 536–541.

[30] Wos, L., G. Robinson, D. Carson, and L. Shalla: 1967, ‘The Concept of Demodulation in Theorem Proving’. *Journal of the Association for Computing Machinery* **14**(4), 698–709.

[31] Youn, P.: 2004, ‘The Analysis of Cryptographic APIs Using Formal Methods’. Master’s thesis, Massachusetts Institute of Technology.

A Complete Model

```
% Description:
% Shows how adversary can derive user’s pin from a crypto box.
%  Notation:
%   e(x,k) denote encryption of x with key k.
%   UE(x) means "User knows encryption x".
%   UN(x) means "User knows non-encryption x".
%   KM is the master key for the 4758.
%   KEK represents a key-encrypting key.
%   EXP1 represents an exporter key.
%   K3 represents the final part of KEK that is known to
%      the attacker.
%   P is the Pin Derivation key.
%   DATA is a control vector designating a key as a data key.
%   IMP is a control vector designating a key as an importer.
%   EXP is a control vector designating a key as an exporter key.
%   PIN is a control vector designating a key as an pin derivation key.
%   KP represents the key part bit in a control vector. In this
%    specification, we do not use kp(x) to designate a control vector x
%    is a key part, but rather use x $oplus$ KP. In particular, this
%    requires careful representation of the Key Part Import,
%    Key Import, and Key Export functions to accurately model the 4758.

% OTTER INPUT FLAGS
set(process_input).

% output information
clear(print_kept).
clear(print_new_demod).
clear(print_back_demod).
clear(print_back_sub).
clear(control_memory).

%runtime constraints, memory usage
assign(max_mem, 500000).
assign(stats_level,1).
%maximum allowed running time.
assign(max_seconds,1000000).
%maximum number of proofs produced.
assign(max_proofs, 10).
%maximum depth of attack
assign(max_levels, 100).

%disable paramodulation
clear(para_from).
clear(para_into).
clear(para_from_right).
clear(para_into_right).
clear(para_from_vars).
set(eq_units_both_ways).


%Use lexicographical ordering
set(lrpo).

%Perform a breadth first search
set(sos_queue).

%use hyperresolution
set(hyper_res).
set(order_hyper).

%The lexicographical ordering of terms. The term KM is listed
%as having the highest lexicographical ordering because it
%often appears in clauses XORed with variables. Because we
%do not know what constant will unify with that variable, by
%making KM have the highest lexicographical order, we can still
%ensure that all possible unifications will occur.
lex( [ ID, Acc, DATA, IMP, K3, KEK, PIN, EXP1, EXP, KP,KM] ).



%This list represents all actions an attacker can take.
list(usable).

%----------------------------------------------
%          Attacker Abilities
%----------------------------------------------

%Offline things Adversary can do:

%encrypt something
-UN(x) | -UN(y) | UE(e(x,y)).

%decrypt if knows stuff.
-UE(e(x,y)) | -UN(y) | UN(x).

% ability to xor
-UN(x) | -UN(y) | UN(xor(x,y)).


%----------------------------------------------
%              Transaction Set
%----------------------------------------------

% Command : Encrypt using data key
-UN(x)          | -UE(e(y,xor(DATA,KM)))
```

```
                |  UE(e(x,y)).

%Command: Clear_Key_Import
-UN(x) | UE(e(x,xor(DATA,KM))).


% Command : Key Import
% Note that the predicate INTUE keeps the term xor(x,KP) and not
% just the term x. By doing this, if the claimed type x were a
% key part (included XOR with KP), the two KP terms will cancel
% when INTUE is demodulated, and the predicate will not unify
% with the second clause.
%INTERMEDIATE STYLE:
-UN(x) | -UE(e(z,xor(IMP,KM))) | INTUE(e(y,xor(z,x)),xor(x,KP)).

-INTUE(e(y,x),xor(w,KP)) | -UE(e(y,x))
        | UE(e(y,xor(w,KM))).


% Command : Key Part Import
% Here, the claimed type is required to involve the XOR with KP
% to enforce that the claimed type must be a key part.
% The first transaction represents calling Key Part Import and
% completing the key. The second transaction represents calling
% the function and not completing the key.
-UN(xor(x, KP)) | -UN(y)        | -UE(e(z,xor(x,xor(KP, KM))))
                |  UE(e(xor(z,y), xor(x, KM))).

%Don’t complete
-UN(xor(x, KP)) | -UN(y)        | -UE(e(z,xor(x,xor(KP, KM))))
                |  UE(e(xor(z,y), xor(x, xor(KP, KM)))).

% Command : Key Export

% This command does the opposite of key import.
% It takes an exporter key (a key with type EXP,
% rather than IMP) and uses it to export any key
% encrypted under the local master key KM and
% encrypts it (with it’s type) under the exporter.
%Here dealing with cancellation explicitly will save time.
%no cancellation with KM
-UN(x)                  |
-UE(e(y,xor(x,KM)))     |
-UE(e(z,xor(EXP,KM)))   |
UE(e(y,xor(z,x))).

%cancellation with KM
-UN(xor(x, KM)) |
-UE(e(y,x))     |
-UE(e(z,xor(EXP,KM)))   |
UE(e(y,xor(z,xor(x,KM)))).

% Command : Decrypt using data key

% This does the opposite of the encrypt with
% data key command.
-UE(e(x,y))             |
-UE(e(y,xor(DATA,KM)))  |
UN(x).


end_of_list.

%This list contains the security goals of the system, as well as
%all initial knowledge.
list(sos).
%Security goals of the 4758
-UE(e(Acc, P)).
-UN(KEK).
-UN(KM).
-UN(P).

%---------------------------------------------
%         Initial Knowledge
%---------------------------------------------

UN(DATA).
UN(PIN).
UN(ID).
UN(IMP).
UN(K3).
UN(Acc).
UN(KP).
UN(EXP).

UE(e(P,xor(KEK,PIN))).

UE(e(xor(K3,KEK),xor(IMP,xor(KM, KP)))).

%The result of calling key generate
UE(e(KEK2,xor(IMP,xor(KM, KP)))).
UE(e(KEK2,xor(EXP,xor(KM, KP)))).

UE(e(EXP1,xor(KM,EXP))).

end_of_list.

%XOR demodulators
list(demodulators).
```

```
xor(x,y) = xor(y,x).
xor(x, xor(y,z))= xor(y,xor(x,z)).
xor(x,x) = ID.

xor(ID,x) = x.

% The rest of these demodulators have proved to be useful in
% allowing Otter to reason quickly, but are not strictly
% necessary.
xor(xor(x,y),z) = xor(x,xor(y,z)).

xor(x, xor(y, xor(z, xor(x,w)))) = xor(y, xor(z,w)).
xor(x, xor(y, xor(z, xor(w,x)))) = xor(y, xor(z,w)).
xor(x, xor(y, xor(x, z))) = xor(y,z).
xor(x, xor(y, xor(z, x))) = xor(y,z).
xor(x, xor(x,y))= y.
end_of_list.
```