The Analysis of Cryptographic APIs Using The Theorem Prover Otter

by

Paul Youn

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004 [June 2004]

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .... [illegible] ....
Ronald Rivest
Professor
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Analysis of Cryptographic APIs Using The Theorem Prover Otter

by

Paul Youn

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

Abstract

In 2000, Bond and Anderson exposed a new family of attacks on application programming interfaces (APIs) of security modules. These attacks elicit compromising behaviors using an unexpected sequence of legal calls to the module, uncovering severe security flaws even in widely-deployed cryptographic hardware. Because these attacks do not depend on the underlying cryptographic mechanisms, they often succeed even under the assumption of ideal cryptographic primitives.

This thesis presents a methodology for the automatic detection of API attacks. Taking a cue from previous work on the formal analysis of security protocols and noting these attacks' independence from precise cryptographic mechanisms, we model APIs opaquely, purely according to specifications. We use a theorem prover tool and adapt it to the security API context. Several specifications of Cryptographic APIs are implemented for analysis using a theorem prover known as OTTER . These implementations successfully found known attacks, and provide evidence that OTTER will also be able to find new attacks, and perhaps eventually verify security in arbitrary Cryptographic APIs. Based on these implementations, various strategies, potential problems, and solutions are discussed that can be applied towards the formal analysis of Cryptographic APIs. We detail how, using these formalization and automation techniques, we have confirmed a number of known attacks and exposed an undocumented behavior of the IBM 4758 CCA, a hardware add-on crucial to a large portion of banking transactions worldwide. We show how the confirmed attacks' complexity and unintuitiveness make a very strong case for continued focus on automated formal verification of cryptographic APIs.

Thesis Supervisor: Ronald Rivest
Title: Professor

Acknowledgments

I would like to thank Professor Rivest for his support, help, advice, and ideas. In addition, the rest of my research group including Jon Herzog, Ben Adida, Amerson Lin, Ross Anderson, Mike Bond, and Jolyon Clulow. I would also like to thank the Cambride MIT program for funding this reasearch project.

The writing in this thesis has significant overlap with the paper [31]. In particular, writing from the Abstract, Introduction, and description of the original attack on the 4758 CCA in section 4.2 was done largely by Ben Adida. Jon Herzog assisted in writing Chapter 5. The remaining members of the research group assisted in editing and revising the submitted paper.

Finally, I'd like to thank my friends and family, without whose support I would not have been able to complete my research.

Contents

List of Figures

Chapter 1

Introduction

1.1 Security APIs and Attacks

An *Application Programming Interface (API)* is a set of calls exposed by a hardware or software module for the purpose of giving external components access to packaged functionality. APIs enable a form of abstraction: changes to the implementation remain localized if they do not alter the specified API behavior. Though each API call looks like a function, APIs are not purely functional: internal state can be modified by function calls, and these changes may affect the result of future function calls. Widely-known APIs include the Microsoft Win32 API [19] for Windows client programming and the Posix Interface [27] for Unix programming,

A *security API* is an API that uses cryptography to provide specific security functionality and implements a policy on what interactions a user may perform. Most useful 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 offer key-pair generation, fast encryption and decryption, and public key extraction but never allow private-key extraction.

A *security API attack* is a breach of a target API's stated security goals that respects the API boundary. That is, only legal sequences of legal calls are made. 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 work, 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. This methodology was developed during the analysis of several security APIs (the Visa Security Module, PKCS #11, and the IBM 4758 CCA) which we will use as a running examples. We note as evidence of our technique's effectiveness that our automated reasoning tool found not only all previously-known attacks on these APIs, but also a new variant of an attack on the IBM 4758 CCA unlikely to be discovered by the unaided human.

1.2 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 methods and automated reasoning 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 all previously-known attacks on the studied APIs and expose new variants of an attack on the IBM 4758 CCA. These new variants is surprisingly long and complex: it is unlikely that it would be found by the unaided human.

Modeling & Special Techniques. Our methodology models security APIs with a theorem prover. Specifically, we use OTTER [16]: 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 as 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 implications.

- The user can also make function calls to the API, therefore the function calls are modeled as implications that map knowledge of input to knowledge of 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 `-U(PIN)` as a goal. (The actual form of this goal is slightly more complex, for reasons explained in Section 3.3.1. However, the basic intuition is the same.)

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 implications 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 may cause the tool to miss attacks that use non-canonical forms. However, demodulation is a powerful tool, and it seems infeasible to analyze the API without it.

One of our contributions is a way of modeling an API that avoids both of these problems. We impose typing on the deduction system[1] which restricts the possible deductions to meaningful, well-formed terms. Secondly, we propose a way of representing implications 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."

Re-Derivation of Known Attacks. With these two techniques, it was possible to rediscover in an automated way all previously-known attacks [1, 3, 8, 9], all of which were

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

originally found "by hand". These attacks are similar in that they exploit the APIs' key typing system. By changing the type of a key, we are able to either recover an unknown key or establish a known key in the system.

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.

A New Attack? In the process of confirming existing attacks, we discovered three variants of a particularly complex and devastating attack on the IBM CCA.

The weakness of the CCA's lie in the fact that key typing and secret sharing use the same basic implementation: a simple XOR operation. Using a combination of secret sharing and key unwrapping API calls with cleverly chosen key values, an attacker can cause various forms of type confusion in resulting keys. With type confusion comes function confusion and the ability to export sensitive data (PINs or PIN derivation keys) that should remain wrapped.

The complexity of these attacks — about 12 steps including 8 API calls each — makes this result particularly interesting: the role of automated verification is particularly strong where the intricacy of the attack rules out unaided human discovery. Section 4.2 details the specifics of the CCA transactions and the step-by-step attack.

Upon testing one of these attacks on an actual IBM 4758, we found that one specific API call used in our attack did not function as advertised in the CCA specification documentation. This undocumented behavior makes the attack impractical but doesn't undermine the promise of automated API analysis: given a model correctly programmed from API documentation, we were able to discover a new security attack.

1.3 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 [6]. Automated verification began soon thereafter with Meadows's work on the NRL protocol analyzer [17].

In 1995, Lowe used the CSP/FDR model checker to find and provably fix [13, 14] a serious flaw in the Needham-Schroeder mutual authentication public-key protocol [20]. A theorem prover tool was first used for protocol verification by Paulson [21]. The field has since expanded [18], including complete formal modeling efforts such as Strand Spaces [26].

In 2000, Bond introduced security API attacks [1] and described a number of such attacks against the key management of IBM's CCA [12]. [2, 3, 4, 7, 9] 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 [2, 5, 9] and popular standards such as PKCS#11 [23] in [8].

1.4 This Thesis

In this thesis, we first present evidence that the theorem prover OTTER is an excellent tool for analyzing Cryptographic APIs. We then show how OTTER has re-derived known attacks on the IBM CCA, the Visa Security Module, and the PKCS #11 APIs to show that formal analysis is at least as powerful as manual analysis. We present a powerful, complex, and automatically discovered attack on the IBM CCA documented specifications (Section 4.2) that was previously unknown, as well as two attacks that work on slight variants of the actual IBM CCA. We then use the CCA API as an example with which to discuss the theorem-prover based modeling and analytical techniques we developed and used to derive the new attacks (Chapter 5). 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 (Chapter 6).

Chapter 2

The Theorem Prover Otter

In selecting a formal tool to use to analyze Cryptographic APIs, we considered several important properties:

1. Is the tool easy to use and how well supported is the technology?
2. How quickly can the formal tool find an attack or verify security?
3. How accurately and easily can the input language of the formal tool be used to model the specific functions in an API?

In the end the theorem prover OTTER proved to posses an excellent balance of these properties.

2.1 Using a Theorem Prover

The input to a theorem prover is a set of clauses (see definition 2.1.1). These clauses can specify some initial knowledge, equality relations, implications, and the negation of the theorem one would like to prove. Then, the theorem prover attempts a proof by contradiction.

As an example, say one knows two facts: that John's father is Bill, and Bill's brother is Joe. Lets also say that one knows the implication: someone's father's brother is their uncle. The first two facts represent some initial knowledge about the world, and the implication is something that can be used to derive new knowledge about the world. Finally, let the desired theorem be that John's uncle is Joe, which would be input in the negated form: Joe is *not* John's uncle. The theorem prover would then simply use the first two facts and the implication to derive a contradiction and concludes that John's uncle is Joe.

The language used above is the English translation of what the input to a theorem prover may look like, which in the case of OTTER is *first-order logic.* This is further described in Section 2.1.2.

2.1.1 Basic First-Order Logic Definitions

Before proceeding, it is necessary to understand the following first-order logic definitions from first-order logic. For more detailed definitions, see [28].

Definition. A *predicate* represents a property or a relationship over zero or more terms. An *n-ary predicate* is a predicate over n arguments. For example, the binary predicate $RAIN(x, y)$ could represent that it rained y inches on day x, and $FATHER(x, y)$ could represent the relationship that y is the father of x [28, p. 36,37]. All predicates will be written in capitalized lettering.

Definition. A *constant* is a symbol that has a fixed value. For example TMK is a constant that could represent the Terminal Master Key for a cryptographic card in an ATM. All constants will also be written in capitalized lettering.

Definition. A *variable* is a symbol that does not have a fixed value. In the clause $FATHER(x, y)$, x and y are variables.

Definition. A *term* and a *function* are defined recursively. A variable is a term. A constant is a term. A function over terms is also a term [28, p. 257]. All functions will be written in all lower case lettering.

Definition. If P is an n-ary predicate symbol, there are n variables $t_1, t_2, \cdots, t_n$, then $P(t_1, t_2, \cdots, t_n)$ is an *atom* [28, p. 257].

Definition. A *literal* is an atom or a negation of an atom. An atom is called a *positive literal* and the negation of an atom is called a *negative literal* [28, p. 266]. A literal that contains an *equality* is called an *equality literal.* Here an equality is simply a literal that relates two atoms as equal, denoted by the $=$ symbol.

Definition 2.1.1. A *clause* is a (possibly empty) set of literals. Within a clause, literals are separated by an *or* symbol (denoted by "|" in OTTER). In English, a clause gives us a set of literals, at least one of which must be true. A *positive clause* is a clause that does not contain any negative literals. A *mixed clause* is a clause that contains both positive and negative literals. A *negative clause* is a clause that contains only negative literals [28, p. 266].

