Automated Analysis of Security APIs

by

Amerson H. Lin

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

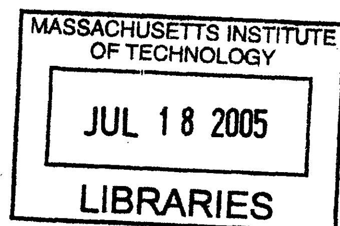
May 2005 [June 2005]

© Massachusetts Institute of Technology 2005. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 18, 2005

Certified by ....
Ronald L. Rivest
Professor
Thesis Supervisor

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



ARCHIVES

Automated Analysis of Security APIs

by

Amerson H. Lin

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Attacks on security systems within the past decade have revealed that security *Application Programming Interfaces* (APIs) expose a large and real attack surface but remain to be a relatively unexplored problem. In 2000, *Bond et al.* discovered API-chaining and type-confusion attacks on hardware security modules used in large banking systems. While these first attacks were found through human inspection of the API specifications, we take the approach of modeling these APIs formally and using an automated-reasoning tool to discover attacks. In particular, we discuss the techniques we used to model the *Trusted Platform Module* (TPM) v1.2 API and how we used OTTER, a theorem-prover, and ALLOY, a model-finder, to find both API-chaining attacks and to manage API complexity. Using ALLOY, we also developed techniques to capture attacks that weaken, but not fully compromise, a system's security. Finally, we demonstrate a number of real and "near-miss" vulnerabilities that were discovered against the TPM.

Thesis Supervisor: Ronald L. Rivest
Title: Professor

Acknowledgments

Firstly, I would like to thank my advisor, Professor R. Rivest, for his guidance, advice and ideas. Secondly, I would also like to appreciate the rest of the Cryptographic API research group: Ben Adida, Ross Anderson, Mike Bond and Jolyon Clulow for their suggestions and comments on earlier drafts. In particular, I would like to express special gratitude to Mike Bond for working so closely with me. Next, I would also like to thank the Cambridge-MIT Institute for funding this research project and providing ample opportunities for collaboration. Lastly, I want to dedicate this thesis to my friends and family who have given me love and support throughout this past year.

Contents

1 Introduction **10**

1.1 Problem with Security APIs . . . 11

1.2 General Approach . . . 12

1.3 Our Contributions . . . 14

2 Previous Work **16**

2.1 A Taste of Security API vulnerabilities . . . 16

2.2 Automated Analysis with OTTER . . . 17

3 Techniques and Tools **19**

3.1 Input/Output API Chaining . . . 19

3.2 Modeling Stateful APIs . . . 20

3.3 Managing API Complexity . . . 21

3.4 Partial Attacks . . . 23

3.5 Tools – OTTER and ALLOY . . . 24

4 Otter – a Theorem Prover **26**

4.1 Overview . . . 26

4.1.1 First-Order Logic . . . 26

4.1.2 Resolution . . . 27

4.1.3 Set-of-support Strategy . . . 29

4.2 Attacks Modeled with Otter . . . 30

4.2.1 VSM . . . 32

4.2.2 IBM 4758 CCA 33

5 Alloy Analyzer – a Model Finder **36**

5.1 Overview 36

5.1.1 Relational Logic 37

5.1.2 Alloy Models 39

5.2 Attacks Modeled with Alloy 43

5.2.1 Triple-mode DES 43

5.2.2 PIN Decimalisation 48

6 Trusted Platform Module **53**

6.1 History of Trusted Computing 53

6.1.1 Trusted Platforms 54

6.1.2 Use Scenarios 54

6.2 TPM Design 55

6.2.1 Features 55

6.2.2 Trusted Platform Module API 56

7 Security Analysis of the TPM **59**

7.1 Initialization 59

7.2 Key Management 64

7.3 Protected Storage 70

7.4 Delegation 75

7.5 Authorization 79

7.6 Model Integration 82

8 Insecure TPM Implementations **85**

8.1 Insecure_Implementation_X 85

8.2 Insecure_Implementation_Y 87

8.3 Insecure_Implementation_Z 87

9 Conclusions and Future Work **91**

9.1 Conclusions 91

9.1.1 Tool Comparison 91

9.1.2 TPM security 92

9.2 Future Work 93

A Partial Attacks **95**

A.1 Triple-mode DES 95

B TPM Models **98**

B.1 Authorization 98

B.2 Key Management and Protected Storage 102

List of Figures

5-1 Alloy Analyzer - model of encryption 42

5-2 ECB|ECB|OFB triple-mode 44

6-1 Roots of trust in a TPM-enabled PC architecture 56

7-1 TPM Storage key hierarchy 65

7-2 Key-handle switching attack 82

List of Tables

5.1 Set and logical operators used in ALLOY 38

7.1 TPM Permanent Flags 60

7.2 TPM Key Types 64

7.3 Delegation Table fields 76

7.4 Family Table fields 76

7.5 OTTER statistics for the integrated model 83

Chapter 1

Introduction

In this connected age, an information security system such as an online banking application has to not only offer services to customers and clients but must also give them assurance of privacy and security across various usage scenarios and threat models. This complexity of this task often results in the system relying on a specialized hardware or software module for security services through a security *Application Programming Interface.*

An *Application Programming Interface* (API) is a set of commands that package and expose complex functionality to external clients in a simple way. These include commands that effect the features directly as well as any other initialization and management commands. Clients can use these commands as building blocks for software applications without having to worry about the underlying details, which are abstracted from them. For example, a client can simply call the command `ConnectToHost(host)` to establish a http connection without having to deal with opening the network sockets and performing checksums on the received packets. A good example of a software API is the Java Standard Development Kit API.

A security API has the same goal of abstraction, except that the commands in a security API are mostly aimed at providing security services and come with multiple implicitly or explicitly stated *security policies.* A security policy is a constraint on the behaviour of the command that guarantees security. For example, an API command in a banking system could be `SecureTransfer(Integer x, Account acc1, Account`

`acc2)`, which transfers x dollars from $acc1$ to $acc2$. There is an implicit assumption here that the data sent to the bank cannot be tampered with and the command may achieve this by opening a Secure-Sockets Layer connection with the bank, although this information is abstracted from the user. This assumption represents *a* security policy of this command. A good example of a software security API is the Microsoft Cryptographic API.

1.1 Problem with Security APIs

In the past decade, the increasing awareness of the importance of security not only fueled the development of security modules but also encouraged better security engineering – many security standards were established, the *Trusted Computing* [1] initiative was started and Microsoft even made security its biggest concern.

As a result, cryptographic primitives, such as key generation and hashing algorithms, encryption and decryption schemes, have undergone rigorous analysis and improvement. In fact, based on past lessons, security engineers know to use established primitives like RSA. Protocols, sequences of steps that setup a working environment such as a key exchange, have also undergone much scrutiny and development, especially after the famous attack on the Needham-Schroeder protocol in 1995 [15]. In the same manner, system designers are careful to use established protocols such as Kerberos or SSL.

Despite these efforts, information security systems reliant on security modules are constantly being broken. We believe that this is often due to poor security API design, which has not received a similar degree of attention as compared to cryptographic primitives and protocols. In fact, Anderson in "Why Cryptosystems Fail" [2] pointed out that many of the failures of retail banking systems were not caused by cryptanalytic or technical weaknesses but by implementation errors and management failures.

Our conviction began when Bond and Anderson found API level attacks on many crypto-processors like the *Visa Security Module* (VSM) and the IBM 4758 *Common*

Cryptographic Architecture (CCA) [17] in 2000. They exploited the VSM's key-typing system, where key types were used to distinguish keys intended for different uses. By manipulating the key-type information, they were able trick the system into accepting an invalid key. They also found a similar attack on the IBM 4758 CCA, which is used in bank networks. In this case, the 4758 XORed key-typing information into key material. Using the XOR properties, they were able to tweak the key-typing information to pass off an invalid key as a valid one for a particular operation. In a short span of a few years, more attacks were found on the Crysalis and PRISM systems [6], giving more evidence that the problem with security APIs was indeed a real one.

The crux of the problem, it seemed, was that security policies could be subverted when API commands were chained in a totally unexpected manner. For a system with a handful of commands, protecting against these attacks would not be a problem. But what about complex real systems that have 50-100 operations working across 4-5 modules? Another problem was that the security policies for the system were not translated down to API-level requirements that could be checked easily by the software developers. We believe that the addition of security statements at API-level documentation is a good start to solving the security API problem.

Despite the likelihood of vulnerability, security APIs have not received due academic attention. This thesis explores this problem with formal automated analysis.

1.2 General Approach

Many of the earlier security API vulnerabilities were found through careful human inspection and analysis, as Bond and Anderson did with the VSM. With larger systems containing more API operations, this requires thinking through an exponentially large search space and verification becomes a tedious and unforgiving process. The most obvious response is to employ mathematics and formal methods together with automation.

But hasn't this already been done? Many ask if security API analysis is all that

different from security protocol analysis. We argue that APIs present a tremendously more complex problem since API operations could potentially use cryptographic primitives, initiate security protocols, interact with the network and operating system and even manipulate non-volatile storage. Furthermore, security APIs tend to have extensive key management features, something that protocols generally do not. Therefore, we argue that the analysis of APIs is necessarily more complex and will require greater considerations than those of protocols.

Nevertheless, studying the analysis of security protocols gave us a wide variety of techniques and tools to choose from. The formal methods used in analyzing security protocols range from belief-based approaches like the *BAN logic* model [18], to model-checking approaches that started with the *Communicating Sequential Processes* (CSP) [11] and *Dolev-Yao* models [8], to proof-based approaches such as induction in reasoning about sequences of steps. We needed a language that could describe API commands and a tool that could search through chains of these operations, returning the exact execution trace if an attack was found.

This led us to current choice of OTTER, a theorem-prover and ALLOY, a model-finder. Theorem-provers find a series of logical statements (i.e. a proof), derived from an initial set of axioms, that prove a particular statement. We model API commands as logical implications and attacks as goal statements. The tool then searches for a proof of the goal statement. If it terminates and finds one, the series of implications will represent the sequence of API calls for the attack. Model-finders work very differently; a model-finder takes a set of constraints as a model and attempts to find an instance that satisfies it. We model the commands and the goal as constraints. If the tool terminates and finds a satisfying instance, the objects in the instance will reveal the sequence of API calls that constitute the attack.

However, all automated analysis can only be done within bounds. OTTER's proof search may terminate with no proof or may not even terminate at all. Likewise, ALLOY requires a user-specified bound and not finding a satisfying instance within those bounds does not preclude an attack. In either case, we can only state that an attack is unlikely and nothing stronger – a fundamental shortcoming of bounded

automated analysis. Despite this deficiency, we were able to model some previously known attacks in OTTER. Then, we saw success with this approach when OTTER found an extremely complicated "near-miss" attack on the IBM 4758 CCA that was prevented only because the product did not fully conform to its advertised behaviour. Later in this thesis, we will present other successes of this general approach.

1.3 Our Contributions

Our contribution to the Cryptographic API group is three-fold: exploring the ALLOY model-finder, understanding the *Trusted Platform Module* (TPM) API and developing new API formal analysis techniques.

- **Alloy:** This lightweight model-finder was designed to analyze small specifications of software systems. We used it to analyze feature level policies, manage API complexity and even model information leakage, tasks that do not play to the strengths of OTTER.

- **Trusted Platform Module:** The TPM is the *Trusted Computing Group's* hardware security device, the answer to *Trusted Platform Computing.* This was a worthwhile target since giants like National Semiconductor and Infineon had already started producing TPM chips and IBM had already incorporated them into their Thinkpad laptops. While we were unable to find attacks consisting of long chains of commands, our analysis revealed some security flaws in the design and documentation.

- **New Techniques:** With a new tool and a new target API, we were able to develop a number of new approaches in analyzing security APIs. These include an extension to the API chaining technique and techniques to model stateful APIs, manage API complexity and model *partial attacks* – attacks that weaken, but not fully compromise, the security of the system. Interestingly, we find that in terms of finding attacks, the discipline of understanding an API sufficiently to model it with a technique is often more effective than the technique itself.

The rest of this thesis is structured as follows. Chapter 2 reviews previous modeling work done by this research group. Chapter 3 describes the approaches we used in our analyses. Chapters 4 and 5 describe the automated-reasoning tools OTTER and ALLOY respectively. Chapter 6 introduces the TPM and chapter 7 presents our analysis on the TPM's API, with some of our findings reported in chapter 8. Finally, we conclude in chapter 9 with some discussion and suggestions for future work.

Chapter 2

Previous Work

BAN logic, introduced by Burrows, Abadi and Needham in 1989 [18], started the work of formal analysis on security protocols. Meadows further developed the notion of automated verification with the NRL protocol analyzer [19]. Then in 1995, Lowe's successful attack and fix on the Needham-Schroeder protocol [15] highlighted the necessity for protocol verification. Despite these efforts in analyzing security protocols, because of the differences between protocols and APIs already noted, verification of security APIs did not begin until Bond and Anderson attacked a number of security APIs in retail banking crypto-processors in 2000, including the VISA hardware security module (VSM) and the IBM 4758 Common Cryptographic Architecture (CCA) [17]. The attacks were made possible through an unexpected chain of API calls. In 2004, research teams in both Cambridge University and MIT, jointly known as the Cryptographic API group, began to direct research efforts towards analyzing security APIs using automated reasoning tools.

2.1 A Taste of Security API vulnerabilities

For a flavour of the nature of security API vulnerabilities, let us take a look at the attacks on the VISA Security Module and the IBM 4758 CCA, which were found by *Bond et al.* [6].

We first consider the VSM, a cryptoprocessor with a concise API set designed to

protect *Personal Identification Numbers* (PINs) transmitted over the banking network supported by VISA. It employs a key-typing system, where key-types are used to distinguish keys meant for different uses. The VSM's `Encrypt_Communcations_Key` command takes in a key and types it as a *communication key*. The `Translate_Communications_Key` command takes a communication key and encrypts it under a *terminal master key* – a master key of an *Automated Teller Machine* (ATM). The VSM's failure to distinguish between a *PIN derivation key* and a terminal master key made it possible to type a bank account number as a communications key using `Encrypt_Communications_Key` and then encrypt it under the PIN derivation key using `Translate_Communications_Key`, giving the attacker the bank account number encrypted under the PIN derivation key – the PIN associated with that bank account!

Now we consider the IBM 4758 CCA, an API implemented by the majority of IBM's banking products. It has 150 functions and supports a wide range of banking operations. In particular, it uses a key-typing mechanism where keys are typed by encrypting them under a *key control vector* XORed with the master key. A key control vector is a simple string containing type information. At the same time, it supports a `Key_Part_Import` transaction that allows a module to combine partial keys to form the final key, thereby allowing a policy of shared control. The problem is that `Key_Part_Import` also uses the XOR operation as part of its computation, allowing adversaries to XOR cleverly chosen values with partial keys and then using the `Key_Part_Import` operation to re-type the key into a different role. Constructing these keys formed the starting point for a number of attacks [6].

2.2 Automated Analysis with Otter

While the previous attacks were found by hand, effort in modeling API operations in *first-order logic* (FOL) showed that attacks could be found more efficiently by machine. For example, the general decrypt operation could be stated as the implication "If A knows the value of $DATA$ encrypted under key K , A knows the value of the key K, he can decrypt to get $DATA$", which can be subsequently written as the FOL

clause:

$$-A(E(DATA, K)) \mid -A(K) \mid A(DATA)$$

where the predicate $A(x)$ represents that x is a value known to the attacker and $E(x, y)$ represents to the value of x encrypted under y. Finally, we assert that some values y are secret by writing $-A(y)$. Then, the theorem-prover will try find a contradiction to $-A(y)$, thereby producing an attack. This method proved to be extremely valuable when we were able to verify known attacks on the IBM 4758 but the real success came when OTTER discovered a variant of the powerful PIN derivation attack on the IBM 4758. Again, the same weakness: that the XOR operation was used both in key typing and secret sharing, was exploited. This variant required 12 steps and involved 8 different commands, attesting to the importance of the role of automated verification in discovering intricate and complicated attacks. More detail on this modeling approach, including techniques to reduce search space explosion, can be found in chapter 4.

Chapter 3

Techniques and Tools

Any practical automated reasoning approach consists of a technique and a tool that effectively executes the technique. Our aim was to use established techniques, with a few enhancements, in a less established field of research. These include representing values symbolically and modeling the attacker's knowledge set. This way, we could harness the power of techniques that had been proven to work. Next, we had to find tools to execute these techniques. Our choice of OTTER, a theorem-prover, and ALLOY, a model-finder, allowed us to achieve success with our techniques, especially since the tools complemented one another. This chapter summarizes these techniques and the tools used to effect them.

3.1 Input/Output API Chaining

This technique stems from the work that *Youn et al* did on the IBM 4758 CCA [24], where the *first-order logic* model of the 4758's API commands allowed the tool to chain commands together in a many-to-many fashion. Basically, the model works by defining a set of attacker knowledge, a set of possible actions and a set of desired goals. The attacker is then allowed to feed any of his known values into the commands and add any output values to his set of knowledge. This framework captures the attacker's initial knowledge, his ability to use the system API methods, his ability to perform typical primitive operations (e.g. XOR, concatenation, encrypt, decrypt) and the

scenarios that constitute attacks. This way, an exploration of the entire search space would yield any potential API-level vulnerabilities.

We extend this idea by noticing that APIs tend to re-use data structures for various commands. For example, all encrypted blobs, be they encryptions of data or keys, may be returned to the user in the same data structure. This can create opportunities for input/output confusion: an API command expecting an encrypted key could be passed an encryption meant for another command. Our new approach captures these possibilities by under-constraining the inputs and outputs to each API call.

3.2 Modeling Stateful APIs

Many systems maintain some kind of state in memory. The problem with modeling state is that it greatly increases the search space of the automated tool. If there are k bits of state, the search space could potentially increase by a 2^k factor. On the other hand, a stateless model of a stateful API gives the attacker more power; any real attacks will still be found but the analysis may also yield attacks that are possible in the model but impossible in the real system. Therefore, capturing state in our models is important because it will constrain the attacker's abilities.

However, many systems retain a lot of state information. We propose that a way of managing this complexity is to model state based on how it will constrain the attacker's abilities. Firstly, we can do away with all state that is not security-sensitive, state that does not have security related dependencies. A good example is statistical data that is kept solely for performance benefits. However, it is important to note that deciding whether state is security-sensitive requires an in-depth understanding of the API's threat model. It is possible that a security-insensitive flag may affect the operation of a security-sensitive command, thereby making it actually security-sensitive. Secondly, we can partition all security-sensitive state objects into either "loose" or "bound" state. In vague terms, the attacker can independently manipulate any state that is not bound to the rest of the system - this state is described as "loose".

All other state is considered "bound".

"Loose-state" objects are objects that can take on any value regardless of the values in the remaining state objects. In other words, they are not bound to the rest of the configuration and the attacker can independently manipulate their values. A good example is a system flag that can be flipped at any time whatsoever. "Bound-state" objects, on the other hand, take on values that are bound to the rest of the configuration, values that the attacker cannot freely manipulate. Consider two flags in the system bound by the following policy: the second can be set only if the first is set. In this case, the first is "loose-state" while the second flag is "bound-state". It is important to note that deciding whether state objects are "loose" or "bound" also requires understanding the threat model of the API.

Later in this thesis, we present a method for modeling state effectively by grouping types of state together and representing them in OTTER using multiple predicates. Then, all the state is moved through each API command with an over-arching state predicate that captures these multiple predicates. This way, commands affecting a small portion of the state only need to be concerned with the one or two predicates. This technique can be seen in section 7.1.

3.3 Managing API Complexity

A key component to analyzing a system and its API is managing complexity. Automated tools, however powerful, still suffer from search-space explosion. Although some tools are intelligent enough to automatically prune their search-space (e.g. by recognizing symmetries), the best solution is still for the user to manage complexity at the level of the model – understanding what can be left out without compromising the analysis.

The first strategy that we present is *targeting*. Based on experience analyzing security systems, we find that vulnerabilities usually occur across:

- **Trust boundaries:** Multiple systems functioning in the real world represent different trust domains. In a company, for example, one system could belong

to the legal department and another to the IT department, which has a very different security policy. However, the two departments may need to share sensitive information. Such functionality is a likely vulnerability since it is always difficult for the sender to maintain control over information once it is in the hands of the receiver.

- **Module boundaries:** Modules are a natural way to manage complex systems and many systems use some form of key-typing to bind key material to a particular module. This is because an operation may need to know whether a particular key is a master key, a communications key or a storage key. However, key material that is passed between modules could be maliciously manipulated to produce an attack. The VSM and IBM 4758 CCA attacks discussed in the last chapter revealed such weaknesses.

- **Version boundaries:** A commercial system that rolls out upgrades or patches must still support old versions for legacy reasons and it is not uncommon that the development teams are different for two different versions. Poor documentation and the necessary "shoe-horning" in new versions required to ensure backward compatibility make it likely that security policies are overlooked in the process.

In general, enforcing security policies across boundaries often requires a complex solution and complex solutions are often seen to break. Understanding these boundaries will allow the user to first, breakdown the modeling problem into smaller problems without the fear of excluding too many possible attacks and second, focus only on subsections of an API.

The second strategy is one familiar to mathematicians: *reductions*. Many system APIs are unnecessarily complex because they offer multiple ways to perform the same task. While it is true that these multiple ways could potentially offer more opportunities for vulnerabilities, it may also be the case that these multiple ways are actually equivalent. In these cases, the model need only consider the simplest one. A proof by hand of their equivalence may be sufficiently tedious and complicated to

deter designers. In chapter 7, we demonstrate how an automated tool can help us with these reductions.

3.4 Partial Attacks

Partial attacks can be described as the weakening of security, which differ significantly from the attacks we were trying to model earlier, where security was completely compromised. Partial attacks often lead the way for brute-force attacks to succeed much more quickly.

Our work has been focused on attacks that discover "more-than-intended" information about a secret, or what we term *information leakage.* Our general technique is to create models of systems that operate on reduced-size secrets so that the automated tool can reason about the secrets directly – examples include using 8-bit numbers instead of 64-bit numbers, single-digit *Personal Identification Numbers* (PINs) instead of 4-digit PINs. Within this technique, we have also developed two frameworks for modeling systems in order to discover such weaknesses:

- **Entropy Reduction:** In this approach, the API of the system is modeled such that the automated tool is able to reason about the input/output (I/O) characteristics of each method. I/O characteristics can range from linearly-dependent output to completely random output. Subsequently, systems that use encryption to protect secrets can be analyzed for information leakage by analyzing the I/O characteristics after a chain of operations. By describing I/O properties that represent a reduction in entropy, we can then ask the tool to find a sequence of events (i.e. an attack) that leads to such a characteristic. We used this technique to analyze the ECB|ECB|OFB triple-mode block-cipher, which will be described in greater detail in section 5.2.

- **Possibility Sets:** This involves modeling APIs using variables with values that belong to a possibility sets, representing the attacker's possible guesses. Information leakage can clearly be observed if the number of possibilities, say for a

password, reduces more quickly than it should with a series of cleverly chosen API calls. We employed this approach to model the discovery of PINs with repeated API calls to the IBM 4758 cryptoprocessor, where each call leaked enough information for the attacker to perform an efficient process of elimination. This is also detailed in section 5.2.

3.5 Tools – Otter and Alloy

Our tool suite now includes both a theorem-prover, OTTER, and a model-finder, ALLOY. Both tools are based on *first-order logic*, so quantification is done only over atoms. OTTER takes a set of logical statements and tries to prove a goal statement based on them. Its first application was to produce the derivations and proofs of mathematical theorems. ALLOY is a declarative model-finder, based on *first-order relational logic*, designed to be a lightweight tool for reasoning about software and safety-critical systems. It takes a set of constraints and attempts to find a model that satisfies these constraints using a powerful SAT solver.

Before understanding in detail how OTTER and ALLOY work in sections 4 and 5, let us first describe their relative strengths:

- **State:** State can be modeled in OTTER by tagging each clause with a state predicate that changes with each operation. This can be cumbersome when the amount of state increases. In ALLOY, state is modeled by adding an extra dimension to all the relations. While this is a natural use of ALLOY, the model requires an extra *frame condition* for each additional state object since all objects have to be explicitly constrained to stay the same across states when not acted upon by any operation.

- **Chaining:** OTTER was naturally designed for derivation and thus is very well suited to finding API-chaining attacks. Also, once an attack is found, OTTER's proof trace will tell us exactly how the attack occurred. On the other hand, when ALLOY finds an instance containing a violation of a policy, the instance

only shows the objects that constitute the violation and not how the violation arose. In most cases, however, discovering the attack trace is relatively easy.

- **Composition:** OTTER is excellent at API-chaining within a single system but when an analysis is required against multiple systems, these systems have to be explicitly created in the model. In ALLOY, however, the model-finding approach ensures that all possible combinations of multiple systems (up to a specified bound) are searched.

We will cover OTTER and ALLOY in greater detail, including examples and sample code, in the next two chapters.

Chapter 4

Otter – a Theorem Prover

4.1 Overview

The traditional theorem-prover uses a process that starts with a set of axioms, produces inferences rules using a set of deductive rules and constructs a proof based on a traditional proof technique. OTTER (Organized Techniques for Theorem-proving and Effective Research) is a resolution-style theorem-prover based on the set-of-support strategy, designed to prove theorems written in *first-order logic* (FOL) by searching for a contradiction. While the main application of OTTER is pure mathematical research, we found its deduction system to be very suitable in modeling APIs and searching through chains of API sequences. In this overview of OTTER, we will use a simple model of encryption as an illustration of the modeling approach. We will also describe some techniques for managing search-space explosion in OTTER.

4.1.1 First-Order Logic

At the core of OTTER lies FOL, resolution and the set-of-support strategy. We will describe these concepts one at a time. To get a grasp of what FOL is, let us consider where it falls in the three basic categorizations of logic: *zeroth-order logic, first-order logic* and *higher-order logic.* Zeroth-order logic, also known as *propositional logic*, is reasoning in terms of true or false statements. First-order logic augments it by adding

three notions: terms, predicates and quantifiers. These allow us to reason about quantified statements over individual atoms[1]. Higher-order logic further extends the expressiveness of first-order logic by allowing quantification over atoms and sets of atoms. The following terminology used in OTTER's FOL language syntax will be useful in understanding this chapter:

- **constant** – any symbol not starting with {u,v,w,x,y,z}
- **variable** – any symbol starting with {u,v,w,x,y,z}
- **term** – a term is either a variable, constant or a function symbol $f(t_1, \cdots, t_n)$ applied to term(s).
- **predicate** – a predicate P expresses a property over term(s) and is either true or false
- **atom** – an atom is a predicate applied to term(s) (e.g. $P(x)$)
- **literal** – a literal is an atom or the negation of an atom (e.g. $-P(x)$)
- **clause** – a clause is a disjunction of literals (e.g. $-P(x) \mid -Q(x) \mid R(x)$). A clause can have no literals, in which case it is the empty clause.

With this language, we can express many cryptographic operations in a symbolic way. For example, in most of our OTTER models, we use the predicate $U(x)$ to express that the user knows value x and the binary predicate $E(x, k)$ to represent the encryption of x under key k. Now that we can read statements written in FOL, we can start to describe OTTER's reasoning process.

4.1.2 Resolution

Resolution [21] was developed as a method for quickly finding a contradiction within a set of clauses by searching to see if the empty clause can be derived. When this happens, the set of clauses cannot be satisfied together (since the empty clause is

[1]Most of mathematics can be symbolized and reasoned about in FOL

always false) and the set of clauses is termed *unsatisfiable*. The following algorithm describes how resolution works:

1. find two clauses $p = (p_1 \vee \cdots \vee p_n)$ and $q = (q_1 \vee \cdots \vee q_n)$ containing the same predicate literal, negated in one but not in the other. Let us say predicates p_i and q_j are the same.

2. *unify* the two predicates, producing θ, the substitution that will make p_i equal q_j symbolically (e.g. $f(x)$ and $-f(v)$ unify with the substitution x/v) .

3. perform a disjunction of the two clauses p, q and discard the two unified predicates: $(p_1 \vee \cdots \vee p_{i-1} \vee p_{i+1} \vee \cdots \vee p_n \vee q_1 \vee \cdots \vee q_{j-1} \vee q_{j+1} \vee \cdots \vee q_n)$

4. substitute any remaining variables using θ, producing the final *resolvent*

5. add the resolvent to the set of clauses

Resolution aims to derive the empty clause $\{\}$, which can only happen when two completely complementary clauses are resolved (e.g. $P(x)$ and $-P(x)$). It is easy to see that when that happens, step 3 will produce a clause with no literals. It is also important to note that resolution is a sound inference rule and resolution-refutation, which proves a theorem by negating the goal statement to be proved and producing a proof-by-contradiction on the goal statement, is also complete.

In OTTER's FOL language, we write statements as clauses that contain a disjunction (OR) of literals. Consider the encryption operation written as a disjunction of literals:

```
-U(x) | -U(KEY) | U(E(x,KEY))
```

where the predicate $U(x)$ represents that the user knows value x and the term $E(x, y)$ represents the value of x encrypted under y. In OTTER's syntax, there is an implicit *universal quantification* over variables. Therefore, this says that if the user knows any value x and the constant *KEY*, he can encrypt x with *KEY*. With all statements written in this form, OTTER tries to prove that this set of clauses is unsatisfiable using

resolution. When it does find a proof, OTTER prints out the trace of the conflicting clauses.

OTTER supports a number of inference rules, including *binary-resolution* and *hyper-resolution.* Binary-resolution is resolution performed on clauses with one positive and one negative literal (e.g. $-P(x) \mid Q(x)$). By clashing $P(x)$ with $-P(x)$, we can resolve to obtain $Q(x)$. Hyper-resolution generalizes this process to clauses with multiple negative literals and one positive literal and is thus more efficient.

4.1.3 Set-of-support Strategy

OTTER's basic deductive mechanism is the *given-clause algorithm*, a simple implementation of the set-of-support strategy [14]. OTTER maintains four lists for this algorithm:

- **usable** – contains clauses that are available for making inferences
- **sos** – contains clauses that are waiting to participate in the search [2] (i.e. they must be transferred into the `usable` list first).
- **passive** – contains clauses used for forward subsumption and contradiction.
- **demodulators** – contains equalities used to rewrite newly inferred clauses

The given-clause algorithm is basically a loop:

```
while (sos is not empty) AND (empty clause has not been derived)
   1) select the best clause in sos to be the given_clause
   2) move given_clause from sos to usable
   3) infer all possible clauses using the given_clause
   4) apply retention tests to each newly inferred clause
   5) place the retained clauses in sos
```

In step (3), it is important to note that any inferences made must use the given_clause as a parent. In step (4), the retention tests include *demodulation* and *subsumption.*

[2] sos stands for "set-of-support"

Demodulation is the rewriting of newly inferred clauses into a canonical form (e.g. rewriting `add(add(1,2),3)` to `add(1,add(2,3))`) and subsumption is the discarding of newly inferred clauses in favour of a more general clause (e.g. discarding `P(1)` since `P(x)` had already been derived)

4.2 Attacks Modeled with Otter

This section describes previous work done on modeling the Visa Security Module (VSM) and IBM 4758 Common Cryptographic Architecture (CCA) security APIs using OTTER [24]. The VSM and IBM 4758 CCA are hardware-based cryptoprocessors used in bank *Automated Teller Machine* (ATM) networks to enable two important secure-banking features: first, communication between the bank and the ATM and second, *Personal Identification Number* (PIN) verification. Underlying these features are a few building blocks: symmetric-key encryption, a unique master key, *key-sharing* and *key-typing*. Key-sharing is the process of breaking a key into parts such that the key can only be fully reconstructed when all parts are present. Key-typing is the process of tagging on meta-data describing the key's intended use and usage domain. For example, data keys are used for encrypting and decrypting data and communication keys are used to protect session keys.

To authenticate the customer present at the ATM, banks issue a PIN to each customer that is cryptographically derived from their *Primary Account Number* [3] (PAN) by encrypting the customer's PAN with a PIN-derivation key [4] . When customers change their PIN numbers, the numerical difference (i.e. the offset) is stored in the clear on the bank card. During authentication, the crypto-processor must then be able to compute a customer's PIN, add the offset, and then compare this value with the numbers keyed in by the customer. To prevent fraud, both the PIN and the PIN-derivation key must be protected.

[3] the primary account number is also known as the bank account number

[4] there are other PIN generation and verification schemes

General Technique

Both the VSM and the CCA were modeled using the same basic technique: API calls were modeled as implication clauses in the usable list and the attacker's initial knowledge was written as clauses in the sos list. The inputs to the API command were written as negative literals and the output was written as the single positive literal. If the API command produced more than one output, more implication clauses for the same command were required. Then, OTTER was enabled with *hyper-resolution* to enable the unification of the implication clause with multiple positive literals. It must be noted that this technique was developed initially by Youn in 2004. The encryption command is given below:

```
set(hyper_res).
set(sos_queue).

list(usable).
  -U(x) | -U(KEY) | U(E(x,KEY)).
end_of_list.

list(sos).
  U(DATA).
  U(KEY).
end_of_list.
```

Here, `set(hyper_res)` turns on the hyper-resolution inference rule and `set(sos_queue)` instructs OTTER to pick clauses from the sos list in the order in which they were put in. On the first iteration, OTTER will pick `U(DATA)` and place it in the usable list. However, nothing new is inferred at this point. On the second iteration, OTTER picks `U(KEY)` and this time, together with the two clauses in the usable list , it is able to produce the resolvent `U(E(x,KEY))` with the substitution `DATA/x`, giving `U(E(DATA,KEY))`.

4.2.1 VSM

The following attack on the VSM, briefly mentioned in 2.1, enables the attacker to derive any customer's original PIN number. To demonstrate the attack, it suffices to know that communication keys are only used to transfer data from module to module, master keys are used only to encrypt other keys and the PIN derivation key is used only to calculate the PIN from a PAN. The attack uses two API commands:

- `Create_Communications_Key` – creates keys of type "communication key" from a user provided blob.
- `Translate_Communications_Key` – re-encrypts communication keys under a user provided terminal master key.

The OTTER model is shown below, where the goal is to obtain the clause `U(E(Acc,P))`, which represents the attacker obtaining the encryption of an arbitrary account number under the PIN derivation key, yielding the customer's PIN.

```
% Create_Communications_Key
-U(x) | U(E(x,TC)).

% Translate_Communications_Key
-U(E(x,TC)) | -U(E(y,TMK)) | U(E(x,y)).

% User knows the account number
U(Acc).

% User knows the encrypted PIN-derviation key
U(E(P,TMK)).

% Negation of goal
-U(E(Acc,P)).
```

OTTER was able to derive the clause `U(E(Acc,TC))` and then `U(E(Acc,P))` to finally produce a resolution refutation (i.e. the empty clause) with `-U(E(Acc,P))`, even with many other extra commands added to the model, testifying to OTTER's ability to search large spaces quickly.