2.1.2 Theorem Prover Reasoning

Theorem provers generally use an inference rule first introduced by Robinson called *binary resolution* [22] to derive new knowledge. In the most general form, binary resolution asserts that if we know that either A or B is true (denoted $A|B$) and also that $-A|C$ is true, then we can conclude that $B|C$ is true [11, p. 1119]. For clarity, the predicates A, B, and C can be thought to represent some facts in the world. Say A represents that it is raining outside, B represents that children are playing outside, and C represents that the ground is wet. Then, the first clause states that either it is raining or children are playing outside, and the second clause says either it is *not* raining or the ground is wet. After resolving the two clauses, we know that either children are playing outside or the ground is wet.

OTTER uses a form of resolution called *hyperresolution.* Hyperresolution combines the result of any number binary resolutions into a single step. By combining several binary resolutions, OTTER derives far fewer clauses than if simple resolution was used. Hyperresolution applies to a set of clauses, one of which must be a negative or mixed clause, and the remaining of which must be positive clauses. The number of negative literals in the mixed clause must equal the number of positive clauses in the input set [28, p. 75]. The positive clauses are resolved with the negative literals in the mixed clause and the result is always a positive clause. For example, if we know the following clauses[1]:

$$FATHER(Nick, John)$$

$$FATHER(Mary, Nick)$$

$$-FATHER(x, y)| - FATHER(y, z)|GRANDFATHER(x, z)$$

hyperresolution will yield the clause: $GRANDFATHER(Mary, John)$. The matching of variable terms to constant terms is called *unification* [28]. Unification can also be used to

[1]For technical reasons, OTTER prefers this disjunctive form to the equivalent FATHER(x,y) && FATHER(y,z) → GRANDFATHER(x,z)

match variable terms in different clauses. Unification was first introduced by Robinson [22]. Unifying a variable x with another term y will be written as x/y.

2.2 Benefits of Using a Theorem Prover to Model an API

2.2.1 Easy and Efficient Modeling

Previous work with theorem provers has provided excellent examples of modeling systems in first-order logic. The field of AI has already done a significant amount of well documented work modeling real world systems in first-order logic [10]. In particular, the literature describing ways to model systems with state has proved to be very valuable. One of these methods of representing state will be discussed below in Section 2.5, but in general theorem provers tend to be limited in reasoning about systems with a lot of state.

2.2.2 Complete Modeling of an Adversary

Certain theorem provers are *complete.* Completeness means that if a theorem prover terminates without finding a proof, no proof exists. In the ideal case, we would be able to carefully tune OTTER to run in a complete manner. Then, if OTTER analyzes a Cryptographic API and terminates without finding an attack, we can assert that no attack exists for our model of the API and the attacker's abilities. Without completeness, a theorem prover could only be useful in finding security holes and would never be able to *verify* that an API is secure. The issue of verifying security of a Cryptographic API is discussed in Chapter 5 and in Chapter 6.

Unfortunately, theorem provers are not, in general, guaranteed to terminate. For our purposes, however, careful specification of an API will ensure termination in theory, but memory constraints often prevent all possible proof possibilities from being considered.

2.3 Specific Benefits of Using Otter

Initial investigation into theorem provers quickly showed that OTTER is one of the best documented and supported. In fact, on several occasions it has been very easy to contact Larry Wos (one of the fathers of automated reasoning and author of several books about OTTER) as well as the author of OTTER himself: William McCune. Both were very willing to assist our research.

OTTER is a theorem prover that has all of the general benefits specified above in Section 2.2 and also offers a great deal of control over how the reasoning is performed. It is possible for the user to control everything from runtime constraints, such as memory allowances and time bounds, to the actual type of inference rules that are used to reason about the problem. Correctly setting the appropriate controls can allow OTTER to reason in a much more efficient and targeted manner. Some of the controllable settings that deserve particular attention are: weighting and lexicographical ordering, paramodulation, demodulation, and search pattern.

2.3.1 Weighting and Lexicographical Ordering

OTTER assigns every clause a *weight* [15]. The default weighting scheme assigns a clause a weight equal to the number of symbols in the clause. For example, the clause:

```
-FATHER(Jenny, John)|-GIRL(Jenny)|DAUGHTER(Jenny,John).
```

would have a weight of eight. OTTER prefers to first reason about light clauses before reasoning about heavier clauses. This heuristic is centered around the idea that it is more valuable to *know* some specific fact, than to know that one fact is true out of a set of possible facts. Recalling that clauses are composed of a set of literals, one of which must be true (see definition 2.1.1), clauses that are composed of one or two literals are closer to facts and tend to be lighter. Clauses that are composed of several literals tell us less about the world and tend to be heavier [11, p. 1120]. An OTTER user can also define their own custom weighting scheme in an effort to direct OTTER 's search based on some intuition about the problem (see [28, p. 85]).

Weighting also serves to significantly decrease the search space OTTER explores. A user can set a maximum weight for derived clauses, which will cause OTTER to instantly discard any derived clauses that exceed that weight. By adjusting the value of maximum weight, a user can adjust how much of the search space OTTER explores. Unfortunately, setting a maximum weight for clauses means that OTTER cannot possibly be complete: a proof will not be found if it requires an intermediate clause that is heavier than the maximum weight.

OTTER 's default weighting scheme is well suited to search for an attack in a Cryptographic API. In general, longer clauses tend to be more complicated. These clauses may involve the XOR of many different constants, or the encryption of a complicated message with a complicated key. By first looking at lighter clauses, OTTER will search for attacks that involve simpler terms. In practice, this strategy almost always finds an attack faster than a blind breadth-first search.

Weighting also provides an excellent heuristic for demodulation which is described in Section 2.3.3. If OTTER derives two or more clauses that are equivalent, it is much more efficient to only remember one of those clauses and discard the remaining equivalent clauses; it can choose to keep the clause that is lighter.

When two equivalent clauses have the same weight, OTTER needs to rely on *lexicographical ordering.* Clauses are ordered lexicographically based on the comparison of symbols in each clause. The default lexicographical ordering of symbols is alphabetical ordering. For example, the two clauses $FATHER(Andrew, Joe)$ and $FATHER(Andrew, Joseph)$ can be compared although they both weigh three. The first and second clause have the same first symbol $FATHER$, as well as the same second symbol $Andrew$. However, the final symbol Joe is lexicographically smaller than $Joseph$ and so the first clause can be thought of as "simpler" than the second clause. In the case where Joe refers to the same thing as $Joseph$, the second clause can be discarded because it is equivalent. Note that because every symbol is comparable to every other symbol, that lexicographical ordering provides a *total ordering* of clauses. Especially when dealing with functions such as binary XOR, the number of equivalent clauses that have the same weight can be relatively large, but with weights and the total lexicographical ordering, there is a single unique clause that has the lowest ordering. The application of this technique is described more completely in Chapter 5.

2.3.2 Paramodulation

Paramodulation is an inference rule that draws conclusions from pairs of clauses. This inference rule was first described by Lawrence Wos and George Robinson in 1968 [29]. Paramodulation requires that at least one of the clauses contains a positive equality literal. This equality clause, called the *from* clause, is used to make a substitution in the other, *into*, clause (see [28, p. 76]). A simple example would be to paramodulate *from* the clause $Joseph = Joe$ *into* the clause $FATHER(Paul, Joseph)$ to produce the clause $FATHER(Paul, Joe)$. In this case, paramodulation yields a clause that is equivalent to the original *into* clause.

Paramodulation can also be used more generally. The following example is taken from [28, p. 77]. If we paramodulate from the clause $sum(0, x) = x$ into the clause $sum(y, minus(y)) = 0$, we get the clause $minus(0) = 0$. Note that the variable x is unified with the term $minus(y)$, and y is unified with the term 0. Here the derived clause isn't exactly equivalent to the original *into* clause, but a clear result of the two input clauses. Although this example illustrates that paramodulation is a powerful inference rule, for our purposes the derivation of equivalent clauses is the most important use of paramodulation.

Paramodulation can be of great importance when dealing with binary associative and commutative functions such as XOR. When modeling the function XOR, we model it as a function that takes in two arguments. Basic properties of XOR include:

Commutativity:	`xor(x,y) = xor(y,x).`
Associativity:	`xor(x,xor(y,z)) = xor(xor(x,y),z).`
Cancellation:	`xor(x,x) = ID.`
Identity:	`xor(ID,x) = x.`

Say we know the following things:

Implication:	`-U(xor(x,xor(y,z)))|-U(y)|GOAL(z).`
Knowledge1:	`U(xor(xor(B,C), D)).`
Knowledge2:	`U(C).`

where `GOAL(z)` refers to the generic result of this implication. Although we have enough information to conclude `GOAL(D)` and `GOAL(B)`, OTTER will not be able to unify the implication clause with the first knowledge clause. However, OTTER can paramodulate *from* the association clause *into* the implication clause to produce the equivalent rule:

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

At this point unification and hyperresolution can occur, unifying `x/B`, `y/C`, and `z/D` to conclude `GOAL(D)`. To derive `GOAL(B)`, OTTER must do a series of six paramodulations:

```
  Original:  -U(xor(x,xor(y,z)))|-U(y)|GOAL(z).
  Commute:   -U(xor(x,xor(z,y)))|-U(y)|GOAL(z).
Associate:   -U(xor(xor(x,z),y))|-U(y)|GOAL(z).
  Commute:   -U(xor(xor(z,x),y))|-U(y)|GOAL(z).
Associate:   -U(xor(z,xor(x,y)))|-U(y)|GOAL(z).
  Commute:   -U(xor(z,xor(y,x)))|-U(y)|GOAL(z).
Associate:   -U(xor(xor(z,y),x))|-U(y)|GOAL(z).
```

Now unification can occur between the new implication clause and the second knowledge clause unifying `z/B`, `y/C` and `x/A`. The result of this hyperresolution `GOAL(B)`[2]. As this example points out, paramodulation can take up a significant amount of time to derive seemingly simple results. A detailed discussion of this problem appears in Chapter 5.

In the above examples, paramodulation is as an assistant for unification. Sometimes the theorem prover knows a clause that is *equivalent* to a clause required for unification, and paramodulation can change the form of the clause to allow for unification.