4.2.2 IBM 4758 CCA

The following attack on the IBM 4758 CCA, briefly mentioned in 2.1, also enables the attacker to derive any customer's original PIN number. The attack uses three API operations:

- `Key_Import` – uses an importer key to decrypt an encrypted key, thereby importing it into the system.
- `Key_Part_Import` – adds a key part (split during key sharing) to the already existing key parts in the system, generating a full key.
- `Encrypt` – uses a data key to encrypt arbitrary data.

This attacks leveraged on key-typing, insufficient type-checks and XOR cancellation to use a legitimate command (`Key_Import`) in an illegitimate way. Instead of describing the attack in detail, we focus on the modeling techniques that were developed by Youn [24] to control the exponential search space explosion in the model. For details of the attack, please refer to [24].

- **Knowledge Partitioning** – the predicates `UN(x)` and `UE(x)` both refer to the user knowledge of x but one refers to x as a plaintext value and the other refers to x as an encryption. By having `UE(x)` representing encrypted values, we prevent the repeated encryption of encrypted values since the encryption command only takes in non-encrypted values `UN(x)`. This kind of partitioning can be very useful since it can dramatically reduce the branching factor.
- **Demodulation** – the use of XORs in this security API presented a rather difficult problem: a clause containing XORs could be expressed in multiple

different ways. With the list of demodulators, OTTER was able to rewrite every XOR-containing clause to its canonical form.

- **Intermediate Clauses** – the inference rule for `Key_Import` was broken down into two inference rules: the first produced an intermediate clause `INTUE` to be used an input to the second inference rule. This was introduced to solve the problem when the representation required for unification is not the canonical form (and when paramodulation [5] is not available).

- **Lexicographic Weighting** – OTTER's *given-clause algorithm* picks a given-clause from the sos list based on weight. By specifying a lexicographical ordering over symbols, we can guide OTTER into picking certain clauses over others.

A portion of the OTTER model with these techniques in use is shown below:

```
list(usable).
% Key Import, INTUE() is the intermediate clause
-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))).

% Key Part Import
-UN(xor(x,KP)) | -UN(y) | -UE(E(z,xor(x,xor(KP,KM)))) |
  UE(E(xor(z,y), xor(x,KM))).

% Encrypt
-UN(x) | -UE(E(y,xor(DATA,KM))) | UE(E(x,y)).

% Ability to XOR
-UN(x) | -UN(y) | UN(xor(x,y)).
end_of_list.
```

[5] refer to the OTTER manual [14] for an explanation of paramodulation

```
list(sos).
% User has the third key part
UN(K3).

% User knows the DATA key type
UN(DATA).

% User has the encrypted PIN-derivation key
UE(P,xor(KEK,PIN)).

% XOR demodulators (rewrite rules)
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.
end_of_list.
```

Although these methods all help to reduce the search-space explosion, there is the corresponding trade-off of OTTER not visiting certain search paths, potentially missing certain attacks.

Chapter 5

Alloy Analyzer – a Model Finder

5.1 Overview

ALLOY is a lightweight modeling language based on *first-order relational logic* [13], designed primarily for software design. The ALLOY ANALYZER [10] is a model-finder developed to analyze software models written in the ALLOY language. Hereafter, we will use ALLOY to refer to both the language and the tool. As an automated reasoning tool, ALLOY can be described neatly in three parts, each of which will be described in greater detail later.

- *logic* – ALLOY uses first-order relational logic, where reasoning is based on statements written in terms of atoms and relations between atoms. Any property or behaviour is expressed as a constraint using set, logical and relational operators.
- *language* – the ALLOY language supports typing, sub-typing and compile-time type-checking, giving more expressive power on top of the logic. It also allows for generically-described modules to be re-used in different contexts. Also very useful is syntax support for three styles of writing ALLOY model specifications that users can mix and vary at will: *predicate-calculus* style, *relational-calculus* style and *navigational-expression* style [1]. Navigational-expression style, where expressions are sets and are addressed by "navigating" from quantified variables,

[1] refer to the ALLOY manual [13] for explanations and examples of these styles of writing

is the most common and natural.

- *analysis* – ALLOY a model-finder that tries either to find an *instance* of the model or a counter-example to any property assertions specified in the model. An instance is literally an instantiation of the atoms and relations in the model specification. It performs a bounded-analysis by requiring a user-specified bound on the number of atoms instantiated in the model. With this bound, it translates the model into a *boolean satisfiability* (SAT) problem. Then, it hands the SAT problem off to a commercial SAT solver such as Berkmin [9]. The resulting solution is then interpreted under the scope of the model and presented to the user in a graphical user-interface as show in Figure 5-1

5.1.1 Relational Logic

All systems modeled in ALLOY are built from *atoms, tuples* and *relations.* An atom is an indivisible entity, a tuple is an ordered sequence of atoms and a relation is a set of tuples. Because ALLOY's relational logic is first-order, relations can only contain tuples of atoms and not tuples of other relations.

A good way of thinking about relations is to view them as tables. A table has multiple rows, where each row corresponds to a tuple and each column in a tuple corresponds to an atom. Then, a unary relation is a table with one column, a binary relation is a table with two columns and so on. Also, the number of rows in a relation is its *size* and the number of columns is its *arity.* A *scalar* quantity is represented by a singleton set containing exactly one tuple with one atom.

To express constraints and manipulate relations, we use operators. ALLOY's operators fall into three classes: the standard *set operators*, the standard *logical operators* and the *relational operators.* The standard set and logical operators listed in Table 5.1.

The relational operators require a little more treatment. Let p be a relation containing k tuples of the form $\{p_1, \cdots, p_m\}$ and q be a relation containing l

symbol	operator
+	union
-	difference
&	intersection
in	subset
=	equality
!	negation
&&	conjunction
||	disjunction

Table 5.1: Set and logical operators used in ALLOY

tuples of the form $\{q_1, \cdots, q_n\}$.

- `p -> q` – the *relational product* of p and q gives a new relation $r = \{p_1, \cdots, p_m, q_1, \cdots, q_n\}$ for every combination of a tuple from p and a tuple for q (kl pairs).
- `p .q` – the *relational join* of p and q is the relation containing tuples of the form $\{p_1, \cdots, p_{m-1}, q_2, \cdots, q_n\}$ for each pair of tuples where the first is a tuple from p and the second is a tuple from q and the last atom of the first tuple matches the first atom of the second tuple.
- `^p` – the *transitive closure* of p is the smallest relation that contains p and is *transitive*. Transitive means that if the relation contains (a, b) and (b, c), it also contains (a, c). Note that p must a binary relation and that the resulting relation is also a binary relation.
- `*p` – the *reflexive-transitive closure* of p is the smallest relation that contains p and is both transitive and *reflexive*, meaning that all tuples of the form (a, a) are present. Again, p must be a binary relation and the resulting relation is also a binary relation.
- `~p` – the *transpose* of a binary relation r forms a new relation that has the order of atoms in its tuples reversed. Therefore, if p contains (a, b) then r will contain (b, a).
- `p <:q` – the *domain restriction* of p and q is the relation r that contains

those tuples from q that start with an atom in p. Here, p must be a set. Therefore, a tuple $\{q_1, \cdots, q_n\}$ only appears in r if q_1 is found in p.

- `p >:q` – the *range restriction* of p and q is the relation r that contain those tuples from p that end with an atom in q. This time, q must a set. Similarly, a tuple $\{p_1, \cdots, p_m\}$ only appears in r if p_m is found in q.
- `p++q` – the *relational override* of p by q results in relation r that contains all tuples in p except for those tuples whose first atom appears as the first atom in some tuple in q. Those tuples are replaced by their corresponding ones in q.

Now, let us explore a brief tutorial on writing model specifications in ALLOY.

5.1.2 Alloy Models

To illustrate the basic modeling techniques in ALLOY, we will consider a simple model of encryption.

Defining object types

An Alloy model consists of a number of signatures (`sig`), or sets of atoms that describe basic types in the model. Within these signatures, we can define relations that relate these basic types to one another.

First, we define a set of `Value` and `Key` atoms. We also define a relation `enc` that relates a `Key` to a `(Value, Value)` tuple, resulting in tuples of the form `(Key,Value,Value)`. Finally, we also define a singleton set `K1` of type `Key` that contains a single atom. All this is expressed with the following:

```
// define a set of Value objects
sig Value {}

// define a set of Key objects
```

```
sig Key {
  enc: Value one -> one Value
}

// singleton set K1
one sig K1 extends Key {}
```

We use the enc relation to represent the encryption function: a value, encrypted under a key, results in some value. The `one -> one` multiplicity constraint states that the encryption relation `(Value,Value)` for each key is a one-to-one function (there are other multiplicity symbols that can be used to describe injections and partial functions). In this relation, the tuple (k_1, v_1, v_2) reads: v_1 encrypted under k_1 gives v_2 and this generalizes to the representation *(key,plaintext,ciphertext)* [2].

Specifying constraints

After defining object types, we want to be able to specify properties about their behaviour by specifying constraints. Although the encryption relation has been sufficiently defined, we may need to add certain constraints to prevent ALLOY from producing trivial instances or counterexamples during its model-finding. One reasonable constraint is that two different keys have different encryption functions, which is not necessarily the case but is typically true. In other words, the encryption relations for any two keys must differ in at least one tuple. This can be written as a fact:

```
fact {
  // for all distinct k, k' chosen from the set Key
  all disj k , k' : Key |
    some (k.enc - k'.enc)
}
```

[2]the encryption function can be expressed in other equivalent forms, such as `(Value,Key,Value)`

The expression `k.enc` is a relational join between the singleton relation k and the `enc` relation, giving a binary relation `(Value,Value)` that represents the encryption function of key k. The last statement `"some (k.enc - k'.enc)"` says that there is at least one tuple in the difference of the encryption relations of two different key atoms.

Running analyses

With the model in place, we can proceed to perform some analyses. We can *simulate* the model by defining predicates on the model which can either be true or false depending on the instance. These are different from factual constraints, which are always required to be true in any instance. Simulation of a predicate is the process of instructing ALLOY to find an instance for which the predicate is true. To obtain an instance of the model without imposing any further constraints, we instruct ALLOY to simulate the empty predicate `show()`, which is effectively a non-constraint. In this case, ALLOY just has to find an instance that satisfies the object definitions and their factual constraints. Since ALLOY only performs bounded-analyses, we specify a maximum of 4 atoms in each object type (i.e. each `sig`).

```
pred show() {}
run show for 4
```

Figure 5-1 shows ALLOY's graphical output for this model of encryption. We added the constraint `#Value = 4` to enforce that there be exactly 4 `Value` objects in any instance. In the instance that was generated, ALLOY has chosen one key (`K1`) and four values. The arrow labeled `enc[Value2]` from `K10` to `Value3` means that `Value2` encrypted under key `K1` gives `Value3`.

Alternatively, we can also define assertions on properties of the model that we believe to be true. *Checking* is the process of instructing ALLOY to find an instance satisfying the model but not the assertion. For example, we might

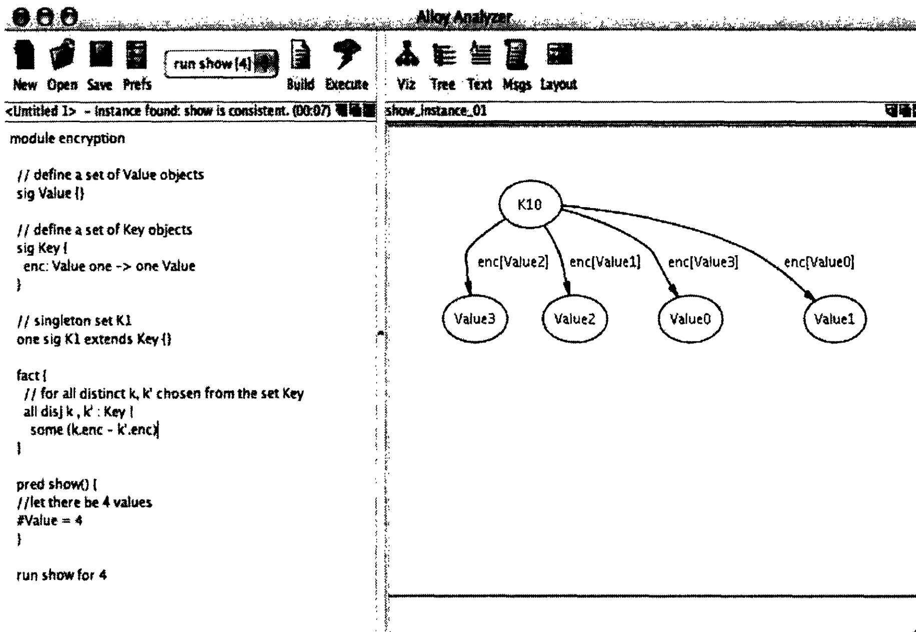


Figure 5-1: Alloy Analyzer - model of encryption

want to assert that the decryption functions of any two keys are different on at least one value since we believe it follows from the constraint that we put on encryption earlier. We write this as:

```
assert decryption {
  // the transpose (~) represents the decryption relation
  all disj k,k': Key | some (~(k.enc) - ~(k'.enc))
}
check decryption for 4
```

ALLOY will then proceed to find an instance that satisfies the model but acts as a counterexample to the assertion, an instance where two different keys have the same decryption relation.

All of ALLOY's analyses have to be run within a size bound on the number of atoms in each object type. ALLOY will check all models within this bound until a satisfying instance or counter-example is found. Given that the running time of the analysis increases exponentially with size, we can only check small models

[3] in reasonable time. However, the philosophy behind ALLOY is that if there are any inconsistencies with the model, they are extremely likely to appear even in small models.

5.2 Attacks Modeled with Alloy

In this section, we describe how ALLOY was used to model the cryptanalytic attack found against the DES triple-mode block-cipher ECB|ECB|OFB [5] and the guessing attack against the PIN-verification API in the IBM 4758 CCA [16]. This discussion aims to draw out the strengths of ALLOY that make it an excellent complement to OTTER.

5.2.1 Triple-mode DES

The block cipher *Data Encryption Standard* (DES) was developed by IBM in 1974 in response to a U.S. government invitation for an encryption algorithm. A block-cipher is basically an algorithm that encrypts data in blocks [4]. However, DES's key strength of 56 bits became a cause for concern as computing speed and technology evolved. Subsequently, multiple modes of operation were developed as a way of improving the strength of block-ciphers, in particular DES. A multiple-mode of operation is one that consists of multiple modes of operation chained together, each having its own key.

There are many modes of operation but the ones we are concerned with are the *Electronic Code Book* (ECB) and *Output FeedBack* (OFB) mode. The ECB mode of operation is the simplest way to use a block cipher. The data is divided into blocks and encrypted by the cipher one at a time. Therefore, two identical plaintext values will give the same ciphertext. The OFB mode of operation creates a stream of values by repeatedly encrypting a single initial value. This stream of values is then XORed with the input plaintext to produce the ciphertext. In this case, two similar plaintext values could yield different ciphertexts.

[3]small models contain 8-12 atoms per object type
[4]usually 64-bits

In 1994, Eli Biham showed that many multiple modes were much less secure than expected [4] and in 1996, Biham extended these results to conclude that all the triple-modes of operation were theoretically not much more secure than a single encryption [5], except the ECB|ECB|ECB mode. We now focus on the ECB|ECB|OFB triple-mode of operation and show how ALLOY managed to find the same cryptanalytic attack that Biham did. In this triple-mode of operation, the plaintext is encrypted with two ECB DES ciphers in series before it is fed into the OFB DES stream for a third encryption. The ECB|ECB|OFB triple-mode schematic is shown in Figure 5-2:

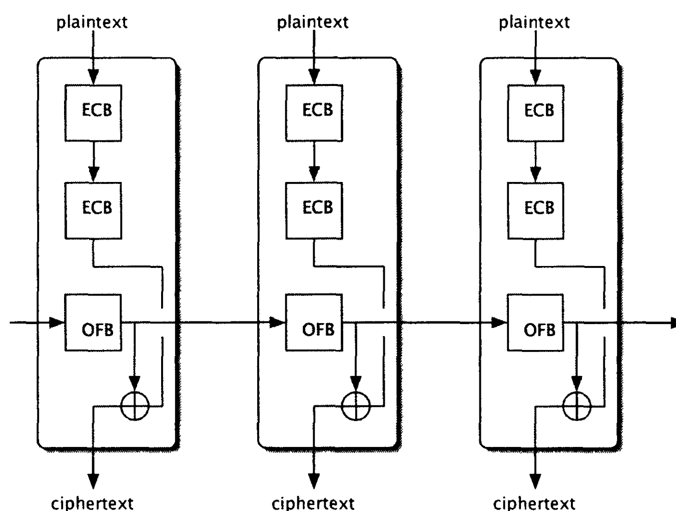


Figure 5-2: ECB|ECB|OFB triple-mode

To model this triple-mode of operation in ALLOY, we had to model i) encryption, ii) XOR, iii) the modes of operation and iv) the chaining of inputs to outputs between the various components. We have already modeled encryption earlier so let us proceed with the XOR operation. First, we defined a signature `Value`, a set of atoms with type `Value`, within which we defined the `xor` relation containing triples of the form (v_1, v_2, v_3), where each v is of type `Value` and $v_1 \oplus v_2 = v_3$.

```
sig Value {
  xor : Value one -> one Value
}
```

Next, we had to define the three properties of XOR:

- `all b : Value | xor.b = ~(xor.b)`

  the *commutativity* property states that $v_1 \oplus v_2 = v_2 \oplus v_1$ for any v_1, v_2. The relation `xor.b` is a binary relation (v_1, v_2) representing the pairs of values v_1 and v_2 that XOR to give b. By saying that `xor.b` is equal to its transpose, we are enforcing the constraint that if (v_1, v_2) exists in `xor.b`, then so must (v_2, v_1). This succinctly captures the required property.