2.3.3 Demodulation

The goals of demodulation are to simplify and canonicalize clauses. For our purposes, we can deal with these goals simultaneously. We use demodulation to canonicalize a target clause into the simplest form. Demodulation is very similar to paramodulation. Demodulation was first described by Wos, Robinson, Carson, and Shalla in 1967 [30]. Similar to paramodulation, demodulation acts on two clauses, one of which must be a positive *unit* equality clause. However, demodulation will only act to simplify a clause, while paramodulation could derive a clause that is heavier than the original *into* clause. The clause derived from demodulation will be equivalent to the original *into* clause, but have the lowest possible weight, and the lowest possible lexicographical order for that weight. An additional restriction is that the equality clause used must come from a special set of clauses called the *demodulators*. Finally, this simplification rule *replaces* the *into* clause with the derived clause. This resulting clause is necessarily unique because any two clauses that have equal weight and lexicographical ordering must consist of the same symbols in the same order. In other words, if two clauses $C1$ and $C2$ are equivalent, they will demodulate to the same

[2]Note that the knowledge clause could also have been paramodulated to fit the implication clause, but this specific paramodulation would not necessarily be as widely usable as paramodulating the implication itself.

clause $C3$. Demodulation is applied to every initial clause and to every clause that is derived by the active inference rule(s) (in our case, hyperresolution).

For example, if the target clause is:

$$FATHER(FATHER(Joe, John), Jerry)$$

and we have the demodulator:

$$FATHER(FATHER(x, y), z) = GRANDFATHER(x, z)$$

the target clause will be demodulated to:

$$GRANDFATHER(Joe, Jerry)$$

However, the reverse transformation is not possible. If the target clause is:

$$GRANDFATHER(Joe, Jerry)$$

and the demodulator is the same, the target clause will not be changed because the equivalent clause:

$$FATHER(FATHER(Joe, John), Jerry)$$

has weight five, while the target clause has weight three.

In another example, say we have the target clause $xor(xor(A, B), C)$ and a demodulator $xor(xor(x, y), z) = xor(x, xor(y, z))$. The target clause will be demodulated to: $xor(A, xor(B, C))$. Although the demodulated clause and the original target clause both have the same weight, the demodulated clause has lower lexicographical ordering because the first non-matching term $xor(A, B)$ has a higher order than the term A (we define function terms to have a higher lexicographical order than non-function terms).

2.3.4 Search Pattern

OTTER has two possible search methods of particular interest. The first is a weight based search method that reasons about the lightest clauses first as mentioned in Section 2.3.1. The second is a breadth first search. The first method is almost always faster at finding a proof (an attack) because simpler terms tend to be more likely to be involved in an attack. However, the second search method is very important. In the event that an API is too large to exhaustively explore, it is more useful to be able to claim that we have searched through all possible attacks that involve k or fewer steps, than to claim that we have searched through all possible attacks that involve clauses with less than n symbols.

2.4 The Selection of Otter

OTTER clearly meets all three of the criteria outlined at the beginning of this chapter. It is extremely well supported both in literature, as well as via direct contact with experts. In addition, the optimizations and controls that are available allow for very efficient searching. Finally, an API can be accurately and easily transformed into first-order logic. All of these reasons led us to begin experimenting with OTTER .

2.5 Modeling an API Using Otter

Modeling an API in first-order logic for OTTER can be tricky. This Section will cover basic modeling techniques that deal with state and knowledge, while Chapter 5 will deal with more specific and advanced techniques for efficiently modeling an API.

In modeling an API, there are only two things of which we need to keep track. The first is a change in the state of the device we are exploring. If a function call, for example, changes the master key of a device, that function call can have an impact on how the device behaves from that point forward. Certain states may restrict what function calls are allowed on particular inputs, and to accurately specify an API, the state of a device needs to be modeled. If we ignored the state of the device, we may find an attack that uses a function call that is only possible in state S, followed by a function call that is only possible in state S'. However, if it is impossible to get the device from state S to state S', the attack is not valid.

The second thing that we need to keep track of is the change in attacker's knowledge. Every function call that gives the attacker some knowledge will allow the attacker to potentially use that knowledge to gain more knowledge, and eventually learn something he is not supposed to know.

The fundamental unit of the model is the *user knowledge predicate* `U`. That is, user knowledge of a value `FACT` would be represented by setting `U(FACT)` to true. Merely placing `U(FACT)` in the model will suffice to specify that it is true.

To model state, we use the same technique as is often used in the field of AI. We start by using the binary predicate $T(x, y)$ to denote that x is true in state y. For example, we model truths in the original state with: $T(x, \mathit{Initial})$. Then, when a state change happens due to some action S, we model truths in that new state by $T(x, S(\mathit{Initial}))$. A truth x may be true before a state transition, but false after the transition. On the other hand, most of the other truths may be unaffected by that given transition. This is called the *Frame Problem* [10, p. 271] because only some truths are affected while the rest of the world stays the same [3]. A classic problem with state is modeling blocks on top of a table [10, p. 28].

Say there are originally 4 blocks on the table. Let $On(x, y)$ denote that a block x is directly on top of some object y. Initially, say all the blocks are on the table, so you have:

```
T(On(B1,Table),Initial).
T(On(B2,Table),Initial).
T(On(B3,Table),Initial).
T(On(B4,Table),Initial).
```

The state change that moves `B1` onto `B2` adds the relationship:

```
T(On(B2,B1),Put(B2,B1,Initial)).
```

Here, `Put(x,y,z)` is a ternary predicate specifying the state where block `x` has been moved on top of `y` from the state `z`. In addition to the direct result of moving a block onto another block, we need to also know that all previous unaffected relationships still hold in the new state `Put(B2,B1,Initial)`. So, we say that any truths that are unaffected by the state transition, are still true. Specifically:

```
T(On(B1,Table),Put(B2,B1,Initial)).
T(On(B3,Table),Put(B2,B1,Initial)).
T(On(B4,Table),Put(B2,B1,Initial)).
```

2.6 Other Tools Considered

Model checkers are another class of formal tool that was considered for the project. Model checkers take in a description of a State Machine and attempt to verify that user defined

[3]This term comes from drawing cartoons where the background frame of a cartoon is often stationary while the foreground has movement.

invariants hold by searching through reachable states. It is very possible that tools such as these will prove more effective in the future analysis of cryptographic APIs. Indeed, Chapter 6 discusses a potential use for model checkers in analyzing APIs with state. At the same time, the purpose of our research is to show that formal tools *can* be used for analyzing Cryptographic APIs, and at this point finding the best tool is less important than finding an effective tool.

Chapter 3

Formal Derivation of Known Cryptographic API Attacks

We have specified several Cryptographic APIs in the first-order logic and input them into OTTER . These specifications have been successfully used to find known attacks on these APIs. The particular way in which an API is specified before fed into OTTER , as well as the specific mode you run OTTER in, has proven to make a very significant difference both in terms of the running time, and completeness of OTTER .

3.1 The Visa Security Module and the IBM 4758 CCA

The Visa Security Module (VSM) and the IBM Common Cryptographic Architecture 4758 (CCA) API are hardware-based architectures used to facilitate banking transactions. Both devices have similar security goals, and have similar restrictions to enforce those goals. These basic principles are described in this Section and specific attacks on each device are discussed in more detail below in Sections 3.2 and 3.3.

The VSM and IBM CCA enable secure banking operations, and are used in *Automatic Teller Machine (ATM)* networks for two important tasks: secure bank-to-ATM/bank-to-bank communication and *Personal Identification Number (PIN)* verification.

Both the VSM and the IBM CCA accomplish these tasks using:

- a symmetric-key encryption scheme,
- a unique master key,

- secret sharing [24],
- *key wrapping*, and
- *key typing*

Key wrapping consists of protecting a sensitive symmetric key by encrypting it with another key. The symmetric-key-as-plaintext is said to be "wrapped" under the key that encrypts it. This key wrapping allows the module to maintain no more than a single master key KM in secure storage. If the module needs to work with another key k, k is wrapped under the master key (i.e., encrypted as $\{k\}_{KM}$) and stored in insecure memory external to the device. Once keys are wrapped, the device 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.

Finally, *key typing* provides a mechanism for specifying a key's precise usage domain. A $DATA$ key, for example, can be used to encrypt and decrypt data in a customer transaction. However, a $DATA$ key should not be used to wrap other keys: allowing this wrapping would allow an attacker to easily unwrap the key and obtain it in the clear by issuing a simple decrypt call. The VSM and CCA APIs are designed to prevent this (among other things).

Along with preventing fraudulent behavior, both hardware devices need to:

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

The devices attempt to meet these goals by issuing each customer a Personal Identification Number (PIN) which is *cryptographically* derived from their Primary Account Number (PAN)[1]. That is, the bank randomly generates a *PIN-derivation key* P, which will be used to generate PINs for all customers. Then, a customer's PIN is generated by encrypting that customer's PAN with this key. We denote the encryption operation as:

$$PIN := \{PAN\}_P \tag{3.1}$$

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

(Technically, the PIN is only the first few digits of this encryption, but this will make no difference to our model. Also, some banks allow their customers to "change" their PIN, which only means that a PIN *offset* is stored in the clear on the bank card.)

Thus, the banking hardware devices must:

1. compute a customer's PIN by encrypting the PAN of the swiped card under the PIN-derivation key, and
2. compare this true PIN with the number punched in by the human customer.

If these two values match, the device will allow the transaction to proceed. To protect the customer and to limit fraud, the device must prevent the customer's true PIN and the PIN-derivation key from ever leaving the card. 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.

3.2 An Attack on the VSM

3.2.1 Overview of Relevant VSM Policy and Functionality

In order for banks to set up a new remote ATM, they have to somehow securely transfer secret keys to the VSM in the new ATM. To accomplish this task requires a new set of functions. When a bank initially deploys an ATM, they need to establish a shared secret key. The bank accomplishes this task by creating two key shares $KMT1$ and $KMT2$ and sending each share to the new ATM. Each share would be entered separately by a different person, and then XORed together to create a whole key KMT [3, p. 5]. This initial shared secret key is called the *terminal master key* or KMT and can then be used to obtain other keys [3, p. 4]. Once the KMT is established, various other *master keys* can be put on the remote VSM. Because the VSM has limited storage, other keys (non-master keys) are then stored off of the device, but encrypted under a master key. The particular master key used to encrypt another key defines the type of that key. For example, one type of master key is the *master terminal communications key*, or TMC. All keys that are stored off the device encrypted under TMC are considered *terminal communications keys.* The type of a key determines the functionality of that key and how securely the VSM protects that key. The attack described below takes advantage of weak security policy associated with terminal communications keys.

A *terminal communications key*, from now on referred to as the KC, is used to compute the MAC of messages received by the VSM. This key is not viewed to be of critical importance because the cleartext of the message received is assumed to be known and so KC can be used to encrypt and decrypt arbitrary data [3, p. 6]. These functions will be referred to as "Encrypt with Communications Key"

$$m, \{KC\}_{TMC} \rightarrow \{m\}_{KC}$$

and "Decrypt with Communications Key"

$$\{m\}_{KC}, \{KC\}_{TMC} \rightarrow m$$

By contrast, the PIN derivation key can't be used to encrypt arbitrary data, otherwise anyone could calculate a PIN number from a primary account number. Because a terminal communications key KC is viewed as non-critical, there is a transaction that allows it to be entered into the VSM in the clear, and the key is then immediately encrypted under the appropriate master key to yield $\{KC\}_{TMC}$. This function will be called "Create Communications Key".

$$KC \rightarrow \{KC\}_{TMC}$$

In order for KC to be useful for computing the MAC of messages sent between VSM modules, it needs to be distributed. As such, there is a second function that allows a terminal communications key to be re-encrypted under any key of type "terminal master key". This function will be called "Re-wrap Communications Key".

$$\{KC\}_{TMC} \rightarrow \{KC\}_{TMK'}$$

where TMK' is an arbitrary terminal master key [3, p. 6].

The VSM only stores nine master keys, which only allows for nine different types of keys. As such, the PIN derivation key P and the terminal master key KMT are both considered the same type. This may seem reasonable because both P and KMT are of similar critical importance.

3.2.2 The Attack

Given the outlined facts above, we can now mount an attack on the VSM.

1. Enter in the PAN of an ATM card as a terminal communications key by calling "Create Communications Key". This yields $\{PAN\}_{TMC}$.

2. Then, the PAN can be re-wrapped under the "terminal master key" P by calling "Re wrap Communications Key". This yields $\{PAN\}_P$.

3. The attacker now knows the PIN number for the account PAN.

For a more detailed description of this attack, see [3].

3.2.3 Verifying the Attack

A first-order logic model of the VSM API was quickly written based on a single spec sheet describing the transactions. Once the initial model was created, OTTER quickly found the attack described after generating about 7.4 billion clauses and keeping about 11 thousand clauses. Each generated clause is the result of using existing knowledge and applying an online transaction. Most of the clauses are discarded immediately because they exceed the weight limit as described in Section 2.3.1. Because of variance in actual timing data on different machines under unknown load, the number of generated clauses and kept clauses is the standard measure of how fast OTTER reasoned about a problem. For reference, however, this attack was found in about four minutes of wall-clock running time.

Using a later revision of the model using techniques described in Chapter 5, the number of generated clauses was reduced to 100 clauses generated and 105 clauses kept. Not only did this later model run much faster, it also included off-line abilities such as the ability to bitwise XOR two strings and to encrypt one string under another string. In addition, OTTER was run in a mode that we believe to be complete. If we could prove that OTTER was indeed complete, it would allow us to assert that no other API attack exists on the VSM that uses the functions we defined in our model.

3.2.4 Conclusions

Modeling the VSM and successfully finding a known attack served three main purposes:

1. This immediate and satisfying result partially motivated us to focus more attention on OTTER as a potentially effective tool for analyzing Cryptographic APIs.

2. Because no further attacks were found, combined with the fact that OTTER was running in what we believe to be in a complete mode, allows us to believe that our VSM model has exactly one API attack. As will be pointed out in Chapter 5, the assertion that a model is in fact a replication of the actual abilities of an adversary is a problem with no immediate solution. Chapter 6 will discuss the the necessity to *prove* that OTTER is running in a complete mode.

3. Revisions of the VSM model using coding strategies developed later served as a powerful benchmark that proved our new strategies are effective.

3.3 The 4758 CCA Co-Processor

3.3.1 Overview of the 4758

In order to transmit sensitive data from one module to another, the IBM CCA requires that two communicating 4758's must share at least one communication key *KEK* (Key-Encrypting Key). *KEK* is stored locally at each end in external memory, wrapped under that CCA's master key. Once each CCA has a copy of *KEK*, communication between the two can easily be secured. Much like the VSM, simple *secret sharing* bootstraps the secure transmission of *KEK* from one CCA module to another: 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.* The key parts, when XORed together, produce the original key KEK. Each key part is sent by individual courier and recombined using special CCA API calls. 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.

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

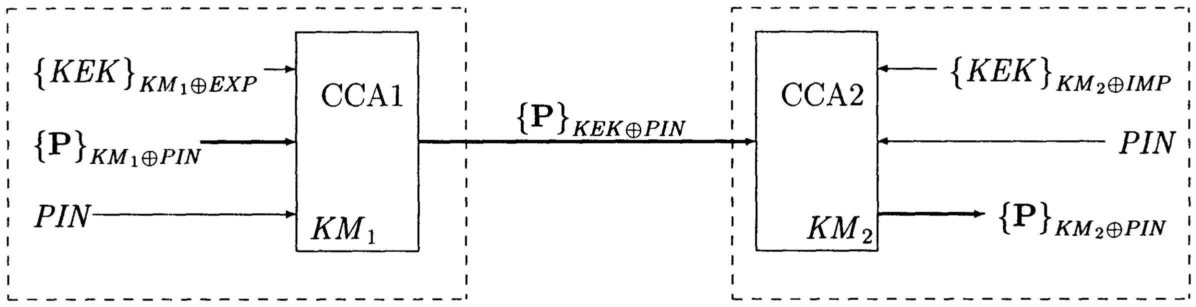
$\{KEK\}_{KM_1 \oplus EXP}$
$\{P\}_{KM_1 \oplus PIN}$
PIN
CCA1
KM_1
$\{P\}_{KEK \oplus PIN}$
CCA2
KM_2
$\{KEK\}_{KM_2 \oplus IMP}$
PIN
$\{P\}_{KM_2 \oplus PIN}$

Figure 3-1: Transfer of P from CCA-1 to CCA-2 via *KEK*

the PIN-derivation key P) can be wrapped with it and sent from one to the other. 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,
- 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 export keys wrap the key to be secured. For example, KEK will be generated by one CCA and split into three parts. The generating CCA might also designate *KEK* as an exporter key. After this key is reconstructed from the three parts, the receiving CCA will most likely designate *KEK* as an importer key. Thus, the first CCA can generate a new key $K1$ and export it by wrapping it with the exporter key *KEK* to yield $\{K1\}_{KEK}$. Likewise, the second CCA can import it by decrypting (unwrapping) $\{K1\}_{KEK}$ 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.

Because much of the 4758 CCA transaction set (as well as our attacks) concern this dedication of keys to roles, 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

$\{P\}_{KM}$ can be 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 types, but we need only consider four:

- *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 $\{P\}_{PIN \oplus KM}$. 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 $\{K\}_{DATA \oplus KM}$ is stored externally to the 4758. Similarly, if the ciphertext $\{K\}_{IMP \oplus KM}$ 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. For example, a key K can be both of type EXP and of type PIN. This key K would be equal to $\{K\}_{EXP \oplus PIN \oplus KM}$.

Note that the method for recombining key parts also uses the XOR operation. This dual use of XOR will be basis of our attacks. Before we describe the attacks, though, we present the required API commands in detail.

3.3.2 4758 Functions

The 4758 CCA offers dozens of functions. Here, we present only those functions necessary for our attack. Unsurprisingly, these functions also concern key management as described in the previous sub-section.

Encrypt with Data Key. A $DATA$ key can be used to encrypt arbitrary data. This call takes as input an arbitrary message m and a data key, and produces the corresponding ciphertext:

$$m, \{K\}_{KM \oplus DATA} \rightarrow \{m\}_K \tag{3.2}$$

Decrypt with Data Key. Conversely, any ciphertext can be decrypted with a *DATA*key:

$$\{m\}_K, \{K\}_{KM \oplus DATA} \rightarrow m \tag{3.3}$$

Key Import. If $K1$ is an importer key, then any key $K2$ wrapped with $K1$ can be imported. Note the difference between $K1$, a key wrapped under the CCA's master key KM and typed as an importer key, and $K2$ wrapped under $K1$ and typed as T. The type T of $K2$ must be known in advance and provided as an input to this function call:

$$T, \{K1\}_{KM \oplus IMP}, \{K2\}_{K1 \oplus T} \rightarrow \{K2\}_{KM \oplus T} \tag{3.4}$$

Key Export. This is the sister function of Key Import. If $K2$ is wrapped and typed as T and $K1$ is an exporter key, then this call will produce $K2$ wrapped under $K1$ and of type T. Again, the type T must be provided as input:

$$T, \{K1\}_{KM \oplus EXP}, \{K2\}_{KM \oplus T} \rightarrow \{K2\}_{K1 \oplus T} \tag{3.5}$$

The intent is for this wrapping to be sent to another CCA which has $K1$ as an importer key.

Key Part Import. As mentioned above, the original key shared between two CCAs, KEK, is split into three parts, $K1$, $K2$ and $K3$. These parts are sent in the clear. The Key Part Import function is used to merge them at the other end.

We will only describe the use of this function to import the third part. After the first two parts (say, $K1$ and $K2$) are entered, their XOR sum is wrapped under the receiving CCA's master key KM. Furthermore, $K1 \oplus K2$ will have already been typed as T, the intended type of the completed key. However, the *key-part bit*, a special bit, will have been set to indicate that this is only a partial key and not yet completed. (We denote this via the function $kp()$ on the control vector.)

When the third part is entered, the completed key $KEK = K1 \oplus K2 \oplus K3$ will be produced, wrapped under KM as type T, with the key-part bit turned off.

$$K3, T, \{K1 \oplus K2\}_{KM \oplus kp(T)} \rightarrow \{K1 \oplus K2 \oplus K3\}_{KM \oplus T} \tag{3.6}$$

Key Generate. The "Key Generate" function is used to generate matching pairs of import and export keys. The key is randomly generated inside the 4758 and revealed only once wrapped under the master key. One use of this function, as published in the 4758 manual, is to allow a user to change the key type of another key. The initial step is to:

> Use the Key_Generate verb to generate an IMPORTER/EXPORTER pair of (Key Encryption Keys) with the KEY-PART control vector bit set on. Then use the Key_Part_Import verb to enter an additional key part... [25, p. 407]

This function allows a CCA to generate a new key-encrypting key *KEK2* key. The importer key version is sent to another CCA (via a *KEK* wrapping), while the exporter key is kept locally. *KEK2* becomes a shared key between the two CCAs with appropriate typing to export keys on one end and import them on the other.

$$\begin{gathered} \mathit{KEK2} := \mathit{RandomKey}() \\ () \rightarrow \{\mathit{KEK2}\}_{KM \oplus kp(IMP)}, \{\mathit{KEK2}\}_{KM \oplus kp(EXP)} \end{gathered} \tag{3.7}$$

3.3.3 Preliminary Conditions

The following attack relies upon a number of assumptions about the environment in which the 4758 is operated, manifested in the initial knowledge, and of course upon the specifics of the model. We assume an exchange of a key encryption key, KEK, for the first time between two banks, followed by sharing of the source bank's PIN-derivation key, P. KEK is transported by multiple couriers and entered using the Key Part Import command, while P is received in encrypted form, wrapped with KEK.