- `all a, b, c : Value | xor.c.(xor.b.a) = xor.(xor.c.b).a`

  the *associativity* property states that $(v_1 \oplus v_2) \oplus v_3 = v_1 \oplus (v_2 \oplus v_3)$ for any v_1, v_2, v_3. The expression `xor.c.b` evaluates to the `Value` a such that $a \oplus b = c$. By putting the parenthesis in the correct places in the expression, we state the property as we would have done mathematically.

- `one identity : Value | no identity.xor - iden`

  the *identity* is the special value such that $identity \oplus v = v$ for any v. The expression `identity.xor` represents the XOR function for exactly one chosen value "`one identity`". The `iden` constant represents the universal *identity relation*, which is a binary relation containing tuples of the form (a, a) for any atom a defined in the ALLOY model. Therefore, the identity statement says that the identity's XOR relation with the all identity tuples removed is empty, which translates to saying that the identity XOR relation only contains identity tuples. Writing `identity.xor = iden` would be incorrect since this means that the XOR function must contain the identity tuples over atoms not defined in the `xor` relation.

Now, let us proceed to modeling the DES blocks and their input/output behaviour. We defined a set of blocks `sig Block`, where each block contains a plaintext `p`, an output feedback stream value `ofb` and an output ciphertext `c`. Next, we imposed an ordering on the blocks with the general `util/ordering` module packaged in ALLOY. The module also provides operations (`first(), last(), next(x), prev(x)`)

to help us navigate through the ordering: `first()` and `last()` return the first and last elements of the ordering while `next(x)` and `prev(x)` return the element after or before the element `x`.

```
open util/ordering[Block]
sig Block {
  p: one Value,
  ofb: one Value,
  c: one Value
}
```

Next, we proceeded to describe the data flow within these blocks. Since we were only attacking the ECB|OFB boundary, we used a single ECB encryption to represent the two ECB encryptions in series. This is actually equivalent if the single ECB encryption uses a key of twice the original length. The block's ciphertext is the result of the encryption of its plaintext under `K1`, a `Key` atom, XORed with the OFB value of the block. Also, the OFB value of the next block is the encryption of this block's OFB value under the OFB key.

```
fact {
  //the ciphertext is a function of the plaintext and the ofb value
  all b: Block | b.c = (K1.enc[b.p]).xor[b.ofb]

  // setting up the OFB sequence through the blocks
  all b: Block-last() | let b' = next(b) |
  b'.ofb = OFB.enc[b.ofb]
}
```

Finally, let us consider the cryptanalytic attack on this triple-mode of operation, as presented by Biham [5]. Assuming that the block size is 64-bits, providing the same 64-bit value v as plaintext 2^{64} times will result in 2^{64} ciphertext values of the form:

$$c_i = \{\{v\}_{K1}\}_{K2} \oplus o_i$$

where the braces {} represent encryption, o_i is the OFB stream value such that $o_{i+1} = \{o_i\}_{K_{OFB}}$ and *K1*,*K2* are the 2 ECB encryption keys. This basically allows us to isolate the OFB layer of encryption since it is equivalent to having a complete OFB stream XORed forward by a single value $\{\{v\}_{K1}\}_{K2}$. This means that it should only take 2^{64} amount of work to break the key used in the OFB layer. After peeling off the OFB layer, we can then proceed to break the double ECB layer by the standard meet-in-the-middle attack [23].

To break the OFB layer, we do the following:

1. choose an arbitrary value u.
2. encrypt u under all possible keys and get $u_1 = \{u\}_k$ and $u_2 = \{u_1\}_k$.
3. then check that $u \oplus u_1 = c \oplus c_1$ and $u_1 \oplus u_2 = c_1 \oplus c_2$ for a consecutive triple of ciphertext c, c_1, c_2.

On average, we need to try about $2^{64}/period$ times, where *period* is the expected period of the OFB stream, which is expected to be small. To observe that the cryptanalytic attack was successful, the sequence of ciphertexts must meet the following property:

$$\forall c_i, o_j \ (c_i \oplus c_{i+1} = o_j \oplus o_{j+1}) \vee (c_{i+1} \oplus c_{i+2} = o_{j+1} \oplus o_{j+2})$$

where (c_i, c_{i+1}, c_{i+2}) is a consecutive triple of ciphertexts and (o_i, o_{i+1}, o_{i+2}) is a consecutive triple of OFB stream values. We write this in ALLOY as:

```
pred FindAttack() {
  // the set contains the relevant properties that two sequential XORs
  // of outputs match two sequential XORs of OFB encryptions
  all b': Block - first() - last() | let b=prev(b'), b''=next(b') |
  some d:OFB.enc.Value | let d'=OFB.enc[d], d''=OFB.enc[d'] |
  b.c.xor[b'.c] = d.xor[d'] && b'.c.xor[b''.c] = d'.xor[d'']
}
```

```
run FindAttack for 8 but exactly 2 Key
```

The full model can be found in the Appendix A.1. Using this model, ALLOY was able to verify the attack with a scope of 8 atoms in 2 minutes and 25 seconds. However, it is important to note that the ciphertext property in our model could have been achieved with a few different sets of input plaintexts. Since ALLOY works non-deterministically, it could have found any one of those sets.

While the attack on the triple-mode block-cipher showed ALLOY's native power in dealing with mathematical properties in its relational logic, the following attack on PIN guessing demonstrates ALLOY's ability to work with sets.

5.2.2 PIN Decimalisation

To use Automated Teller Machines (ATMs), customers just have to swipe a debit card and punch in a PIN. In 2003, *Bond et al.* found that some banking security modules using *decimalisation tables* to generate PINs were susceptible to what they termed the *PIN-decimalisation* attack [16] [7], allowing the attacker to guess the customer's PIN in ~15 tries, thereby subverting the primary security mechanism against debit card fraud.

To understand the vulnerability, we must first understand a PIN generation algorithm that uses decimalisation tables. The PIN-derivation algorithm used in IBM's banking modules is as follows:

1. calculate the original PIN by encryption the account number with a secret "PIN generation" DES key
2. convert the resulting ciphertext into hexadecimal
3. convert the hexademical into a PIN using a "decimalisation table". The actual PIN is usually the first 4 or 6 numerals of the result.

The standard decimalisation table is shown below:

```
0123456789ABCDEF
0123456789012345
```

During decimalisation, the hexadecimal values on the top line are converted to the corresponding numerical digits on the second line. Some customers may choose to change their PINs. In this case, the numerical offset between the changed PIN and the real PIN is kept on the card and in the mainframe database. When the customer enters a PIN, the ATM subtracts this offset before checking it against the decimalised PIN.

The IBM *Common Cryptographic Architecture* (CCA) [12] is a financial API implemented by the IBM 4758, a hardware security module used in banking ATM networks. The CCA PIN verification command, `Encrypted_PIN_Verify` allows the user to enter a decimalisation table, the *primary account number* (PAN), also known as the bank account number, and an encrypted PIN. The encrypted PIN is first decrypted and then compared to the PIN generated from the PAN using the generation algorithm described earlier. The command returns true if the PINs match, false otherwise. Although we do not have access to the PIN encryption key, any trial PIN can be encrypted using the CCA command `Clear_PIN_Encrypt`.

The attack, as presented in *Bond et al.* [16], can be broken down into two stages: the first determines which digits are present in the PIN and the second determines the true PIN by trying all possible digit combinations. The first stage hinges on the vulnerability that CCA developers allowed clients to specify the decimalisation table. Instead, they should have hard-coded the standard decimalisation table into the commands.

We will present a simpler version of the attack against a single-digit PIN, thereby eliminating the need for the second stage. To determine the digit present in the PIN, we pass in the desired PAN, an encrypted trial PIN of 0 and a *binary* decimalisation table, where all entries are 0 except the entry for hex-digit i, which is 1. The trial PIN of 0 will fail to verify only if the encrypted PAN (before decimalisation) contains hex-digit i. However, instead of engineering this attack into our model, we would like for ALLOY to find this attack. Therefore, we set up the system by modeling the

attacker's decimalisation and trial PIN inputs and his actions after getting an answer from the Encrypted_PIN_Verify command.

First, we defined a set of NUM (numbers) and HEX (hexadecimal) atoms, as well as a singleton set TrueHEX representing the correct pre-decimalised PIN digit. Next, we defined a set of State atoms, where in each state the user performs an Encrypted_PIN_Verify API call using the decimalisation table dectab, which is a mapping from HEX to NUM and the trial PIN trialPIN. Here, we abstracted away the fact that the input PIN was encrypted. Also, guess represents the set of possible hexadecimal guesses that the attacker has to choose from in each state. As API calls are made, this set should decrease in size until the attacker uncovers the real hexadecimal value. He can then compare this value against the standard decimalisation table (shown earlier) to find the real PIN.

```
open util/ordering[State]

// set of numbers
sig NUM{}

// set of hexadecimal digits
sig HEX{}

// one hex digit representing the
one sig TrueHEX extends HEX {}

// in each state, the attack picks a trialPIN, a decimalisation table
// and has a set of guesses of what the pre-decimalised PIN is
sig State {
  trialPIN: one NUM,
  dectab: HEX -> one NUM ,
  guess: set HEX
}
```

Next, we modeled the event of the attacker choosing a particular number as a trial PIN. We used a predicate to constrain the possible guesses `guess` in two states s, s' based on the choice of trial PINs in state s. If the trial PIN verifies, then the pre-decimalised PIN must contain the hexadecimal values corresponding to the trial PIN in the given decimalisation table, therefore, we intersect the attacker's current possibility set with those values. Otherwise, we intersect the attacker's current possibility set with the hexadecimal values corresponding to the numbers other than the trial PIN in the decimalisation table. In either case, the attacker is gaining knowledge about the value of the pre-decimalised PIN.

```
pred choose(s,s': State) {
// if trial PIN verifies
( (s.trialPIN = s.dectab[TrueHEX]) &&
  (s'.guess = s.guess & s.dectab.(s.trialPIN)) )
||
// if trial PIN does not verify
( (s.trialPIN != s.dectab[TrueHEX]) &&
  (s'.guess = s.guess & s.dectab.(NUM-(s.trialPIN))) )
}
```

Finally, we set up the attack by defining transition operations across the states. Here, we defined a predicate `init` on a state to set the possibility set to that of all the hexadecimal digits. The predicate `Attack` is meant to simulate the attack as the attacker makes the API calls with different decimalisation tables and trial PIN choices. The predicate defines a transition relation between each state and the next and also sets constraints on the last state that the correct value be found. Then, by running this `Attack` predicate, ALLOY will find an instance satisfying all the constraints in `Attack` which also means satisfying the `init` and `choose` predicates.

```
pred init(s:State) {
  // in the first state s, the attacker has no information,
  // so his set of guesses covers all the hexadecimal values
```

```
  s.guess = HEX
}
pred Attack () {
  // initialize on the first state atom
  init(first())

  // for all states other than the last, the attacker makes
  // a choice after issuing an API call
  all s:State-last() | let s'=next(s) |  choose(s,s')

  // in the last state, the attacker finds the correct HEX
  #(last().poss) = 1
}
```

ALLOY was able to find an instance of the attack for 6 `State` atoms, 16 `HEX` atoms and 10 `NUM` atoms in 28 seconds. The last two constraints in the `Attack` predicate demonstrate that some creative constraining is required to force ALLOY not to find trivial instances. Without those constraints, ALLOY would have allowed the attacker to discover the correct value on the very first try, a very "lucky" scenario that is unlikely to happen. This is an instance of a more general difficulty with constraint modeling – while it is easy for ALLOY to reason about the worst or best-case scenarios, it is extremely challenging to model the notion of an "average" case. In this example, ALLOY was able to find the attack but did not choose the decimalisation tables as we did which would guarantee the best performance on average.

Chapter 6

Trusted Platform Module

As software on computing platforms get increasingly complex, the number of security bugs also increases. Subsequently, protecting sensitive data through a software-only security mechanism suffers from two inherent weaknesses. First, large software will have implementation bugs and therefore can potentially be attacked. Second, it may be impossible to tell if software has been attacked. In fact, experts in information security have concluded that some security problems are unsolvable without a bootstrap to protected hardware [22]. The *Trusted Platform Module* (TPM) is the *Trusted Computing Group's* (TCG) [1] answer to this problem. In this chapter, we present an overview of the TPM.

6.1 History of Trusted Computing

Prior to 1999, although there were a number of secure coprocessors (separate shielded computing units like the IBM 4758) that were able to attest to the integrity of their software, there were no full "computing platform" security solutions that could do the same. Subsequently, users were assured of security only by blindly trusting that the platform was doing what it was supposed to do. In 1999, the *Trusted Computing Platform Alliance* (TCPA) was formed to address this issue at two levels: increasing the security of platforms and enabling trust mechanisms between platforms. This led to the development of *Trusted Platforms*. Later in 2003 the TCPA became the

Trusted Computing Group (TCG), which took on the necessary burden of encouraging worldwide industry participation in such an endeavour.

6.1.1 Trusted Platforms

Under the TCG's definition of trust, Trusted Platforms are platforms that can be expected to always behave in a certain manner for an intended purpose. Furthermore, the users do not have to make this decision blindly but rather can request for the platform to prove its trustworthiness by asking for certain metrics and certificates. A Trusted Platform should provide at least these basic features:

- **Protected Capabilities**: the ability to perform computation and securely store data in a trustworthy manner.
- **Integrity Measurement**: the ability to trustworthily measure metrics describing the platform's configuration.
- **Attestation**: the ability to vouch for information.

In essence, these features mean that a Trusted Platform should be protected against tampering, be able to attest to the authenticity and integrity of platform code and data, perform guarded execution of code and maintain the confidentiality of sensitive information. To establish trust, a Trusted Platform can reliably measure any metric about itself and attest to it. Some useful metrics include the software loaded on the platform and any device firmware. The user will then need to verify these metrics against trusted values obtained separately to decide if this platform is trustworthy. Immediately, we can see that these abilities can enable a variety of new applications.

6.1.2 Use Scenarios

A solution enabling clients or servers to trust one another's software state gives hope to many commercial applications. A Digital Rights Management (DRM) solution is possible since servers can have confidence that client machines are not subverting the media policies. Auctions and various fair-games like online poker could leverage

on server-client trust relationships to execute fair game play. Massive computation requiring confidentiality can be outsourced to computing grids, as long as the grid attests that it will not reveal any secrets. The TCG's solution to turning regular computing platforms into Trusted Platforms resulted in the development of a specification known as the *Trusted Platform Module.*

6.2 TPM Design

The TCG specification for this single tamper-resistant chip comprises both informative comments and normative statements but does not come with an implementation. Rather, it is up to the individual chip manufacturers to work with the specification requirements.

6.2.1 Features

Conceptually, the TPM will create three *Roots of Trust* on its parent platform that are used to effect trust and security mechanisms:

- **Root of Trust for Measurement (RTM)**: reliably measures any user-defined metric of the platform configuration. The RTM starts out as trusted code in the motherboard's Boot ROM but extends its trust domain during system boot to the entire platform through the process of "inductive trust". In this process, the RTM measures the next piece of code to be loaded, checks that the measurement is correct and then transfers control. This process of extending the trust domain continues until the a trusted operating system is booted.

- **Root of Trust for Reporting (RTR)**: allowed to access protected locations for storage, including the *Platform Configuration Registers* (PCRs) and non-volatile memory, and also attests to the authenticity of these stored values using signing keys. PCRs are storage registers that not only store values but also the order in which the values were put in.

- **Root of Trust of Storage (RTS)**: protects keys and sensitive data entrusted to the TPM. The RTS basically refers to all the key management functionality, including key generation, key management, encryption and guarded decryption.

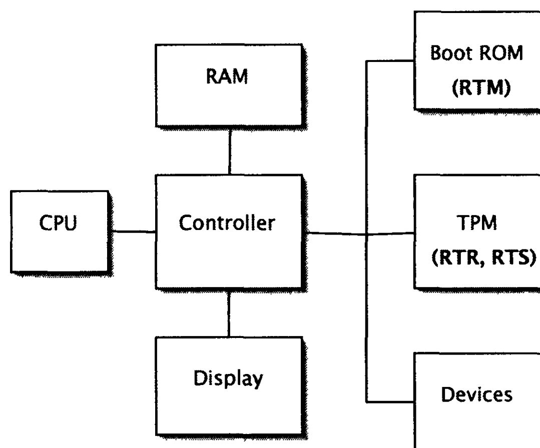


Figure 6-1: Roots of trust in a TPM-enabled PC architecture

Figure 6-1 shows how the TPM fits into a regular PC architecture and where the various roots of trust reside. These roots of trust provide the means of knowing whether a platform *can* be trusted. It is up to the client to decide whether the platform *should* be trusted. If the client trusts the credential certifying the correct manufacturing of the platform, the RTM, RTR and the RTS, then he can trust the values measured by the RTM, stored by the RTS and reported by the RTR. The client must then compare these reported values against other credentials (e.g. those issued by a trusted third party) to determine if the platform state is "trustable".

6.2.2 Trusted Platform Module API

In section, we will introduce the TPM's API by describing it in the order that a typical user would encounter them when using a TPM-enabled platform. An owner of a brand new TPM must first take ownership of the chip. This involves setting an owner password and the TPM generating for itself a *storage root key* (SRK) and a master secret (*tpmProof*). These secrets enable much of the TPM's functionality. This

module also contains operations that allow the owner to set operational flags, disable and enable certain operations and reset or deactivate the TPM. All this functionality can be described as the *initialization* module.

To start using the TPM, the user should generate a TPM identity key and some encryption keys. The *key management* module provides methods to generate different types of keys, assign and change passwords to keys and migrate keys to other entities for backup purposes. The different key types include storage keys, signing keys, identity keys and binding keys.

Some API calls and access to most TPM resources require password authorization. Using the *authorization* module, users open a transport session to each resource (keys, memory indices, etc) by providing the correct usage password. Since this session will be used to manage all future API calls using that resource, it becomes a one-time overhead that prevents replay attacks and relinquishes the need for multiple password submission. Furthermore, these sessions can be securely cached onto disk if memory in the TPM runs out.

The simple password-based model of authorization made it difficult to micro-manage TPM functionality. If the TPM owner password is revealed to a trusted user in order for him to perform a particular owner-authorized operation, all other owner operations are effectively under this user's control. Therefore, in the second major release of the TPM specifications (v 1.2), the *delegation* module was introduced to allow the delegation of privileges to other users and/or system processes.

At this point, the user may desire the ability to make specific measurements of platform characteristics. These measurements represent the software state of the platform configuration and can be used to determined when certain secrets or code should be released or executed. The *integrity measurement* module provides API methods to "extend" the metrics stored in Platform Configuration Registers (PCRs) with the following formula:

$$Extend(x) : PCR_{new} = SHA1(PCR_{old} \mid x)$$

This means that the order in which metrics were recorded is also captured. This module also manages functionality to read and write values in the TPM's non-volatile memory. Subsequently, these measured metrics have to be communicated in a trustworthy manner, which is the job of the *integrity reporting* module. Reporting is basically a signing operation that be performed on various keys, measurements and audit logs recorded in the TPM.

Finally, the user may want to enable the TPM for applications in more than one trust domain, meaning that sensitive information under these multiple trust boundaries has to be logically separated. The user can use the TPM's *protected storage* module to bind and seal application secrets. Binding is the encryption of sensitive data under a particular key while sealing is the binding of data to a particular platform configuration. Sealed data is only released under a specific platform configuration, enforced by checking the PCR values.

Although there are many other micro-modules in the TPM that deal with maintenance, credential handling, general purpose I/O and the TPM's monotonic counters, the modules described above represent the more important and useful ones. In chapter 7, we will analyze some of the more interesting modules in greater detail.

Chapter 7

Security Analysis of the TPM

This chapter covers the modeling and analysis of the *Trusted Platform Module* (TPM) API version 1.2 specification [1]. The API exposed by the TPM covers a wide range of functionality, including taking ownership of the TPM, initializing the chip, running self-tests, generating different types of keys, establishing transport sessions, auditing, encrypting and decrypting, delegating commands, migrating data to another TPM. We break these operations down into five major areas of functionality and present them individually, together with the techniques and tools used to analyze them.

7.1 Initialization

The initialization commands manage the operational state of the TPM – from taking ownership of the TPM to disallowing certain API commands to resetting and clearing the chip's memory. We focus on the commands that manipulate the `TPM_PERMANENT_FLAGS`, a set of system flags that persist in non-volatile memory, and how we analyzed them for inconsistencies. All the modeling in this section was done with OTTER's first-order language.

To find an attack within this section of the API would be to find an inconsistent system state – a state where the permanent flags are set in a way that violates a security policy. To do this, we first need a way of capturing the state of the system. Secondly, we need a way of exploring all possible state configurations and thirdly, we

need to understand the security policies around these API calls to understand what constitutes an inconsistent state.

To capture the relevant permanent flags, we used a single predicate,

```
PermFlags(x1,x2,x3,x4,x5,x6)
```

that stores the values of the permanent flags during a state configuration. The flags and what they signal are shown in Table 7.1. The owner is an entity that has special privilege to perform certain restricted capabilities. The operator is an entity that only has special privilege to manage the TPM's operational state. To explore all possible

flag	description
x1	set to 'T' to disable the TPM (commands that use resources fail)
x2	set to 'T' to allow installing the TPM owner
x3	set to 'T' to deactivated the TPM (all commands fail)
x4	set to 'T' to disable clearing of the TPM owner
x5	set to 'T' when the TPM has an owner
x6	set to 'T' when the TPM has an operator password

Table 7.1: TPM Permanent Flags

state configurations with the API calls in this section, each API call is modeled as an implication rule that takes in a `PermFlag` clause and produces a modified `PermFlag` clause. In strict OTTER jargon, each API FOL clause unifies with an existing `PermFlag` clause before resolving to produce a modified `PermFlag` clause. For example, we write the API call `TPM_SetOwnerInstall`, which sets the permanent flag that allows the ability to insert an owner, as:

```
-Bool(y) | -U(physicalPresence) | -PermFlags(x1,x2,x3,x4,F,x6) |
PermFlags(x1,y,x3,x4,F,x6).
```

The predicate `Bool(y)` is used to represent the caller's choice of whether to set the flag. `-U(physicalPresence)` says that the user must be asserting physical presence for this command to succeed. To assert physical presence is to literally perform an action on the chip itself (what the action is depends on the chip). In this case, the API call requires `x5` to be false, meaning that it requires that no owner be present.

Some of the more important commands that manipulate these permanent flags are listed below:

- `TPM_SetOwnerInstall` – sets the permanent flag that allows inserting an owner. Physical presence must be asserted for the command to succeed.
- `TPM_OwnerSetDisable` – an owner-authorized command that sets the flag disabling the TPM.
- `TPM_PhysicalEnable` – sets the TPM disable flag to FALSE. Disabling the TPM means that all commands that use resources will fail. Physical presence must be asserted.
- `TPM_PhysicalDisable` – sets the TPM disable flag to TRUE. Physical presence must be asserted.
- `TPM_PhysicalSetDeactivated` – sets the TPM deactivated flag. Deactivating the TPM means that all commands fail.
- `TPM_SetOperatorAuth` – installs the operator password and sets the operator password flag to TRUE.
- `TPM_TakeOwnership` – installs an owner into the TPM, sets the owner flag to TRUE and stores the owner password.
- `TPM_OwnerClear` – removes the owner, sets the owner and operator flags to FALSE
- `TPM_DisableOwnerClear` – sets the flag that disables the clearing of the owner.

Security Policies

Although the documentation did not specify any security policies, there are certain combinations of these `TPM_PERMANENT_FLAGS` that can subvert the operation of the TPM. We call these "inconsistent states" and three of them are:

1. there is no owner installed, the flag allowing the installation of an owner is set to FALSE and the user cannot assert physical presence. In this case, an owner can never be installed.
2. there is no owner nor operator installed and the flag allowing the installation of an owner is disabled. Again, an owner cannot be installed in this case.
3. once an owner is installed, a user without physical access should not be able to re-install a new owner.

Given the initial default permanent flag values, our initial state clause is

```
PermFlags(T,T,T,F,F,F)
```

From there, we want to see if we can reach any inconsistent `PermFlag` state clauses using the API commands listed before.

Discussion and Results

OTTER's analysis of the model terminated, showing that neither of these inconsistent states could be reached. However, when the user in our model was given physical access to the chip (by adding the clause `U(physicalPresence)` to the sos list), our model found that almost all states were reachable. This behaviour is in line with the intentions of the TPM designers, where asserting physical presence overrides all.

Our method of capturing all interesting state in a single predicate can lead itself to modeling problems when there is an extremely large amount of state. In particular, OTTER tends to slow down when dealing with large clauses (i.e. clauses with many variables). Also, the readability and management of the model becomes tedious.

Therefore, we suggest a technique for dealing with large amounts of state. If the state can be separated into collections such that commands only work on one collection at one time, then defining multiple predicates, one for each collection of state, will help the model's readability and management. The technical advantage is that most of the inference rules representing API calls need only be concerned with a single state predicate. However, since the API calls only modify a single predicate at

a time, we need a way to keep track of which clauses belong together. To accomplish this, we use a single overarching state predicate. We illustrate this technique with the TPM.

The TPM further contains some other flags – the `TPM_STCLEAR_FLAGS` – that are reset on the command `TPM_Startup`. To add these state flags into our model, we define a separate predicate,

```
STClearFlags(x1,x2,x3,x4,x5)
```

for the five relevant volatile system flags and define a larger `State` predicate that contains both the `STClearFlags` and `PermFlags` predicates:

```
State(PermFlags(...),STClearFlags(...))
```

The `TPM_SetOwnerInstall` command, which only deals with the permanent flags, is modeled as:

```
-Bool(ybool) | -U(physical presence) |
State(PermFlags(x1,x2,x3,x4,F,x6),y) |
State(PermFlags(x1,y,x3,x4,F,x6),y)
```

The `TPM_Startup` command, which only deals with the volatile flags, is modeled as:

```
-Bool(ybool) | -State(x,STClearFlags(x1,x2,x3,x4,x5)) |
State(x,STClearFlags(x1,T,F,F,xPCR)).
```

and the command `TPM_SetTempDeactivated`, which uses both volatile and permanent flags, is modeled as:

```
-U(physicalpresence) |
-State(PermFlags(x1,x2,x3,x4,x5,T),STClearFlags(x1,x2,x3,x4,x5)) |
State(PermFlags(x1,x2,x3,x4,x5,T),STClearFlags(T,x2,x3,x4,x5)).
```

We find that most API calls are concerned with only one state predicate and therefore this technique will help make the model much more readable and also prevent OTTER from reasoning about too many useless variables and constants.

7.2 Key Management

At the crux of the TPM's security features is the key management module. This module provides functionality that handles the generation, protection and control of key material, all of which are integral to the protection of sensitive data.

Subsequently, it is also the key management modules in security systems that are most likely to be vulnerable as demonstrated by Bond and Youn after they found attacks on the *key-typing* mechanisms in both the IBM 4758 CCA and VSM [6][24]. Following in their footsteps, we modeled the key management API commands in OTTER so as to leverage on OTTER's ability to perform extensive API-chaining easily.

There are seven different types of keys used in the TPM as shown in Table 7.2. In this section, we focus on storage keys. These keys have a number of control

key type	**intended usage**
signing	used for signing operations
identity	used to certify TPM knowledge of data
authchange	used in transport session encryption
bind	used to encrypt data
legacy	keys in older versions of the TPM are typed as legacy keys
storage	keys used to protect and store sensitive data, including key material

Table 7.2: TPM Key Types

parameters. First, there are fields that describe various characteristics of the key, such as the key size, the generation algorithm and the intended usage schemes. Second, each key comes with two passwords - a usage password and a migration password. The usage password authorizes usage of the key for encryption and decryption operations but does not allow the discovery of the actual key value. The migration password authorizes the user to export a key under a specified public key.

The two most basic operations in the key management module are the generation and loading of key material:

- `TPM_CreateWrapKey` enables the user to request for the generation of an asym-

metric key from the TPM. This key can either be a *migratable* or a *non-migratable* key, where migratability refers to the property of being transferrable to another TPM. This is usually done for backup or storage purposes. Furthermore, the user is allowed to provide the usage and migration passwords for the new key. However, in the case of non-migratable keys, the TPM will set the new key's migration password to be a special value known as *tpmProof*, which is the TPM's master secret. Every key is generated under a parent key, except the storage root key, which is generated as part of the TPM initialization process. A diagrammatic representation of the TPM storage key hierarchy is shown in Figure 7-1. The newly generated key is then encrypted under the parent key and the resulting blob is returned to the user in a special data structure.

- `TPM_LoadKey` enables the user to load a key blob into the TPM's protected memory by specifying the correct parent key for decryption. At the same time, the TPM also performs some checks to ensure that the key is well-formed and that it adheres to the security requirements (e.g. key strength).

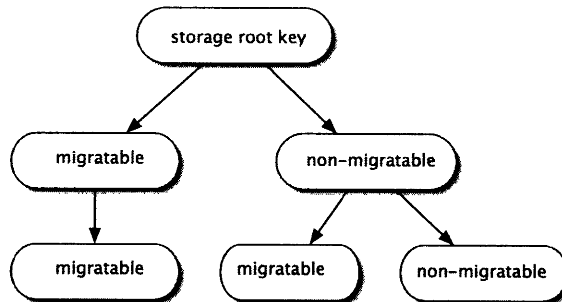


Figure 7-1: TPM Storage key hierarchy

For migratable keys, the TPM also provides functionality to manage the transfer and control of these keys:

- `TPM_AuthorizeMigrationKey` is an owner-authorized command for specifying the *migration-key*, meaning that the caller must present the owner password to use the command. The migration-key is the key that wraps and protects the

migrating key. It is also the key at the destination TPM that will be used to receive the migrating key.

- `TPM_CreateMigrationBlob` creates a special *migration-blob* data structure that contains the migrating key encrypted under the migration-key and OAEP encoded [3].

- `TPM_ConvertMigrationBlob` allows the receiving TPM to convert the migration-blob data structure back into a key-blob. To use the imported key, the key-blob must then be loaded into the TPM using `TPM_LoadKey`.

The commands just described make up the key management functionality in version 1.1 of the TPM. However, after receiving feedback from numerous industry partners, the TPM designers felt that having only migratable and non-migratable keys was insufficient. They wanted to a new type of migratability that would bridge the two extremes. Therefore, in version 1.2 of the design, they introduced the concept of *certifiable migration* – migration that was controlled and therefore could be certified against.

With certifiable-migration, users would have the ability to specify trusted third-parties as *migration authorities* (MAs) and ensure that a key is not migrated to a particular destination without the permission of the MA. In the TPM version 1.2, three new certifiable-migration key (CMK) commands were introduced:

- `TPM_CMK_CreateKey` is an owner-authorized command that creates keys whose migration is controlled by a *migration authority* (MA) entity. The migration authority is identified by a public key specified by the owner. This command is very similar to `TPM_CreateWrapKey` except that it only creates certified migratable keys that can only be migrated by `TPM_CMK_CreateBlob`

- `TPM_CMK_CreateTicket` is an owner-authorized command producing a hash (known as a *sig-ticket*) certifying that a public key verifies a particular signature on some data. Its intended use is for migration authorities to specify the migration-key for a CMK by signing off on the migration-key's hash.

- `TPM_CMK_CreateBlob` is used to migrate certifiable-migration keys. There are two migration mechanisms - direct migration to a *migration selection authority* (MSA) key, hereafter known as the destination-key, and indirect migration via a *migration authority* (MA) key. To perform direct certifiable migration:

  - authorize $DEST_{pub}$ (public half of the destination key) as the migration-key using `TPM_AuthorizeMigrationKey`.
  - create a key with $DEST_{pub}$ as the destination key using `TPM_CMK_CreateKey`.
  - use `TPM_CMK_CreateBlob` to create a CMK migration-blob with the output of the two previous commands.
  - use $DEST_{priv}$ to receive the key at the destination.

  To perform indirect certifiable migration:

  - authorize MA_{pub} as the migration key using `TPM_AuthorizeMigrationKey`.
  - create a key with MA_{pub} as the MA using `TPM_CMK_CreateKey`.
  - request that the MA certify the destination key $DEST_{pub}$ by signing a blob containing digests of MA_{pub}, $DEST_{pub}$ and CMK_{pub}. Then obtain a sig-ticket from `TPM_CreateTicket` using the signature.
  - create the migration-blob using `TPM_CMK_CreateBlob` which performs the relevant checks to ensure that the MA did approve the migration to the destination key.
  - use $DEST_{priv}$ to receive the key at the destination

  In general, it is either the owner of the CMK or the migration authority who picks the destination key. The owner picks it at CMK creation time while the migration authority specifies it by providing a signature on the intended destination key.