Our model considers the scenario where type-casting is still used by the recipient bank (i.e. the improved integrity control on key part entry in version 2.41 of the CCA has not been enabled in the access control configuration).

We assume that the first two couriers have correctly delivered their key parts $K1$ and $K2$, to begin the construction of a $KEK = K1 \oplus K2 \oplus K3$. Thus, $K1 \oplus K2$ (more conveniently written as $KEK \oplus K3$) exists in wrapped form as:

$$\{KEK \oplus K3\}_{KM \oplus kp(IMP)}$$

In our attack, the third courier is an accomplice.

3.3.4 The Known Attack

The previously known attack on the 4758 CCA type casts the existing PIN Derivation Key P to be of type $DATA$, and then use that key to encrypt any Primary Account Number. Recall that the encryption of a PAN is the PIN number for that account (see Section 3.1). That is the attacker knows $K3$ and $\{P\}_{KEK \oplus PIN}$.

1. The third courier delivering $K3$ as the final part of the initial importer key KEK delivers $K3 \oplus DATA \oplus PIN$ instead. This outputs: $\{KEK \oplus DATA \oplus PIN\}_{KM \oplus IMP}$ instead of the expected $\{KEK\}_{KM \oplus IMP}$. In English, this yields the key $KEK \oplus DATA \oplus PIN$ as an importer key.

2. The attacker now waits for the originating bank to send the encrypted version of the PIN-derivation key P across the wire: $\{P\}_{KEK \oplus PIN}$.

3. The attacker now calls Key Import using the importer key from the first step to import the Pin Derivation Key under the claimed type $DATA$. Note that: $\{P\}_{KEK \oplus DATA \oplus PIN \oplus DATA}$ $= \{P\}_{KEK \oplus PIN}$ which is assumed to be known. The result is: $\{P\}_{KM \oplus DATA}$, or the Pin Derivation Key as a Data Key.

4. Now, any PIN number can be derived from a Primary Account Number by feeding the Primary Account Number into the card as data to be encrypted under the $DATA$ key P.

3.3.5 The Verification of the Attack

Once the specification had been written, the attack was initially verified by generating 5.45 billion clauses and keeping 8312 clauses. Later, we trimmed the search down to generating only 11,379 clauses clauses and keeping 5,456 clauses. In wall clock running time, this was an improvement of about a thousand fold. The original attack took around sixteen minutes to run, while this attack took a mere second to run. Note that the original search was conducted with OTTER running in a non-complete mode. In other words, it would have been possible for OTTER to miss the attack, but the necessary clauses happen to be sufficiently simple that they were not discarded as too heavy (see Section 2.3.1). Most of the simplification involved efficiently dealing with the off-line ability of the attacker to preform XOR and using demodulation to eliminate redundant clauses. In both forms of the attack, we had to limit the attacker to only using keys that were not the result of a previous encryption to perform new encryptions. For a justification of these approaches, see Chapter 5.

3.4 Public-Key Cryptography Standards #11

As stated in [23, pg. 12]:

> (PKCS #11) was intended from the beginning to be an interface between applications and all kinds of portable cryptographic devices, such as those based on smart cards, PCMCIA cards, and small diskettes.

At the heart of PKCS #11 is an API designed to verify the digital identities of users. A person's digital identity is usually in the form of a private key that is stored on a smart card (or similar device) she carries around much like a drivers license. Then, this smart card can interface with various hardware systems to positively identify the user. PKCS #11 describes a standard for the interaction between hardware systems, also known as tokens, and the associated software that can leverage that identity to perform some application. The actual token can do any number of things, and in the words of Jolyon Clulow:

> a token is a device that stores objects (e.g. Keys, Data and Certificates) and performs cryptographic functions [8, pg. 412].

As an example, one could design a hardware/software device for secure e-mail. The basic device could take in a digital identity card as well as a password to grant a user the appropriate level of security clearance to access an encrypted e-mail database. The device would need to be able to generate key pairs to issue to users for authentication. To manage these keys, it should be possible to export a wrapped key to a user's ID card, and import a key from a user's ID card. A normal user, once identified, should be able to only access their stored e-mail, and should also would be able to sign/encrypt e-mail using their unique private key. This level of complexity requires that keys be explicitly typed. Recall that a key should not be able to both decrypt arbitrary data, and wrap a key for exporting (see Section 3.1).

In general, PKCS #11 is not a strict API as the other devices discussed in this thesis. PKCS #11 describes *potential* functions a token *could* implement. In addition, the specification places as few restrictions on the functions as possible, leaving it up to the implementors of a token to specify the exact behavior of those functions. On one hand, this places a large burden on implementors of PKCS #11 devices. A device could meet the PKCS #11 specification, but offer very little real security. On the other hand, this allows the standard to be as widely applicable as possible. Finally, although the attack described below *could* work on an implementation of PKCS #11 because the functionality required to perform the attack is within the standard, it is likely that good implementations of PKCS #11 will resist this attack.

3.4.1 The Attack

The main security of PKCS #11 is based on various keys stored on a token along with the attributes of that key. The attribute vector of a key serves the same purpose as key typing served in the VSM: they are necessary in order to deal with key management and the normal cryptographic functionality of the token. The attributes specify if a key can:

- Wrap another key for exporting (encrypt another key).
- Be wrapped for exporting.
- Encrypt data.
- Decrypt data.
- Sign data.
- Be exported in the clear.
- Sign data.
- Verify data.

Unlike the VSM which has very little state, a PKCS #11 token has an attribute vector associated with each key in the device. In order to manage keys, PKCS #11 describes a function SET_ATTRIBUTE that allows the attribute vector for a key to be changed. This function takes in a key, an attribute, and a new value and alters the attribute vector of that key.

An attacker could:

1. Call SET_ATTRIBUTE on a key $K1$ to alter its attribute vector to allow it to wrap another key and also decrypt data.

2. Then a target key's attribute vector can be changed to allow it to be wrapped and extracted by another call to SET_ATTRIBUTE.

3. The target key $K2$ can be exported wrapped by $K2$ to yield $\{K2\}_{K1}$.

4. Finally, the "message" $\{K2\}_{K1}$ can be decrypted by the key $K1$ to yield $K2$ in the clear.

This attack was originally conceived by Jolyon Clulow in his paper [8, pg. 416, 417].

3.4.2 Verifying the Attack

Our specification of a PKCS #11 token was our first exposure to modeling state in an API. Our initial model used a method first described in section 2.5. In particular, the predicate `ATT(x,y)` denoted the key `x` has attribute `y` set to true. For example, if the constant `DES1` represented a DES key, and the constant `WR` represented that a key could wrap another key, then `ATT(DES1,WR)` would represent that the key `DES1` could wrap another key. However, because of state, we need to know in which states a particular attribute is true. The predicate `T(x,y)` was used to denote that `x` is true in state `y`. For example, `T(-ATT(DES1,WR), INITIAL)` represents that in the Initial state the key `DES1` can *not* be used to wrap other keys. Changing that attribute constitutes a state change and can be represented by the predicate: `T(Att(DES1,WR), TOGGLE(ATT(DES1,WR),INITIAL))` where the predicate `TOGGLE(ATT(DES1,WR),INITIAL)` represents the new state that results from toggling the value of the wrap attribute for the `DES1` key from the Initial state.

After the partial specification of PKCS #11 was complete, OTTER was used to verify the attack described above in Section 3.4.1. OTTER generated 55,170 clauses and kept 5,249 clauses in finding the attack. By using an idea that is further described in Chapter 5, we were able to trim down the search to 22 clauses generated and 44 clauses kept. The basic idea is to essentially ignore state and search for attacks, then verify that the generated attacks actually work in the true model of the API.

3.4.3 Conclusions

State transitions greatly increase the size of the search space. In particular, we must re-derive truths in every reachable state. PKCS #11 has 2^{n*k} reachable states where n is the number of attributes per key, and k is the number of keys in the device, making the search tree significantly larger. The most important benefit gained from exploring PKCS #11 was to expose ourselves to the potential problems that state presented, and force us to formulate ideas on how to manage state.

Chapter 4

Formal Derivation of New Cryptographic API Attacks

4.1 Attacks on a Flawed Model of the 4758 CCA

Once the 4758 CCA code had been optimized as much as possible using the techniques described in Chapter 5, we tried to run OTTER in what is believed to be a complete mode. Using this mode, and much more efficient reasoning, we were able to discover three new attacks on the 4758 CCA. Although two of the attacks are protected against in the latest implementation of the 4758 CCA assuming customers follow the suggested security measures, the attacks are much more complex than previously discovered attacks, and offer strong evidence that formal methods will aid humans in their search for new attacks on a Cryptographic API. The first two attacks were found based on a slightly flawed model that was previously considered correct, and had escaped the human eye.

We assume the same assumptions as given in Section 3.3.3, and also that the attacker can call Key Part Import multiple times outside of the context of actually importing the third key part $K3$.

All attacks involve recovering the PIN-Derivation Key P in the clear. Along the way, when P is also established as an Exporter key, *any* key on the device can be exported under P and decrypted in the clear. The first attack proceeds as follows:

1. Use the function Key Part Import with a claimed Type IMP (importer key), clear key part $K3 \oplus kp(DATA \oplus PIN)$, and the partially assembled key $\{KEK \oplus K3\}_{kp(IMP) \oplus KM}$.

The return is: $\{KEK \oplus kp(DATA \oplus PIN)\}_{IMP \oplus KM}$, or in other words, the key $KEK \oplus kp(DATA \oplus PIN)$ as an importer key.

2. Repeat the above step 1, but use the key part $K3 \oplus PIN \oplus EXP$. The return this time is: $\{KEK \oplus EXP \oplus PIN\}_{IMP \oplus KM}$. In English, the key $KEK \oplus EXP \oplus PIN$ as an importer key.

3. Repeat the above step 1 one more time but with the key part $K3 \oplus IMP \oplus PIN$. The return this time is: $\{KEK \oplus IMP \oplus PIN\}_{KM \oplus IMP}$, the key $KEK \oplus IMP \oplus PIN$ as an importer key.

   We now have 3 slightly different importer keys.

4. Call Import Key with the claimed type EXP, the key to be imported $\{P\}_{KEK \oplus PIN}$, and the import key from 2. Note that due to XOR canceling, $\{P\}_{KEK \oplus PIN} = \{P\}_{KEK \oplus PIN \oplus EXP \oplus EXP}$ as required by the function. The return of the function call is: $\{P\}_{EXP \oplus KM}$, P as an exporter key.

5. Call Import Key again with the same key to be imported as step 4 but with the claimed type is IMP instead of EXP, and the importer key from step 3. The return is: $\{P\}_{IMP \oplus KM}$, P as an Importer key.

6. Again call Import key using the claimed type $kp(DATA)$, the importer key from step 1, and the same key to be imported as in step 4. The return is: $\{P\}_{kp(DATA) \oplus KM}$, the PIN-derivation key as a key part and a data key.

7. Call Key Part Import once again with the claimed type $DATA$, key part of any known string such as EXP, and the partial key from the previous step 6. The return is: $\{P \oplus EXP\}_{DATA \oplus KM}$, a data key.

8. Call Key Export using the result of step 4 as both the key to be exported, and the exporter key, with claimed type EXP. This results in: $\{P\}_{P \oplus EXP}$.

9. Call Decrypt Data using the result from the previous step 8 and the Data Key from step 7. This returns P in the clear.

This attack is prevented by the real API. Key Import can not be called on partial keys as it is in step 6. Secondly, the ability to import a partial key and the ability to *complete* it, which is necessary before the key can be used, can be assigned to different people. In fact, IBM suggests that these tasks are assigned to separate people. Even though the underlying model of the API was flawed, this complex attack clearly shows that the *strategy* of using OTTER to find attacks is powerful.

The second attack proceeds as follows:

1. Call Key Part Import with claimed type IMP, clear key part $DATA \oplus K3 \oplus PIN$, and partially assembled key $\{KEK \oplus K3\}_{kp(IMP) \oplus KM}$. The return is: $\{DATA \oplus KEK \oplus PIN\}_{IMP \oplus KM}$, an importer key.

2. Again call Key Part Import with the same arguments as above in step 1 except with the clear key part $EXP \oplus K3 \oplus PIN$. The return value is: $\{EXP \oplus KEK \oplus PIN\}_{IMP \oplus KM}$, another importer key.

3. Call Key Import with the importer key from step 1 above, claimed type $DATA$, and the key to be imported $\{P\}_{KEK \oplus PIN}$. The return is: $\{P\}_{DATA \oplus KM}$, or P as a $DATA$ key.

4. Again call Key Import but with a claimed type EXP, the importer key from step 2, and the key to be imported $\{P\}_{KEK \oplus PIN}$. The return is: $\{P\}_{EXP \oplus KM}$, or P as an EXP key.

5. Call Key Part Import with the clear key part EXP, the "partially assembled" key $\{P\}_{DATA \oplus KM}$ from step 3, and the claimed type $DATA$. The return is: $\{P \oplus EXP\}_{kp(DATA) \oplus KM}$.

6. Call Key Part Import to complete the key generated in the previous step 5. The return is: $\{P \oplus EXP\}_{DATA \oplus KM}$.

7. Call Key Export using the result of step 4 as both the key to be exported, and the exporter key, with claimed type EXP. This results in: $\{P\}_{P \oplus EXP}$.

8. Call Decrypt Data using the result from the previous step 7 and the Data Key from step 6. This returns P in the clear.

This second attack is again prevented in the real API. First of all, Key Part Import requires that the claimed type of the part being imported is a Key Part, which is violated by step

5. Secondly, again this attack requires both the importing of partial keys as well as the completion of the assembled keys, which may require two different people.

4.2 Formally Found API Flaw in the IBM 4758

Our final new attack variant also violates the above-stated security goal: the attacker obtains the PIN-derivation key P from a small constant number of interactions with the CCA. The attacker can then derive any user PIN off-line. Because the 4758 CCA does not fully conform to its advertised behavior, our final tests on a real 4758 cannot confirm our analytical results. However, at the very least, this attack constitutes a major breach of the CCA specifications.

We assume the same assumptions as given in Section 3.3.3.

1. The third courier delivers $K3 \oplus PIN$ instead of $K3$. Using "Key Part Import":

$$K3 \oplus PIN, IMP, \{KEK \oplus K3\}_{kp(IMP) \oplus KM \oplus} \xrightarrow{(3.6)} \{KEK \oplus PIN\}_{IMP \oplus KM}$$

2. The attacker calls "Key Generate" and obtains, for some random $K4$ determined internally by the 4758 (and unknown to the attacker):

$$\emptyset \xrightarrow{(3.7)} \{K4\}_{KM \oplus kp(IMP)} \quad \text{and} \quad \{K4\}_{kp(EXP) \oplus KM}.$$

3. The attacker completes the keys generated in step 2 with "Key Part Import" operations: the importer key is completed with the *IMP* control vector submitted as if it were a key part, while the exporter key is completed with the *DATA* control vector as if it were a key part. The keys obtained are complete importer and exporter keys:

$$\begin{aligned} IMP, IMP, \{K4\}_{kp(IMP) \oplus KM} &\xrightarrow{(3.6)} \{K4 \oplus IMP\}_{IMP \oplus KM} \\ DATA, EXP, \{K4\}_{kp(EXP) \oplus KM} &\xrightarrow{(3.6)} \{K4 \oplus DATA\}_{EXP \oplus KM} \end{aligned}$$

4. The attacker exports the key obtained in step 1 using the export key completed in step 3, using the claimed type *IMP*. Since the modified *KEK* from step 1 is indeed a key of type *IMP*, this works flawlessly and the attacker obtains:

$$IMP, \{K4 \oplus DATA\}_{EXP \oplus KM}, \{KEK \oplus PIN\}_{IMP \oplus KM} \xrightarrow{(3.5)} \{KEK \oplus PIN\}_{DATA \oplus IMP \oplus K4}$$

5. Note at this point the type confusion enabled by having XORed the *DATA* and *IMP* control vectors into the export and import keys at step 3. The key created in step 4 was exported as type *IMP* but can now be imported as type *DATA*.

6. The attacker imports the key obtained in step 4 with claimed type *DATA* and importer key from step 3. The attacker obtains:

$$DATA, \{KEK \oplus PIN\}_{DATA \oplus IMP \oplus K4}, \{K4 \oplus IMP\}_{IMP \oplus KM} \xrightarrow{(3.4)} \{KEK \oplus PIN\}_{DATA \oplus KM}$$

7. At some later point, the 4758 receives the PIN-derivation key typed as *PIN* and wrapped with the *KEK* that the remote bank system uses unchanged (the modification of $K3$ was only performed locally). The attacker thus has access to $\{P\}_{KEK \oplus PIN}$. Using the key from step 6, the attacker can use "Decrypt with Data Key":

$$\{P\}_{KEK \oplus PIN}, \{KEK \oplus PIN\}_{DATA \oplus KM} \xrightarrow{(3.3)} P$$

8. Using P, the attacker can now generate PINs off-line.

Chapter 5

Difficulties Facing Formal Verification and Potential Solutions

In the ideal case, a formal tool would be able to exhaust every potential attack on a Cryptographic API, find every legitimate attack, and terminate with the claim that no stone was left unturned. This ideal formal tool could then exactly specify *all* weaknesses in an API. In reality, it is very difficult for a formal tool to exhaustively search through every potential attack.

Consider a typical API which may contain fifty to a hundred function calls, and an attacker that can perform generic functions such as encryption and XOR offline. As an approximation, say each function takes in two terms, and outputs a single term. If we define the depth of an attack as the number of consecutive API and offline function calls required, searching through all attacks up to a depth of n takes time and space slightly worse than exponential in n. At level $k \leq n$ 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 $n^2 * b$ function calls available. The search has a total of n such levels, thus a complete search up to a depth of n will take $O(n^{n^2 * b})$ time, which is rewritten as $O(c^{2*n*log(n)})$ time.

Though this simplified model ignores possible constraints on function data types, it illustrates the combinatorial explosion at the heart of formal verification. We present the development of strategies for using OTTER efficiently and methods for simplifying API models while maintaining accuracy. The following three sections will describe the motivation and discovery of various techniques that we used to model the IBM 4758 CCA efficiently. Note that these techniques can be widely applied to any Cryptographic API. Finally, this chapter concludes with a discussion of how to efficiently deal with state and human error.

5.1 A Simple Model of the 4758

Functions can be used to build compound terms. For example, the expression `e(M, KEK)` can be used to represent the encryption of `M` under key `KEK`, and `xor(K1, K2)` can be used to represent the bit-wise XOR of keys `K1` and `K2`. Thus, if $\{KEK\}_{KM \oplus IMP}$ is stored outside the 4758 and thus available to the user, we can specify this to OTTER via the statement

```
U(e(KEK, xor(KM, IMP))).
```

OTTER can also operate over equalities. For example, the relevant properties of `xor` can be specified as:

```
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 implications of this deduction system will be of two types: those deductions which the user can make offline, and API function calls. The first type of implication is 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
- 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.

However, although each individual implication is simple to specify, how to decide which implications to include? It is impossible to include all implications that a real adversary might make on the actual bit-strings; some abstraction must be made. We chose a high level of abstraction, based on the *ideal encryption* assumption of the Dolev-Yao model. Briefly, this assumption states that the only operations available to the adversary with respect to the cryptography are encryption and decryption—and then only if it knows the proper keys:

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

The API functions are also simple to represent as implications: given that the user knows some given input to a function call, it can use the 4758 to learn the output. For example, the "Encrypt with Data Key" function call can be represented as:

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

Here `x` represents the data to be encrypted, and `K` is the data key, typed by encryption under the master key `KM` and the data control vector `DATA`.

When OTTER is run, it attempts to prove an inconsistency in its input. As such, theorems to be proved 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 it will present a proof of how that security goal has been violated.

A model of the 4758 created in this way will be perfectly valid. However, it will also take an extremely long time for OTTER to reason about it. When specified in the simple way described above, the branching tree of possible deductions grows large very quickly. In the next two sections, we describe two sources for this growth, and for each we provide an alternate specification method that eliminates it.

5.2 Partitioning Adversary Knowledge

The above naive specification of the user's offline encryption and XOR 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 [3] on the 4758. Recreating the attacks required OTTER to derive a total of 5,458,647 clauses, and OTTER reasoned about only 8,312 clauses. The process finished in 28 system CPU seconds and 987 seconds of wall clock time. OTTER only reasoned about a small fraction of the generated clauses because the process was run under a mode where clauses that contained more than ten symbols were immediately discarded.

5.3 Harnessing Demodulation

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:

Implication:	`-UN(x)|-UN(y)|-UN(xor(x,y))|GOAL(x).`
Knowledge1:	`UN(xor(B,C)).`
Knowledge2:	`UN(xor(D,E)).`
Knowledge3:	`UN(xor(B,xor(C,xor(D,E)))).`
Negated Goal:	`-GOAL(xor(B,C)).`

will not unify and a proof will not be derived. To resolve these clauses, OTTER may attempt to unify `x/xor(B,C)`, and `y/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 known form (also the simplest form), `GOAL(xor(B,C))` cannot be derived.

We introduce an *intermediate clause* to solve this problem. We do not know if this approach has been used before. We split the implication clause into two separate clauses:

```
-UN(x)|-UN(y)|INTERMEDIATE(xor(x,y),x).
-INTERMEDIATE(z,x) | UN(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, OTTER unifies `z/xor(A,xor(B,xor(C,D)))`, `x/xor(B,C)`, allowing `GOAL(xor(B,C))` to 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.