Security Policies

In the specifications for these API commands, the designers mention a number of security policies under the "informative comments" sections. We list the more crucial ones here:

1. keys must meet certain security requirements, depending on their type, before they will be loaded into the TPM. (e.g. a storage key must be 2048 bits long).
2. only keys with a parent key of type storage can be loaded using `TPM_LoadKey`
3. a non-migratable key can only be loaded if its migration password is *tpmProof*

However, the designers did not specify the higher level security policies that when compromised would result in serious attacks against the key management module. Nevertheless, we list some of them here

1. the master secret *tpmProof* should never be discovered
2. non-migratable keys cannot be created under migratable parent keys
3. non-migratable keys cannot be injected into a TPM

Model and Results

Our OTTER model of the key management module consisted of 7 API commands and a number of other attacker abilities such as the ability to hash, generate raw key material and encrypt arbitrary data. The APIs were modeled as described in section 4.2, where the API command is written as a clause containing both positive and negative literals, the negative literals representing inputs or requirements for the command and the positive literals representing the effects and return values of the command. To give an example, we show the OTTER description for the `TPM_CreateWrapKey` command for generating non-migratable keys. The lines starting with "%" are comments.

```
% TPM(x) = TPM has loaded x
% U(x) = User knows x
```

```
% TPMKEY(x): the TPM generated this key x

% KEY(x,xuauth,xmauth,xtype,xmig) represents a key
%   x = key identifier
%   xuauth = usage password
%   xmauth = migration password
%   xtype = STORAGE, IDENTITY, etc
%   xmig = MIG, NMIG (migratability)

% priv(x) identifies the private half of key x
% pub(x) identifies the public half of key x

% ASYM(priv(x),xuauth,xmauth,pub(y)) represents a key-blob
%   priv(x) = private key of x
%   xuauth = usage password
%   xmauth = migration password
%   pub(y) = encrypted under public key of y

% TPM_CreateWrapKey for a non-migratable key (next four lines)
% putting in key generation parameters
-KEY(x,xuauth,xmauth,xtype,NMIG) | -TPMKEY(x) |

% parent key of correct type, flag and user presents correct password
-KEY(y,yuauth,ymauth,STORAGE,NMIG) | -U(yuauth) |

% TPM has loaded the parent key
-TPM(y) |

% returns TPM_STORE_ASYMKEY blob
ASYM(priv(x),xuauth,xmauth, pub(y)) .
```

Although we do want to model the attacker being able to pass in any value as input to these API commands, such a model would be exploring too many search paths unlikely to yield results, thereby polluting the analysis. Therefore, our model uses the idea of data partitioning that was first introduced by *Youn et al* [24]. We created the predicates `ASYM` and `MIGBLOB` to represent asymmetric key blobs and migration blobs respectively. This way, the attacker in our model would never be able to pass off a migration blob as input to a command expecting an asymmetric key blob. The real command would probably throw an exception since the data structures are so different. However, we did not go so far as to define separate predicates for regular asymmetric keys and certifiable-migration keys because these keys, although different in content, utilize the same data structure. By using the same predicate, we are allowing for *blob-confusion*, where a certifiable-migration key can be passed off as a regular asymmetric key and vice versa.

Although we did not find any "real" attacks on TPM v1.2 specification, we found two "near-miss" attacks that are only possible in a slightly flawed implementation of the TPM. Those attacks are covered in chapter 8.

7.3 Protected Storage

In this section, we describe the commands that allow TPM users to protect sensitive data under two security measures - a cryptographic key and the software state of the platform. Only with the correct key and the platform in the specified software state will the TPM reveal the secret.

The first measure is accomplished via encryption and the second is accomplished by specifying *Platform Configuration Register* (PCR) values. Recall from chapter 6 that the PCR are registers that contain hashes of software that were loaded onto the platform. As more software is loaded, the PCR is *extended* by setting the new PCR value to be the hash of the old PCR value concatenated with a digest of the loaded software. Using this extension operation, a register can capture what software was loaded and the order in which the software was loaded in a single value. Therefore,

by specifying values for each register (i.e. a PCR constraint), the user can select the desired software state for secret release. The three commands that effect this feature are:

- `TPM_Extend` extends a specified PCR with a specified value.
- `TPM_Seal` allows software to seal a secret to both a key and a "trusted" platform configuration. It does this by encrypting the secret with the specified key and appending the required PCR values to the encrypted blob in a tamper-proof data structure
- `TPM_Unseal` decrypts and reveals a secret sealed by `TPM_Seal` if the user provides authorization to use the correct key and the platform configuration state matches.

Since there were only three commands in this module, we decided to analyze the commands together with other modules in the API. In the process, we had to take into consideration that the TPM design specifications require a minimum of 16 registers in the chip. However, if we proved that a single-register system could provide the functionality and power of a multi-register system, our models would only need to consider a single register.

To show that a multi-register system is equivalent to a single-register system, we employed the idea of a *simulation*. A simulation is an algorithm describing how one system mimics the functionality of another system; to show equivalence between the two systems, we have to define simulations in both directions. It is easy to see how a multi-register system can simulate a single-register system – it simply uses only one of its registers. As for the other direction, we were unable to find a satisfactory solution but nevertheless we will describe one of our incorrect approaches.

The single-register system can simulate the multi-register system by extending its register every time a piece of software is hashed into any register in the multi-register system. This way, the single register captures all the software as well as the order that the software was hashed. However, this gives rise to problems when multi-register PCR constraints are re-written for this single-register system. Recall once

again that a secret is released only when the values in the registers match those in the PCR constraints specified in the secret. Consider a secret that is released only if the "secure network manager" is loaded after the "secure kernel" and that its multi-register PCR constraint consists of a value for just one register. Although the single-register system could have recorded the "secure kernel" before the "secure network manager", it could have recorded with many other software modules in between. To rewrite the multi-register PCR constraint for the single-register system, the user would have to consider all possible interleavings of other software between the kernel and the network manager and provide multiple PCR constraints such that the secret would be released as long as one was satisfied. Considering that some unknown software module (e.g. a kernel extension) might be loaded as well, this becomes absolutely impractical, if not impossible for the user. Nevertheless, we will present the ALLOY model for the above simulation.

Model

We defined an ALLOY model containing `PCR` atoms. One of these atoms, `sPCR`, represents the single-register system and the other atoms collectively represent the PCRs in the multi-register system. Software programs loaded into the system are identified by `Code` atoms. At the same time, they also represent the software digests used to extend the registers. The extension operations done to each `PCR` object were ordered with a one-to-one mapping to a set of ordered `Extend` atoms.

```
open util/ordering[Extend]

sig Extend {}

// set of software objects
sig Code {}

abstract sig PCR {
  sequence: Extend lone -> lone Code
```

```
}

// set of registers for the multi-register system
sig mPCR extends PCR {}
// a single register for the single-register system
one sig sPCR extends PCR {}

fact {
  // all software loaded is recorded
  mPCR.sequence[Extend] = Code
  sPCR.sequence[Extend] = Code

  // extensions are contiguous
  all m: mPCR | let exts = m.sequence.Code, mx = max(exts) |
  all e: Extend-exts | e in nexts(mx)

  // software is loaded one at a time
  // but can be extended into one or more mPCR registers
  all disj c,c':Code | let pcrs =  (sequence.c.Extend &
  sequence.c'.Extend) |
  !(some (pcrs.sequence.c & prevs(pcrs.sequence.c'))
  &&
  some (pcrs.sequence.c & nexts(pcrs.sequence.c')))
}
```

We also defined a set of `Secret` atoms that represent secrets that are only revealed if a particular platform configuration state is met. The required platform configuration state for the release of a particular secret is represented as pairs of `(Code, Code)` constraints, where (c_1, c_2) means that the secret requires c_1 to have been loaded before c_2. The secret may also require some software to be present before release but this software has no temporal constraints on other pieces of software. Such

information is not captured because that piece of software, if unloaded, can be loaded unconditionally to allow the release of the secret. The PCR constraints were expressed as:

```
  sig Secret {
    constraints: Code -> Code
  }
```

Next, we had to model the simulation described earlier by defining a predicate on the system:

```
pred Simulate(){
  // for any register that was extended with c,c'
  all disj c,c':Code | some p : sequence.c.Extend & sequence.c'.Extend |
  {
    // if c was recorded before c' in p
    // then let it be the same for the single register
    p.sequence.c in prevs(p.sequence.c')
    =>
    sPCR.sequence.c in prevs(sPCR.sequence.c')

    // if c was recorded after c' in p
    // let it be the same for the single register
    p.sequence.c in nexts(p.sequence.c')
    =>
    sPCR.sequence.c in nexts(sPCR.sequence.c')
  }
}
```

ALLOY was able to find an instance of this simulation in 32 seconds.

7.4 Delegation

In the TPM version 1.1 design, the TPM-owner was responsible for many managerial and administrative tasks. In version 1.2, the concept of delegation was introduced to give the owner the ability to release some of these responsibilities in a controlled fashion, thereby shifting towards a role-based administration based on the principle of least privilege.

The idea was to to enable the owner to delegate specific owner privileges to specific entities, rather than reveal the owner password, which would give the entity all owner privileges. The entity can either be an actual user, in which case the owner will have to hand him the delegated password, or a process identified by PCR values.

The TPM delegation design consists of delegations assigning an entity to the delegated task and delegation families representing groups of delegations. The families are there largely for administrative purposes and can be thought of as security groups. The TPM owner can create and manage delegation families. The owner can also create delegations by choosing a new password, the target tasks, the target entity and the target delegation family. Every delegation and delegation family has a *verification count* that is used to determine if a delegation is still valid. To invalidate existing delegations, the owner must increment the verification count of the delegation family, thereby invalidating all delegations belonging to that family, and then proceed to update the verification counts for each of the desired delegations.

All delegation information is stored within the TPM in the family and delegation tables but because the number of delegations is variable and possibly large, some delegations exist as tamper-proof blobs externally. Tables 7.3 and 7.4 show the data fields in delegation and delegation family tables respectively.

The following API commands make up the delegation section of the TPM:

- `TPM_Delegate_Manage` is an owner-authorized command that manages the family row. It can create, remove, disable or enable use of a delegation family.

- `TPM_Delegate_CreateKeyDelegation` creates a delegation to use a particular key. Only one who knows the key's usage password can authorize this operation.

Field	Description
ASCII Label	delegation name
Family ID	the family that this delegation belongs to
Verification Count	32-bit number
Capabilities	the task that is being delegated and the entity that it is delegated to

Table 7.3: Delegation Table fields

Field	Description
Family ID	32-bit number
Row Label	the family name
Verification Count	32-bit number for invalidating delegations belong to this family
Family Flags	enable/disable/admin flags

Table 7.4: Family Table fields

- `TPM_Delegate_CreateOwnerDelegation` is an owner-authorized command that creates a delegation to perform a particular owner-authorized task.

- `TPM_Delegate_LoadOwnerDelegation` loads an owner delegation. Key delegations are not loaded; instead, they are presented when the user attempts to use a key.

- `TPM_Delegate_UpdateVerification` is an owner-authorized command that updates a delegation's verification count to that of the delegation family's.

Model and Discussion

Our previous ALLOY models were static models – models that describe a single snapshot of the system. For the delegation module, we decided to use a dynamic model – one that models state transitions – since it could capture the changes in verification counts and delegation or family table changes. Describing a dynamic model is more challenging, especially in ALLOY, because of the "Frame Problem" – a famous artificial intelligence problem regarding the issue of retaining state information state across transitions [20]. Because ALLOY assumes that anything can happen unless

constrained otherwise, it is necessary to write two constraints for each API command: a constraint expressing the state changes if the command was made and a second constraint expressing that the state does not change if the command was not made. For example, we describe the `TPM_UpdateVerificationCount` command with two predicates:

```
pred UpdateVcount(s,s': State, d:Del){
  cnt.s' = cnt.s ++ d -> (dels.s.d).vcnt.s
}

pred NoUpdateVcount(s,s': State){
  cnt.s' = cnt.s
}
```

Next, we also included the `Authorize` predicate that is true only when the user is authorized to perform a particular command in a particular state. In our model, the user is not the TPM owner and therefore must receive delegations to perform owner-authorized tasks. Next, we had to organize these constraints across states by constraining that between any two states, either the user is authorized and the command succeeds or nothing happens:

```
pred Trace()  {
  // for all states
  all s: State - SOrd/last() | let s' = SOrd/next(s) |
  {
    // for some delegation family that hasn't been created
    // either the user is authorized and it is created
    // or it isn't created
    some f:vcnt.s'.Count |
    Authorize(s,Manage) and CreateFRow(s,s',f) or NoCreateFRow(s,s')

    some f: vcnt.s'.Count |
```

```
    Authorize(s,Manage) and IncFamily(s,s',f) or NoIncFamily(s,s')

    some d:task.s'.Cmd, c:Cmd, f:vcnt.s'.Count |
    Authorize(s,Create) and CreateDel(s,s',d,f,c) or NoCreateDel(s,s')

    some drow: DRow | some d:task.s.Cmd |
    Authorize(s,Load) and LoadDel(s,s',drow,d) or NoLoadDel(s,s')

    some d:task.s.Cmd |
    Authorize(s, Update) and UpdateVcount(s,s',d) or NoUpdateVcount(s,s')
  }
}
```

Although a cleaner model would enforce that only a single operation can occur between any two states, this model actually allows for any number of operations to occur simultaneously, as long as they do not act on the same objects. Finally, we performed an analysis on the delegation commands by writing a few assertions:

- a user without a certain privilege and without a delegation to that privilege can never exercise the privilege. This is the fundamental security policy of the delegation module.

- the verification count of any delegation blob is never higher than the value of its family's. If this were true, the delegation would remain valid in the future even though the family count was incremented to invalidate it, thereby representing a breach in temporal security.

ALLOY found these assertions to be valid under a scope of 6 atoms in 34 and 44 seconds respectively.

7.5 Authorization

The authorization model surrounding the use of TPM resources is quite straightforward. The first step to gaining access to any TPM resource (e.g. key, counter, non-volatile memory index) is to create either an *Object Independent Access Protocol* (OIAP), *Object Specific Access Protocol* (OSAP) or *Delegation Specific Access Protocol* (DSAP) transport session to the entity. The transport sessions manage all commands to the resource by creating a storage area for authorization information, creating an authorization handle and providing protective nonces to guard against replay attacks. On top of the transport session, access to any resource is further guarded by the resource's 160-bit usage password.

In this section, we focus on modeling the underlying mechanism of password authorization for the two object-specific transport protocols OSAP and DSAP. By object specific, we mean that these transport sessions are opened for a particular resource and not for a generic resource, which is the case for OIAP. When a OSAP transport session is created, the TPM creates a *session-store* and records the *handle* to that store. It then calculates the shared-secret based on the resource's usage password and records it in the session-store. Subsequently, when a command is made through this session, the TPM retrieves the shared-secret from the session-store and verifies that it was correctly generated from the target resource's usage password. If that verifies, the shared-secret is used to calculate the HMAC of the input parameters. The resulting digest is checked against the input-parameter digest provided by the user. If the digests match, the command proceeds. This process is repeated every time a command is issued.

On the other hand, when a DSAP transport session is established, the TPM first checks the delegation for validity before creating the session-store and generating the shared-secret from the delegated password. Subsequently, when a command is made through this DSAP session, the shared-secret is retrieved, checked to be generated from the delegated password and then used to HMAC the input parameters. If the resulting digest matches the input-parameter digest provided by the user, the

command proceeds. Similarly, this process is repeated every time a command is issued.

Model and Results

We used ALLOY to model the authorization procedure since we needed a tool to automatically reason about an arbitrary number of sessions and resources. In OTTER, these objects would have had to be hard-coded into the model.

We defined a set of `Resource` objects, a set of `Del` objects representing key delegations and a set of `Session` objects representing transport sessions. Each `Session` object points to a target resource and contains a shared-secret of type `Value`. The `Value` atoms are split into two subsets: `Known` atoms representing known values and `Secret` atoms representing unknown secrets.

```
// abstract means that Value objects do not actually exist
abstract sig Value {}

sig Known, Secret extends Value {}

sig Resource {
  password: one Value
}

one sig Del {
  // each delegation has exactly one (resource, value) pair,
  // where Value is the delegated password
  map: one (Resource -> Value)
}
```

We also defined a set of `Command` objects consisting of a transport session, a password and a target resource. The commands would succeed only if the transport session logic successfully verified the command input-parameter digest:

```
  sig Command {
    session: one Session,
    password: one Known,
    target: one Resource
}
```

To analyze the security of the OSAP and DSAP transport session logic, we asserted that the user knows the target resource's usage password for all OSAP authorized commands that succeeded. We did the same for the DSAP transport sessions:

```
  assert DSAPisSafe {
    // for all commands, either the user knows the password
    // or he received a delegation for it
    all c:Command |
    c.target.password in Known || c.target in Del.map.Value
  }
```

Our model found that the DSAP transport session was susceptible to a key-handle switching attack by producing a counterexample within a scope of 10 atoms in 18 seconds (the full model can be found in Appendix B.1). In ALLOY's counterexample, the attacker was able to successfully issue a command using a resource for which he did not know the usage password and did not receive a delegation for.

The DSAP session command does not check that the target resource in the input parameters matches the resource specified in delegation because the usage password of the target resource was not used to generate the session shared-secret. Rather, it is the delegation password that was used. Therefore, it is possible for an adversary to specify a completely arbitrary resource handle while still providing the delegated password. The command would still validate and the user would be given usage access to the resource. This is illustrated in Figure 7-2.

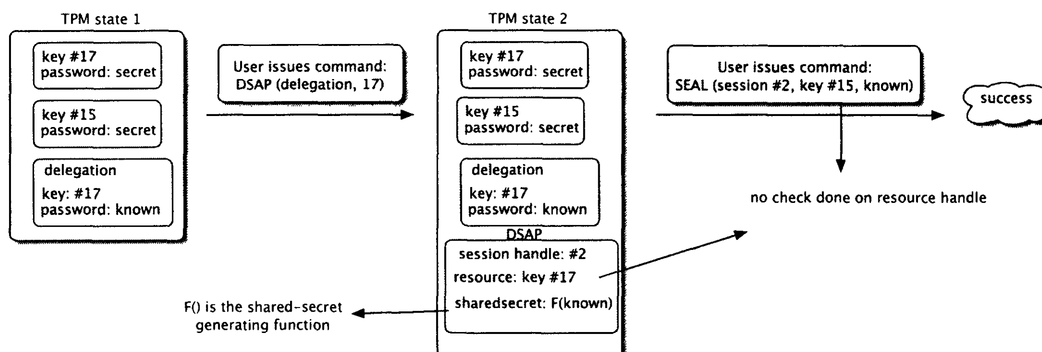


Figure 7-2: Key-handle switching attack

7.6 Model Integration

After analyzing each section of the API individually, we decided to assemble the pieces incrementally to build a model of the full API. Our main vehicle for this task was OTTER since our main goal was to derive serious API-chaining attacks. For those API commands that were modeled in OTTER, we merged them into a single file. For those that were modeled in ALLOY, we took the results from their analyses and incorporated them into the existing OTTER file. We first started by assembling the key management and protected storage modules since both dealt with key material and secret data. The model consisted of 36 implication rules including the ones describing the attacker's primitive abilities of encryption, decryption, key generation and hashing. However, it proved too large for OTTER to handle and we were unfortunately unable to proceed with integrating the other modules, which we leave for future work. In this section, we will describe the issues faced and some possible solutions.

We chose to run our search in breadth-first mode by forcing OTTER's given-clause algorithm to select clauses from the sos list based on the order they were inserted. With this model, OTTER's search-space began to explode by the third level. At this level, OTTER was simply generating too many clauses that were being forward subsumed by already existing clauses, leading to an extreme slow-down in productivity. This integrated model can be found in Appendix B.2 and Table 7.5

shows the statistics of OTTER's analysis of the model.

clauses given	3675
clauses generated	354310
clauses kept	6034
clauses forward subsumed	348370
clauses back subsumed	6
user CPU time	657.85
system CPU time	4.16
wall-clock time	698

Table 7.5: OTTER statistics for the integrated model

There are a number of possible approaches to this problem but all require some kind of trade-off:

- We could try to reduce the number of kept clauses by introducing weighting and change OTTER's given-clause algorithm to pick clauses based on their weight instead of their order in sos. Then, by defining a `max_weight` constant and putting extremely heavy weights on certain clauses, we can instruct OTTER to discard those clauses. The clauses we would want to discard are those that would seem meaningless or nonsensical within the system being considered. For example, secrets that have undergone too many layers of encryption or hashing would have lost their semantic value, so they become worthless. To model this, we use OTTER's *weighting templates* to define the weights of clauses. For example, to discard blobs that contain hashes of hashes of hashes, we can use the `$dots` function. If `$dots(t)` occurs in a weight template, it matches any term that contains `t`:

```
assign(max_weight,1000).
weight_list(pick_and_purge).
% let hashes of hashes of hashes have a weight higher than max_weight
weight(SHA($dots(SHA($dots(SHA(t))))), 1001)
end_of_list.
```

Although this may help in reducing the number of kept-clauses, it seems that the real problem is that OTTER is consuming time generating clauses that are only being subsumed.

- To control the generation of clauses, we could further partition the model's data by creating new predicates for each kind of data blob. This would control the search-space explosion by reducing the branching factor of the search. At the same time, however, it would limit the attacker's ability in trying to provide illegal input to a command. In fact, this approach was tried and did not greatly improve the situation.

- The most ideal way of controlling this blow-up is to remove the branching factor at the source. OTTER was producing so many similar clauses that were being subsumed because there were many paths to the same conclusion, indicating that many of our implication rules were giving similar results. We should remove these "redundant" implication rules from the model. These decisions, however, represent an extremely difficult problem that we leave for future work.

Chapter 8

Insecure TPM Implementations

The level of detail, clarity and coherence of the TPM v1.2 specification leaves many details open to individual interpretation. In this chapter, we intend to highlight the vulnerabilities that could arise from real TPM implementations as a result of following the TPM specifications. We consider three types of implementations of the TPM specification – Insecure_Implementation_X, Y and Z – each with a successively increasing amount of variance from the specified design and highlight the vulnerabilities that could result. In particular, the complete break in security as a result of a long chain of commands in Insecure_Implementation_Z gives a flavour of the attacks we hope to find with our techniques and tools.

8.1 Insecure_Implementation_X

Insecure_Implementation_X represents an implementation where ambiguities are resolved blindly. This could very well happen if an entry-level developer without much security engineering experience was asked to implement the API simply based on the original TPM specification. To see an example of the possible effects, we consider the `TPM_Delegate_UpdateVerification` command.

This command sets the verification count of a delegation blob to that of the family's, thereby validating the delegation. The instructions for this command, as taken from pg 147 and 148 of the *TPM Main Part3: Commands* [1] specification, are:

```
1 Verify the TPM Owner authorizes the command and parameters, on
   error return TPM_AUTHFAIL
2 Determine the type of input data (TPM_DELEGATE_ROW or
   TPM_DELEGATE_BLOB or TPM_TAG_DELB_OWNER_BLOB)
   and map D1 to that structure
3 Locate (D1 -> pub -> familyID) in the TPM_FAMILY_TABLE and set
   familyRow to indicate row, return TPM_BADINDEX if not found
4 Set FR to TPM_FAMILY_TABLE.FamTableRow[familyRow]
5 If delegated, verify that the current delegation family
   (FR -> familyID) == (D1 -> pub -> familyID); otherwise
   return error TPM_DELEGATE_FAMILY
6 If delegated and (FR -> flags TPM_FAMFLAG_ENABLED) is FALSE,
    return TPM_DISABLED_CMD
7 Set D1 -> verificationCount to FR -> verificationCount
8 If D1 is a blob, recreate the blob and return it.
```

The misleading step is step 8; it says to recreate the blob and return it. This command, if implemented incorrectly, will give the attacker basically a way of creating arbitrary valid delegations. A valid delegation blob is one that has a correct HMAC digest keyed with the special value *tpmProof*, a secret known only to the TPM. Therefore, an adversary cannot normally create rogue delegation blobs without knowing this secret. However, the above command does not check the validity of the input delegation blob before recreating the blob with the new verification count in step 8. An invalid delegation blob can be passed in and the command will return an updated and valid version. The fact that this command is owner-authorized (in step 1) makes the attack a little weaker since the owner is allowed to create arbitrary delegations anyway. However, the owner is not allowed to create arbitrary key delegations which he can do so by using this command.

8.2 Insecure_Implementation_Y

Next, we move on to Insecure_Implementation_Y, an implementation where arbitrary design decisions are overridden by the developer for a reasonable purpose (e.g. saving programming and testing resources). Let us consider the `TPM_CMK_CreateTicket` command that is used in certifiable-migration by migration authorities to certify the intended migration key (destination key). Recall that certifiable-migration is migration only to parties either specified by the key owner or the migration authority (MA). The command creates a special *sigTicket* structure:

```
SHA1(tpmProof || SHA1(verification public key) || signed data)
```

Firstly, this data structure is not tagged with any meta-data describing what kind of data it contains. Secondly, the placement of all three items in the hash seems completely arbitrary. By switching the order of the last two items:

```
SHA1(tpmProof || signed data || SHA1(verification public key))
```

we were able to perform a blob-confusion attack on the the `TPM_CMK_CreateKey` command, which requires a hash the form:

```
SHA1(tpmProof || SHA1(MA public key) || SHA1(this public key) )
```

as the migration password of the certifiable-migration key (CMK). Through a series of API commands involving `TPM_Sign`, `TPM_AuthorizeMigrationKey`, `TPM_CMK_CreateKey` and this altered version of `TPM_CreateTicket`, we were able to migrate an MA-controlled certifiable-migration key to an arbitrary key other than the one specified by the MA. Although this attack was found through inspection, it was quickly verified by OTTER and the model can be found in Appendix B.2.

8.3 Insecure_Implementation_Z

Finally, we move to Insecure_Implementation_Z, which is an implementation that strays from the given specification in a single serious way. In this section, we intend

to demonstrate how a single flaw in a fragile design can lead to a complete break in security. In the TPM v1.2 specifications, `TPM_Load` checks that the migration password is *tpmProof* (the TPM master secret) for any non-migratable key before loading the key into memory. We consider a different version, `TPM_Load2`, that leaves out this check for *tpmProof*, allowing us to create a series of very serious attacks.

Injection Attack

The first in this series of attacks is the *injection attack*. This attack relies on the failure of the conventional migration process to provide two-way authentication. An honest source TPM can ensure that keys are migrated to only honest destination TPMs but an honest destination TPM has no way of verifying that an incoming key originated from an honest TPM. Therefore, the obvious attack would be to spoof an incoming migration-blob by doing the following:

1. use `TPM_CreateWrapKey` to create a key, *K1*

2. load the key into the TPM with `TPM_LoadKey`

3. generate an asymmetric key, *K2*, manually and create a migration-blob structure containing *K2* with a non-migratable flag. This can be done by encryption $K2_{priv}$ with $K1_{pub}$ and then OAEP encoding the result as specified in `TPM_CreateMigrationBlob`

4. load the above migration-blob into the victim TPM using `TPM_ConvertMigrationBlob`, which does not perform a check on the key's migratability flag and subsequently `TPM_LoadKey2`, which does not check for the correct migration password given a non-migratable key.

This attack gives the adversary knowledge of the private half of a non-migratable key that is loaded in the TPM.

Discovering *tpmProof*

The next attack aims to discover the TPM master secret *tpmProof* by leveraging on the attacker's previous injection attack. Every non-migratable key created by the TPM has its migration password set to the master secret *tpmProof*. Non-migratability is ensured since no user can ever provide this secret value. However, since we now know the private half of a non-migratable key loaded in the TPM, we can do the following:

1. create a non-migratable key (*K1*) with TPM_CreateWrapKey and designate K2 as the parent key. The command returns an asymmetric key blob encrypted under $K2_{pub}$ with *tpmProof* as the migration password.

2. decrypt the key blob using $K2_{priv}$ to obtain *tpmProof*

Knowledge of this master secret is disastrous since it is used (indirectly or directly) in almost all the important commands in the TPM. As an example, we show how to extract any non-migratable key from the TPM once we have knowledge of *tpmProof*.

Extracting any key

This attack exploits the vulnerability that although there are three measures to protect against illegal migration of non-migratable key, only two are enforced and both are bootstrapped to a single security assumption that *tpmProof* is kept secret. The three measures are:

1. migration-key structures are authenticated since they contain a hashed *tpmProof*.

2. every TPM key contains a migratability flag.

3. every TPM key has the migrate password bound to the private key, which is checked against the user provided password

During the migration process, (2) is never checked and (1) and (3) rely on *tpmProof* being secret. Since our previous attack allows an adversary to discover *tpmProof*, all non-migratable keys can now be extracted by doing the following:

1. Construct the following blob:

$$[K_{PUB}, HMAC(K_{PUB}, tpmProof)]$$

   authorizing K to be a migration key. Ordinarily, this can only be done through `TPM_AuthorizeMigrationKey` since only the TPM knows $tpmProof$. However, the knowledge of tpmProof allows the adversary to make <u>ANY</u> public key a migration key.

2. Export any non-migratable key K' under K by providing the correct migration password *tpmProof*

3. Decrypt the migration-blob obtained from (2) to obtain the private key value of K'.

This allows an adversary to extract keys generated by other applications sharing the TPM.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we have shown the continued application of formal methods and automated reasoning tools to the analysis of security APIs, in particular, the Trusted Platform Module. Using OTTER and ALLOY, we were able to capture many types of API attacks, including a genre of attacks we had never considered before – partial attacks. Also, we have managed to demonstrate techniques for dealing with system state and API complexity. Nevertheless, although we have understood various APIs and discovered attacks, it is interesting to note that the practice of modeling an API often produces results faster than the automated analysis itself, as was the case for the attacks described in this thesis.

9.1.1 Tool Comparison

OTTER's resolution-style theorem-proving is excellent for API-chaining attacks and blob-confusion attacks. Also, the ability to perform optimizations within the given-clause algorithm to guide the search is extremely useful. Furthermore, if the system state could be captured within a single predicate, the traditional "frame-problem" becomes less of a hassle since a single predicate represents a snapshot of the entire system. Lastly, the ability of OTTER to handle user-interaction during the proof-

search is an enticing bonus that may come in helpful. On the other hand, OTTER's main disadvantage is at automatically reasoning about an arbitrary number of objects. All initial information has to be present in the system before OTTER begins reasoning and it is unable to increase, for instance, the number of resources, users and machines in the model during its analysis.

ALLOY is much better at reasoning about arbitrary sets of objects. Given a bound, ALLOY can search all models corresponding to all combinations of users, machines and resources. It is extremely powerful at reasoning about properties and data structures since these can often be precisely and concisely expressed in ALLOY's set, logical and relational notations. We have seen great success at using ALLOY to model previously found information leakage attacks and guessing attacks – notions that are less natural to OTTER. One avenue that has not been explored is ALLOY's power in generating lower-level API security policies based on higher-level ones. Despite these strengths, ALLOY, being constraint-based, is unable to deal with too much detail when it comes to API modeling. An extra detail locally can result in an extra dimension to every constraint globally, resulting in an extremely heavyweight model. Nevertheless, both OTTER and ALLOY are excellent tools and complement each other when it comes to API modeling.

9.1.2 TPM security

The TPM was designed to be the cornerstone for all future "trusted" applications. Although we were unable to find any long sequences of commands yielding attacks on the original specification (i.e. attacks unlikely to be found by hand), we have demonstrated serious one or two step vulnerabilities allowing the attacker to create arbitrary valid delegations and use arbitrary resources. Furthermore, we have shown design weaknesses in the API that compromise its robustness, such as the over-reliance on a single master secret *tpmProof*. These design weaknesses can result in serious vulnerability in a poor implementation of the TPM's ambiguous specification, which is highly likely considering the number of TPM chip manufacturers. It is therefore our recommendation that the TPM specification, as well as other security product

design specifications, come with security policies at the feature and API levels. This way, security engineers can quickly analyze and reason about the design at the feature level while software developers can do the same as they implement the API.

9.2 Future Work

With this work, we have further developed the group's tools, techniques and experience in dealing with security APIs. This leaves us with a number of future directions:

- **Key Typing Language** – we are currently working on trying to classify key-typing meta-data into multiple subcategories like *form, role* and *control.* Form describes the mathematical properties of the key, role describes the intended use of the key and control describes the ownership and responsibility of the key. By creating a language for defining keys, security API designers can better communicate their intentions about various keys and engineers can better analyze their systems with these concepts in mind. Furthermore, it is possible that analyzing an API while only considering certain aspects of key-typing information may help to manage complexity while still yielding results.

- **Europay-Mastercard-Visa** – EMVCo is an organization started in Europe tasked with replacing all credit and debit cards with new chip-and-pin smart-cards. With these more intelligent devices, ATMs and point-of-sale machines require extra API commands to make banking transactions. This would be an extremely rich area for future API research.

- **Modeling Partial Attacks** – after witnessing ALLOY's strength in reasoning about information leakage and guessing attacks, there is great interest in modeling partial attacks that do not compromise the security policy fully but compromise them sufficiently enough to allow brute-force attacks. We foresee this to be a difficult but important area of research as the personal computer becomes more and more powerful.

It is our hope that this research will spark off more academic attention in the area of security API analysis and help achieve higher industry security standards.

Appendix A

Partial Attacks

This section contains the ALLOY model verifying the cryptanalytic attack that Biham [5] found on the ECB|ECB|OFB triple-mode block-cipher, described in section 5.2. Biham's attack allowed the attacker to discover the key used in the OFB layer. It is important to note that this model only describes the end goal of the attack – finding a series of ciphertexts representing encryptions under only the OFB key. Since ALLOY is non-deterministic, it may find different attacks (that yield the same result) other than the one described by Biham when run on different platforms.

A.1 Triple-mode DES

```
/* This Alloy model captures the cryptanalytic attack on the
  ECB|ECB|OFB triple-mode block cipher. In this model, we combine
  the first two ECBs to form a single ECB operation
*/
module ECBOFB
open util/ordering[Block]

sig Value {
  xor : Value one -> one Value
}
```

```
fact XORfacts {
  // associativity
  all a, b, c : Value | xor.c.(xor.b.a) = xor.(xor.c.b).a

  // commutativity
  all b : Value | xor.b = ~(xor.b)

  // identity
  one identity : Value | no identity.xor - iden
}

sig Key{
  enc: Value one -> one Value
}
fact EncFacts {
  all disj k,k':Key | some (k.enc - k'.enc)
}

one sig OFB, K1 extends Key {}

sig Block {
  p: one Value,
  ofb: one Value,
  c: one Value
}

fact BlockFacts {
  //the ciphertext is a function of the plaintext and the ofb value
  all b: Block | b.c = (K1.enc[b.p]).xor[b.ofb]
```

```
  // setting up the OFB sequence through the blocks
  all b: Block-last() | let b' = next(b) |
  b'.ofb = OFB.enc[b.ofb]
}

pred size() {
  #Value > 5
  some xor
}

pred FindAttack(){
  size()

  // the set contains the relevant properties that two sequential XORs
  // of outputs match two sequential XORs of OFB encryptions
  all b':Block-first()-last() | let b=prev(b'), b''=next(b') |
  some d:OFB.enc.Value | let d'=OFB.enc[d], d''=OFB.enc[d'] |
  b.c.xor[b'.c] = d.xor[d'] && b'.c.xor[b''.c] = d'.xor[d'']
}

run size for 8 but exactly 2 Key
run FindAttack for 8 but exactly 2 Key
```

On a 1 GHz PowerPC G4 Apple powerbook with 768 MB of RAM, "`run size for 8 but exactly 2 Key`" completed in 1 minute 10 seconds and "`run FindAttack for 8 but exactly 2 Key`" completed in 2 minutes 25 seconds yielding an instance containing the attack that Biham had described.