In order to model the 4758 function “Key Import”:

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

we must split it into two clauses because the term `xor(z,x)` may be a barrier to unification.

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

This encoding of the transaction allows for every possible unification. By using these techniques, we are able to find the known attack by generating only 11,379 clauses and keeping

5,456 clauses. The process takes .06 system CPU second, and takes one wall clock second. In addition, *no* clauses were discarded which means that OTTER should not miss any potential attacks.

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

```
-UN(xor(x,B))|GOAL(x).
```

would be expanded into:

```
-UN(xor(x,B))|GOAL(x).
-UN(xor(B,x))|GOAL(x).
-UN(x)|GOAL(xor(x,B)).
```

Note that OTTER is not guaranteed to be complete when demodulation is used. This section has been full of examples where unification will fail to derive simple conclusions because demodulation is used. However, we firmly believe that the above strategies will maintain completeness. For every potential problem that demodulation creates for unification, we have found a strategy that allows unification to proceed. A proof of this claim is pending.

5.4 Dealing with State

As was pointed out in Section 3.4.3, modeling state can greatly increase the size of a search tree by a factor of the number of possible states. In the case where there are k bits of state, this increases our search space by a factor of 2^k.

A possible solution to the state problem is to treat state transitions as independent. This model of an API can be created by allowing the attacker to preform any function call in *any* state. Although any function can be called at any time, the affect of each function call on the state of the system *is* maintained.

For example, in a particular implementation of PKCS #11, it may be impossible to set the "exportable" attribute if the key is labeled as "sensitive". In the true model of the API, the state of the "sensitive" attribute bit affects the ability of an attacker to call SET_ATTRIBUTE on the attribute "exportable". However, using the above strategy would allow an attacker to toggle the "exportable" bit regardless of the setting of the "sensitive" bit.

Using this strategy, any attack that is possible in the true API will necessarily be found in the state-independent model. In addition, many attacks that are *not* possible in the real API will also be found. This strategy will require manual or automated verification of the potential attacks that are found.

Applying this strategy to the PKCS #11 API allowed us to find the desired attack after generating only 21 clauses and keeping only 44 clauses. In contrast, our initial search required the generation of 55,182 clauses and 5,107 clauses were kept. But what if this strategy yields an enormous number of potential attacks?

In this case, we can adjust our model to consider certain state dependencies but not others. For example, in PKCS #11, the state transitions alter the attribute vectors for a particular key. Although the entire system may contain n keys, functions operating on a set of keys K are only affected by the attributes of the keys in K. State dealing with a particular key should therefore be considered dependent in the ideal case, but independent of the state of other keys. In the case where we cannot consider all of the dependencies in the state of the API, we can trim the number of state dependencies considered to make the model small enough to analyze. Once the proper compromise has been reached, however, we still may be left with a large set of potential attacks. We now present a possible algorithm for processing these attacks. This algorithm was not implemented.

1. Determine the state changes associated with each potential attack. This may involve adjusting the original model to include a string that describes the necessary state conditions for each function call in the API. Note that the number state changes associated with a potential attack should not be larger than the number of total function calls required for an attack. An attack may require three function calls from the initial state, a function call that changes the state of the device, and finally four function calls from this new state. The attack requires eight function calls, while the attack only contains a single state transition.

2. Create a *state tree* where each edge represents a state transition and the root node r represents the initial state. Each state node n is associated with a set of attacks that require *all* of the state transitions between the root node r and n to preform the

initial portion of the attack. Here, the leaves of the tree represent the state transitions for complete attacks.

3. Create a new model of the API that only considers the possible state transitions of the Cryptographic system. It is possible that a model checker would be ideally suited for this task because they deal solely with state transitions, while theorem provers have no native sense of state.

4. Begin pruning the tree. Any state transition that is not possible in the state tree will allow us to eliminate all attacks associated with a node who's path to the root node contains that transition.

5. Once every state transition has been considered, the remaining leaves of the tree represent valid attacks.

5.5 Human Error

It is extremely difficult to correctly model an API. Because the desired result from running a model through OTTER is usually unknown, spotting bugs in a specification from undesirable program behavior isn't easy. Looking through several thousands of lines of output *can* tell you if your model is behaving inefficiently (this tactic showed us that paramodulation was inefficient, and that infinite chains of function calls needed to be avoided). However, looking through output to search for actual errors in the specification is very difficult. When we model an API with a known attack, verifying that attack serves as a very valuable test we can use to judge if our model is correct. In the case where we would like analyze an API which has no known attacks, careful documentation and peer review is likely to be the most effective method for catching modeling errors.

Finally, proving that an attack exists in our model of an API, or that our model is indeed secure, still leaves us with the problem of proving our model is accurate. In the end, this problem is not entirely solvable. It seems that it will always be possible for an attacker to use a function that was not considered as an ability in our model, or that our functions do not quite match the actual system we are studying. Solving this problem is likely to be beyond the scope of this research project.

5.6 Results of Using Demodulation and Independent States

Using demodulation and independent steps on both the 4758 CCA and the VSM greatly reduced the search space. The data for the VSM includes searching through the entire search space available while the data for the 4758 only includes searching until the attack was found.

API	Strategy Used	Attack Found?	Clauses Generated	Clauses Kept	CPU Time (sec)	Wall Clock (sec)
VSM	None	Yes	$> 7,405,163$	$> 11,217$	> 29.11	> 227
VSM	Data Partition/ Demodulation/ Intermediate	Yes	100	105	.01	< 1
CCA	None	No	$> 114,327,949$	> 11159	> 484.34	> 12398
CCA	Data Partition	Yes	5,458,647	8,312	28	987
CCA	Data Partition/ Demodulation/ Intermediate	Yes	11,379	5,456	.06	1

Figure 5-1: Summary of using data partitioning, demodulation, and intermediate steps results.

The above figure shows how a naive OTTER approach is unable to derive significant attacks against the 4758 CCA API, even those short enough to be found manually. (OTTER is even unable to terminate, as far as we can tell.) The figure also shows the significant value of our modeling techniques, allowing OTTER to trivially re-derive previously-known attacks [1, 3, 9]. In the case of the VSM, the above figure clearly shows that the search space can be dramatically decreased by applying the techniques developed in this thesis.

In modeling PKCS #11, we were also able to make significant improvements by treating state transitions as independent. Note that this result is specific to PKCS #11 and is unlikely to be as effective in APIs with heavily dependent state transitions. The experiments were run until the attack was found.

Strategy Used	Attack Found?	Clauses Generated	Clauses Kept	CPU Time (sec)	Wall Clock (sec)
None	Yes	55,170	5,249	.58	80
State Independence	Yes	21	44	.01	< 1

Figure 5-2: Summary using state independence results

Chapter 6

Conclusions and Future Researh

With this work, we have shown the first successful application of formal tools to the analysis of security APIs. Using these tools, we have discovered and detailed a variant of a major flaw in the specifications of a widespread hardware security module. We believe this very same process can be applied to the analysis of many other security APIs.

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 believe our techniques will allow theorem provers to discover many attacks that could never be found by hand.

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

- **optimized security analysis**: 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.

- **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.

- **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.

- **analysis of stateful APIs**: Although the PKCS #11 standard was studied and attacked, the loose definitions of valid state transitions make it a fairly weak API. Also, the fact that Jolyon Clulow had previously found the attacks we verified [8] make our specification of the API less than impartial. A major step in the formal verification of Cryptographic APIs will be the succesful specification of a new stateful Cryptographic API. Attacking such an API will likely require an algorithm such as the one outlined in Chapter 5 to be implemented, which will be a major effort.

- **towards verification**: Currently, Otter has only been used to find holes in Cryptographic APIs. In order to verify the security of a Cryptographic API will require a careful proof that the strategies described in Chapter 5 maintain completeness. If we can prove that the strategies maintain completeness, then we can claim that all possible attacks are found by Otter (given our model).

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.

Appendix A

Complete Models

A.1 VSM

```
% file: vsm.in
% Author: Paul Youn
% Date: Feb 24, 2004
%
% This file is an input file for the OTTER theorem-proving program.
%
%
% Description:
%   Shows how adversary can derive user's pin from a crypto box.
%   Notation:
%      e(x,k) denote encryption of x with key k.
%      U(x) means "User knows x".
%      C(x) means "They crypto box knows x".
%
%   Crypto box uses the following keys:
%      TMK = the box master key (kept inside box)
%      WK = Working Key (kept inside box)
%      ZCMK = Zone Control Master Key (sent to box, encrypted with km)
%      TC = Terminal Communications key.
%      LP = Local Pin Data
```

```
%
%   Crypto box allows adversary to encrypt anything with kmc.
%
%   Crypto box allows adversary to re-encrypt something encrypted with
%      kmc, using a new key that is supplied in encrypted form (with key km).
%
%   Adversary knows the encryption of pin creation key kp encrypted with km.
%
%   Adversary knows users "primary account number" pan.
%
%   The user's pin for the account number pan is defined to be e(kp,pan).
%
%   Want to show adversary can derive pin associated with pan.
%     [ as follows:
%                    adversary encrypts pan with kmc
%                    adversary re-encrypts pan from key kmc to key kp
%     ]

%Input flags, identical to 4758 flags.
set(process_input).

%Set output style.
clear(print_kept).
clear(print_new_demod).
clear(print_back_demod).
clear(print_back_sub).
assign(stats_level,1).

%control memory/time
clear(control_memory).
assign(max_mem, 120000).
assign(max_seconds,1000000).
```

```
%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).

%use breadth first search.
set(sos_queue).

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

%search for 4 attacks.
assign(max_proofs, 4).

list(usable).

%Keys that are on the card.
C(TMK).
C(WK).
C(ZCMK).
C(TC).
C(LP).

%Initial attacker knowledge.
```

```
UE(e(P, TMK)).

UE(e(W, WK)).

UE(e(T1,W)).

UE(e(T2,LP)).

UE(e(K,TC)).

UN(Acc).


%SECURITY GOALS:

-UN(P).

-UN(T1).

-UN(T2).

-UN(K).

-UE(e(Acc,P)).


end_of_list.


list(sos).

%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(eor(x,y)).


%If the user knows something x, encrypted under something z,

%and z encrypted under ZCMK, then the user knows x encrypted

%under TMK.

%IMPORT A PIN DERIVATION KEY

-UE(e(x,z)) | -UE(e(z,ZCMK)) | UE(e(x,TMK)).
```

```
%EXPORT A PIN DERIVATION KEY
-UE(e(x,TMK)) | -UE(e(z,ZCMK)) | UE(e(x, z)).

%CONVERT TMK TO WK
-UE(e(x, TMK)) | UE(e(x,WK)).

%GENERATE TMK
UE(e(NewTMK, TMK)).

%GENERATE TC
UE(e(NewTC, TC)).

%UPDATE TMK (MKB ADDED WED)
-UE(e(x,TMK)) | -UE(e(y,TMK)) | UE(enc(x,y)).

%Input TC
-UN(x) | UE(e(x,TC)).

%EXPORT TC
-UE(e(x,TC)) | -UE(e(y,TMK)) | UE(e(x,y)).

%Encrypt communications data (MKB MOD)
-UE(e(x,TC)) | -UN(y) | UE(e(y,x)).

%Decrypt communcations data.
-UE(e(x,TC)) | -UE(e(y,x)) | UN(y).

%Export trial pin
-UE(e(x,y)) | -UE(e(y, TMK)) | -UE(e(z, WK)) | UE(e(x,z)).

%Translate trial pin between wks.
-UE(e(x,y)) | -UE(e(y,WK)) | -UE(e(z, WK)) | UE(e(x,z)).
```

```
%Export local pin
-UE(e(x,LP)) | -UE(e(z,WK)) | UE(e(x,z)).

end_of_list.

list(demodulators).

%necessary demodulators for XOR
eor(x,y) = eor(y,x).
eor(x, eor(y,z))= eor(y,eor(x,z)).
eor(x,x) = ID.
eor(ID,x) = x.

%demodulators that speed up Otter
eor(eor(x,y),z) = eor(x,eor(y,z)).
eor(x, eor(y, eor(z, eor(x,w)))) = eor(y, eor(z,w)).
eor(x, eor(y, eor(z, eor(w,x)))) = eor(y, eor(z,w)).
eor(x, eor(y, eor(x, z))) = eor(y,z).
eor(x, eor(y, eor(z, x))) = eor(y,z).
eor(x, eor(x,y))= y.

end_of_list.
```

A.2 4758

```
% file: 4758.in
% Author: Paul Youn, Mike Bond, and Jon Herzog
% Date: Feb 24, 2004
%
% 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.
```

A.3 PKCS #11

```
% file: pkcs.in
% Author: Paul Youn
% Date: January 29, 2004
%
% This is an input file for the OTTER theorem-prover
%
%
% Description: Simple attack exporting an arbitrary key with a DES key.
%
% Notation:
%
% A(x) means "adversary knows x" for some x.
% Exp(x, y) denotes exporting key y wrapped with key x.
% Wr is a constant that represents the flagWrapping.
% Ex is a constant that represents the flag Exporting.
% Dec is a constant that represents the flag Decrypt.
% Sen is a constant that represents  the flag Sensitive.
% In(x) means "x is known to the card".
% Att(x, y) represents the attributes of key x.
% Tog(x, y) is a state changing function that toggles a flag in a key. Toggles
% flag y for key x.
% Initial is the constant representing the initial state.
% T(x, u) reperesents that x is true in state s.
% Do(x, y) represents doing an action (such as toggling an attribute)
%    in state s.
% Decrypt(x, y) represents decrypting message y with key x.

%Run in Automatic mode.
set(auto).

%Only find 1 attack.
assign(max_proofs, 1).
```

```
%Allows the use of formulas. Clauses can only consist of the OR of
%a set of literals. Formuals can include the AND of literals.
formula_list(usable).

% initial constants.
T(In(des), Initial).  % some des key is on the card.
T(In(rsa),Initial). % some larger RSA key is on the card.
T(-Att(des,Wr),Initial). % des can't be used to wrap other keys.
T(-Att(des,Ex),Initial). % rsa key can't be exported.
T(-Att(des,Dec),Initial). % des key can't be used to decrypt messages.
T(Att(des,Sen),Initial). % des key is sensitive.
T(-Att(rsa, Wr), Initial). % rsa key can't be used to wrap keys.
T(-Att(rsa,Ex),Initial). % rsa key can't be exported.
T(-Att(rsa,Dec),Initial). % rsa key can't be used to decrypt messages.
T(Att(rsa,Sen),Initial). % rsa key is sensitive.
T(-Att(key3, Wr), Initial). % dummy key can't be used to wrap other keys.
T(-Att(key3,Ex),Initial). % dummy key can't be exported.
T(Att(key3,Dec),Initial). % dummy key can't be used to decrypt messages.
T(Att(key3,Sen),Initial). % dummy key is sensitive.

%Explicitly define equality. Otter does not know if constants refer
%to the same thing.
key3 != rsa.
key3 != des.
des != rsa.
Wr != Ex.
Wr != Dec.
Wr != Sen.
Ex != Dec.
Ex != Sen.
Dec != Sen.
```

```
%
% Note that state remains in Initial for all function calls.
%

% TOGGLING AN ATTRIBUTE:
%STATELESS VERSION
%If a key x has attribute y set to true, then toggle it to
%false.
all x y (-T(Att( x, y), Initial) | T( -Att( x, y), Initial)).
%If a key x has attribute y set to false, then toggle it to
%true.
all x y (-T(-Att( x, y), Initial) | T( Att( x, y), Initial)).

% EXPORTING A KEY STATELESS:
% If in state s, key x and y are in the card, and x can wrap and y
% can be exported then the Adversary gains knowledge of the key
% y wrapped by x.
all x y u (-T(In(x),u) | -T(In(y),u) | -T(Att(x,Wr),u)
| -T(Att(y, Ex),u) | A(Exp(x,y))).

% Decrypting a key. If key w is in the card, the Adversary knows x
% wrapped with w, and w can Decrypt, then the adversary knows the
% plain key. STATELESS
all x y u (-T(In(x), u) | -A(Exp(x, y)) | -T(Att(x, Dec), u) | A(y)).

% SECURITY GOALS:

% A total failure. Adversary knows the rsa key.
-A(rsa).
end_of_list.
```

Bibliography

[1] Mike Bond. Attacks on cryptoprocessor transaction sets. In *CHES*, pages 220–234, Berlin, 2001. Computer Laboratory, University of Cambridge.

[2] Mike Bond. *Understanding Security APIs.* PhD thesis, Cambridge, 2004.

[3] Mike Bond and Ross Anderson. API-level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, October 2001.

[4] Mike Bond and Ross Anderson. Protocol analysis, composability and computation. *Computer Systems: Theory, Technology and Applications*, Dec. 2003.

[5] Mike Bond and Piotr Zielinkski. Decimalisation table attacks for pin cracking. Technical Report TR-560, University of Cambridge, Cambridge, February 2003.

[6] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions in Computer Systems*, 8(1):18–36, February 1990.

[7] Richard Clayton and Mike Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, volume 2523, pages 579–592, 2003.

[8] Jolyon Clulow. On the security of PKCS #11. In *CHES*, pages 411–425, Berlin, 2003. University of Natal, Department of Mathematics and Statistical Sciences, Durban.

[9] Jolyon S. Clulow. The design and analysis of cryptographic application programming interfaces for devices. Master's thesis, University of Natal, Durban, 2003.

[10] Michael Genesereth and Nilsson Nils. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., Los Altos, 1987.

[11] L. Henschen. *Theorem Proving*, volume 2 of *Encyclopedia of Artificial Intelligence.* Wiley-Interscience Publications, New York, 1987.

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

[13] Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.

[14] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceeedings of* TACAS, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[15] J. McCharen, R. Overbeek, and L. Wos. Complexity and related enhancements for automated theorem- proving programs. *Computers and Mathematics with Applications*, 2:1–16, 1976.

[16] William McCune. *Otter 3.3 Reference Manual.* Aragonne National Laboratory, Argonne, Illinois, Aug. 2003.

[17] Catherine Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 84–89. IEEE, IEEE Computer Society Press, 1994.

[18] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communication*, 21(1):44–54, January 2003.

[19] Brook Miles. Win32 API tutorial.

[20] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[21] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997.

[22] J. A. Robinson. A machine-oriented logic based on the resoultion principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, Jan. 1965.

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

[24] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

[25] Sean Smith, Ron Perez, Steve Weingart, and Vernon Austel. Validating a high-performance programmable secure coprocessor. In *Proc. National Information Systems Security Conference*, number 22, New York, Oct. 1999. Secure Systems and Smart Cards, IBM T.J. Watson Research Center.

[26] F. Javier THAYER Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[27] Stephen R. Walli. The POSIX family of standards. *StandardView*, 3(1):11–17, 1995.

[28] Larry Wos and Gail Pieper. *A Fascinating Country in the World of Computing, Your Guide to Automated Reasoning.* World Scientific Publishing, New Jersey, 1999.

[29] Lawrence Wos and George Robinson. Paramodulation and set of support. In *IRIA Symposium on Automatic Demonstration*, Versailles, 1968. U. S. Atomic Energy Commision.

[30] Lawrence Wos, George Robinson, Daniel Carson, and Leon Shalla. The concept of demodulation in theorem proving. *Journal of the Association for Computing Machinery*, 14(4):698–709, Oct. 1967.

[31] Paul Youn, Ben Adida, Jon Herzog, Ron Rivest, Mike Bond, Jolyon Clulow, Ross Anderson, and Amerson Lin. Robbing the bank with a theorem prover (almost). *Submitted to CCS Conference*, 2004.