Appendix B

TPM Models

B.1 Authorization

This section contains the ALLOY model of the authorization framework used in the TPM. In this model, all three transport sessions (OIAP, OSAP and DSAP) are modeled, together with a set of arbitrary resources and values. This model is able to find an instance where the attacker is able to issue a command for a particular resource without knowing the resource usage password or receiving a delegation for the resource.

```
/* This Alloy model demonstrates the key-handle
switching attack on the TPM's DSAP transport session
*/
module authorization

abstract sig Value {}
sig Known, Secret extends Value {}

// a set of TPM resources (keys, memory locations, etc)
sig Resource {
  password: one Value
```

```
}
fact ResourceFacts{
  // we want a model where no two resources have the same password
  all disj r,r': Resource | r.password != r'.password
}

// a set of delegations
one sig Del {
  map: one (Resource -> Value)
}
fact DelFacts{
  // all delegations delegate the use of a resource for which
  // the password is Secret and the new password is Known
  all d:Del | let delegated = d.map.Value, newpass = d.map[Resource] |
    delegated.password not in Known && newpass in Known
}

// a set of transport sessions
abstract sig Session {
  resource: one Resource,
  sharedsecret: one Value
}
sig OIAP, OSAP extends Session {}
sig DSAP extends Session {
del: one Del
}
fact Sessionfacts{
  // OIAP sessions do not have resources
  all s:OIAP | no (s.resource)
```

```
  // for all OSAP session, the sharedsecret is derived from the resource
  all s:OSAP | s.sharedsecret = s.resource.password

  // we want a model where each sessions is opened for a different resource
  all disj s,s': Session | s.resource != s'.resource

  // all DSAP sessions get their resource and new password
  // from the delegation-blob
  all s:DSAP | s.resource = s.del.map.Value
    && s.sharedsecret = s.del.map[Resource]
}

// when a command atom exists, that means it was authorized
// succesfully
sig Command {
  session: one Session,
  password: one Known,
  target: one Resource
}
fact CommandFacts{
  // for OIAP sessions, the password given during command
  // must match the password of the resource
  all c:Command |
  c.session in OIAP => c.password = c.target.password

  // for all OSAP authorized commands, the TPM checks the target
  // resource and the password provided to make sure they match
  all c: Command |
    c.session in OSAP => c.password = c.session.sharedsecret &&
    c.target = c.session.resource
```

```
  // for all DSAP authorized commands, the TPM checks only the
  // password, not the target resource
  all c:Command |
    c.session in DSAP => c.password = c.session.sharedsecret
}

pred size(){}

// Does OSAP allow unauthorized use of resources?
assert OSAPisSafe {
  // for all OSAP commands succesfully authorized, the user
  // knows the password for the target resource
  all c: Command | c.session in OSAP => c.target.password in Known
}

// Does DSAP allow unauthorized use of resources?
assert DSAPisSafe {
  // user cannot use any target for which the password is unknown
  // and he did not receive a delegation for it
  all c:Command |
    c.target.password in Known || c.target in Del.map.Value
}

run size for 10
check OSAPisSafe for 10
check DSAPisSafe for 10
```

On a 1 GHz PowerPC G4 Apple powerbook with 768 MB of RAM, "`run size for 10`" completed in 21 seconds, "`check OSAPisSafe for 10`" completed in 25 seconds and "`check DSAPIsSafe for 10`" completed in 18 seconds yielding a counterexam-

ple.

B.2 Key Management and Protected Storage

This section contains the OTTER model for the key management and protected storage commands in the TPM. The first section of the model is documentation describing the different predicates used and what they mean. The second section of the model lists the commands, the third section of the model sets up the environment by listing the initial knowledge and the fourth section lists the goals of the attacker. Furthermore, by turning on/off the relevant commands (which will be indicated in the code), this model will find the Insecure_Implementation_Y and Z attacks as described in chapter 8.

```
%-----------------------------------------------------------------
% TPM Specification Version 1.2
% this model captures the Key management and protected storage
% API commands and attacker abilities in an attempt to find an
% API-chaining attack on the system.
%----------------------------------------------------------------
set(process_input).
set(sos_queue).

clear(print_kept).
clear(print_back_demod).
clear(print_back_sub).

set(hyper_res).

assign(max_mem, 500000).
assign(max_seconds, 900 ).
assign(stats_level,1).
```

```
assign(max_proofs, 10).

%-----------------------------------------------------------------------
%                            Function Definitions
%-----------------------------------------------------------------------
% TPM(x) = TPM has loaded x

% U(x) = User knows x

% TPMKEY(x): the TPM generated this key x

% UKEY(x): user generated this key x

% KEY(x,xupass,xmpass,xtype,xmig) =  logical representation of a key
%  x = key identifier
%  xupass = usage password
%  xmpass = migration password
%  xtype = STORAGE, IDENTITY, etc
%  xmig = MIG (migratable), NMIG (non-migratable)

% priv(x) identifies the private half of key x
% pub(x) identifies the public half of key x

% ASYM(priv(x),xupass,xmpass,pub(y)) = user has this TPM_ASYM_KEY struct
%  priv(x) = private key of x
%  xupass = usage password
%  xmpass = migration password
%  pub(y) = encrypted under public half of key y
%  produced by: CreateKey, CMK_CreateKey, ConvertMigrationBlob
```

```
% E(x,y) = encryption of x under y

% MIGBLOB(priv(x),xupass,xmpass,pub(y)) = user has this OAEP encoded
%   TPM_MIGRATE_ASYMKEY struct
%  necessary to model separately because it has payload = TPM_PT_MIGRATE
%  priv(x) = private key
%  xupass = usage password
%  xmpass = migration password
%  pub(y) = encrypted under public key y
%  produced by: CreateMigrationBlob, CMK_CreateBlob

% keyInfo(x,y) = x, y are parameters describing the key to be modelled

% SHA(x) = SHA 1of x
% SHA(x1,x2,x3) = SHA1 of the concatenation of three components

% H(x,y) = hash done with a key, done with SHA1
%  x is the key type data
%‘y is the key flag data.

% SIG(x,y) = user has this signature of data x using private key y

% Sealed(tpmProof,xencpass,xdata,xpcr,pub(key))
% xencpass = usage password for this SEALED blob
% xdata = data that is encrypted
% xpcr = pcr values
% pub(key) = public key used to encrypt data

% Bound(xpayload,xdata,pub(key))
% xpayload = tag meta-data for this BOUND blob
```

```
% xdata = data that is encrypted
% pub(key) = public key used to encrypt data

% ----- START OF USABLE --------
list(usable).

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

%**********************************************************************
% Command: Create restrictTickets
% may need to constrain this a bit more if Otter blows up
-U(SHA(x)) | -U(SHA(y)) | -U(SHA(z)) | U(RTICKET(SHA(x),SHA(y),SHA(z))).

%***********************************************************************
% Command: Generate Signatures, but only on RTICKETS to control blow-up
-U(RTICKET(w,x,y)) | -U(priv(z)) | SIG(RTICKET(w,x,y),priv(z)).

%**********************************************************************
% Command: forging CMK_CreateKey ASYM structures
 -U(priv(w)) | -U(SHA(y)) | -U(pub(z)) |
 ASYM(priv(w),PASS,SHA(y),pub(z)).

  -U(priv(w)) | -U(tpmProof) | -U(pub(z)) |
 ASYM(priv(w),PASS,tpmProof,pub(z)).

 -U(priv(w)) | -U(pub(z)) |
 ASYM(priv(w),PASS,MPASS,pub(z)).
```

```
%***********************************************************************
% Command: forging MIGBLOB structures
 -U(priv(w)) | -U(SHA(y)) | -U(pub(z)) |
 MIGBLOB(priv(w),PASS,SHA(y),pub(z)).

 -U(priv(w)) | -U(pub(z)) |
 MIGBLOB(priv(w),PASS,MPASS,pub(z)).

%**********************************************************************
% Command: Decrypt Blobs
-MIGBLOB(w,x,y,pub(z)) | -U(priv(z)) | U(dec(MIGBLOB(w,x,y,pub(z)))).
-U(dec(MIGBLOB(w,x,y,pub(z)))) | U(w).
-U(dec(MIGBLOB(w,x,y,pub(z)))) | U(x).
-U(dec(MIGBLOB(w,x,y,pub(z)))) | U(y).

-ASYM(w,x,y,pub(z)) | -U(priv(z)) | U(dec(ASYM(w,x,y,pub(z)))).
-U(dec(ASYM(w,x,y,pub(z)))) | U(w).
-U(dec(ASYM(w,x,y,pub(z)))) | U(x).
-U(dec(ASYM(w,x,y,pub(z)))) | U(y).

-SEALED(v,w,x,y,pub(z)) | -U(priv(z)) | U(dec(SEALED(v,w,x,y,pub(z)))).
-U(dec(SEALED(v,w,x,y,pub(z)))) | U(v).
-U(dec(SEALED(v,w,x,y,pub(z)))) | U(w).
-U(dec(SEALED(v,w,x,y,pub(z)))) | U(x).
-U(dec(SEALED(v,w,x,y,pub(z)))) | U(y).


%******************************************************************
% Command: Implicit Knowledge of Blobs
-MIGBLOB(w,x,y,z) | U(MIGBLOB(w,x,y,z)).
```

```
-ASYM(w,x,y,z) | U(ASYM(w,x,y,z)).
-SEALED(v,w,x,y,z) | U(SEALED(v,w,x,y,z)).
-SIG(x,y) | U(SIG(x,y)).


%******************************************************************
% Command: SHA1
%  limit SHA-ing to: public keys
-U(pub(x)) | U(SHA(pub(x))).
%-U(RTICKET(x,y,z)) | U(SHA(RTICKET(x,y,z))). disallowed to prevent blowup


%-----------------------------------------------------------
%                          TPM Transaction Set
% All commands here are separated by a line of **********
%-----------------------------------------------------------

%********************************************************************
% Command: TPM_CreateWrapKey (non-migratable key)
-CreateKey |

% key generation parameters, specify that key is generated by the TPM
-KEY(x,xupass,xmpass,xtype,NMIG) | -TPMKEY(x) |

% parent key of correct type, flag and user presents correct password
-KEY(y,yupass,ympass,STORAGE,NMIG) | -U(yupass) |

% TPM has loaded the parent key y
-TPM(y) |
```

```
% returns TPM_STORE_ASYMKEY blob with tpmProof as migration password
ASYM(priv(x),xupass,tpmProof,pub(y)) .

%---------------------------------------------------------------
% Command: TPM_CreateWrapKey (migratable key)
-CreateKey |

% key generation parameters, specify that key is generated by the TPM
-KEY(x,xupass,xmpass,xtype,MIG) | -TPMKEY(x) |

% user presents usage password
-KEY(y,yupass,ympass,STORAGE,ymig) | -U(yupass) |

% TPM has loaded the parent key y
-TPM(y) |

% returns TPM_STORE_ASYMKEY blob
ASYM(priv(x),xupass,xmpass,pub(y)).


%*******************************************************************
% Command: TPM_LoadKey (non-migratable key)
-LoadKey |

% if key is non-migratable, it must have tpmProof
-KEY(x,xupass,tpmProof,xtype,NMIG) |

% providing blob and TPM has loaded parent key
-ASYM(priv(x),xupass,tpmProof,pub(y)) | -TPM(y) |
```

```
% user provides the correct auth
-KEY(y,yupass,ympass,STORAGE,NMIG) | -U(yupass) |

% TPM loads the key
TPM(x).

%-------------------------------------------------------------
% Command: TPM_LoadKey (migratable key)
-LoadKey |

% if key is non-migratable, it must have tpmProof
-KEY(x,xupass,xmpass,xtype,MIG) |

% providing blob and TPM has loaded parent key
-ASYM(priv(x),xupass,xmpass,pub(y)) | -TPM(y) |

% parent key is of proper type,flag and user provides correct password
-KEY(y,yupass,ympass,STORAGE,ymig) | -U(yupass) |

% TPM loads the key
TPM(x).

%**********************************************************************
% Command: TPM_LoadKey2 flawed implementation
-LoadKey2 |

% tpmProof is not need even if key is non-migratable (FLAW)
-KEY(x,xupass,xmpass,xtype,NMIG) |

% providing blob and TPM has loaded parent key
```

```
-ASYM(priv(x),xupass,xmpass,pub(y)) | -TPM(y) |

% user provides the correct auth
-KEY(y,yupass,ympass,STORAGE,NMIG) | -U(yupass) |

% TPM loads the key
TPM(x).

%******************************************************************
% Command: TPM_Seal
-Seal |

% sealing key is a non-migratable storage key
-KEY(x,xupass,xmpass,xtype,xmig) | -U(xupass) |

% user provides an encrpytion auth value and PCR values
-TPM(x) | -U(xencpass) | -U(xdata) | -U(xpcr) |

% TPM seals the data
SEALED(tpmProof,xencpass,xdata,xpcr,pub(x)).


%***************************************************************
% Command: TPM_Unseal
-Unseal |

% unsealing key is a non-migratable storage key
-KEY(x,xupass,xmpass,STORAGE,NMIG) | -U(xupass) |

% user provides the sealed blob
```

```
-SEALED(tpmProof,xencpass,xdata,xpcr,pub(x)) | -U(xencpass) |

% returns the data
U(xdata).


%*************************************************************
% Command: TPM_Unbind
-Unbind |

% unbinding key must be of type BIND or LEGACY
-KEY(x,xupass,xmpass,BIND,xmig) | -U(xupass) |

% input is a TPM_BOUND_DATA structure, user gets decrypted value
-BOUND(PT_BIND,xdata,pub(x)) | U(xdata).

%*************************************************************
% Command: TPM_Unbind
-Unbind |

% unbinding key must be of type BIND or LEGACY
-KEY(x,xupass,xmpass,LEGACY,xmig) | -U(xupass) |

% input is a TPM_BOUND_DATA structure, user gets decrypted value
-BOUND(PT_BIND,xdata,pub(x)) | U(xdata).


%***************************************************************
% Command: TPM_Sign
-Sign |
```

```
% parent key is of proper type, flag and user provides correct password
-KEY(y,yupass,ympass,SIGN,ymig) | -U(yupass) |

% signs the blob
-U(SHA(x)) | SIG(SHA(x),priv(y)).


%********************************************************************
% Command: TPM_AuthorizeMigrationKey (MIGRATE)
% Migration Schemes = MIGRATE, REWRAP, RESTRICT_MIGRATE, APPROVE,
% APPROVE_DOUBLE
-AuthorizeMigrationKey |

% showing TPM owner password and present migration key
-U(TPMowner) | -Trust(x) |

% return a TPM_MIGRATIONKEYAUTH structure
U(SHA(pub(x),MIGRATE,tpmProof)).


%*****************************************************************
% Command: TPM_AuthorizeMigrationKey (other variants)
-U(TPMowner) | -Trust(x) | U(SHA(pub(x),REWRAP,tpmProof)).
-U(TPMowner) | -Trust(x) | U(SHA(pub(x),RESTRICT_MIGRATE,tpmProof)).
-U(TPMowner) | -Trust(x) | U(SHA(pub(x),APPROVE,tpmProof)).
-U(TPMowner) | -Trust(x) | U(SHA(pub(x),APPROVE_DOUBLE,tpmProof)).

%*******************************************************************
% Command: TPM_CreateMigrationBlob (MIGRATE)
```

```
-CreateMigrationBlob |

% provide TPM_MIGRATIONKEYAUTH containing migration key (z)
-U(SHA(pub(z),MIGRATE,tpmProof)) |

% present TPM_STORE_ASYMKEY containing key to be migrated (x)
-ASYM(priv(x),xupass,xmpass,pub(y)) | -U(xmpass) |

% no restrictions on parent key, flag and user presents correct password
-KEY(y,yupass,ympass,ytype,ymig) | -U(yupass) |

% returns the OAEP encoded TPM_MIGRATE_ASYMKEY structure
MIGBLOB(priv(x),xupass,xmpass,pub(z)).

%---------------------------------------------------------
% Command: TPM_CreateMigrationBlob (REWRAP)
-CreateMigrationBlob |

% provide TPM_MIGRATIONKEYAUTH containing migration key (z)
-U(SHA(pub(z),REWRAP,tpmProof)) |

% present TPM_STORE_ASYMKEY containing key to be migrated (x)
-ASYM(priv(x),xupass,xmpass,pub(y)) | -U(xmpass) |

% no restrictions no parent key, flag and user presents correct password
-KEY(y,yupass,ympass,ytype,ymig) | -U(yupass) |

% returns the TPM_STORE_ASYMKEY rewrapped under migration key z
ASYM(priv(x),xupass,xmpass,pub(z)).
```

```
%*****************************************************************
% Command: TPM_ConvertMigrationBlob (MIGRATE only)
-ConvertMigrationBlob |

% present the TPM_MIGRATE_ASYMKEY structure
-MIGBLOB(priv(x),xupass,xmpass,pub(z)) |

% parent key must have type STORAGE, user presents correct password
-KEY(z,zupass,zmpass,STORAGE,zmig) | -U(zupass) |

% converts TPM_MIGRATE_ASYMKEY to TPM_STORE_ASYM and returns it
ASYM(priv(x),xupass,xmpass,pub(z)).


%*****************************************************************
% Command: TPM_CMK_CreateKey (part I)
-CMK_CreateKey |

% specify the Migration Selection Authority / Migration Authority (z)
-CMKEY(x,xupass,NULL,xtype,MIG) | -U(z) |

% parent key must have type STORAGE, user presents correct password
-KEY(y,yupass,tpmProof,STORAGE,NMIG) | -U(yupass) |

% returns a TPM_KEY12 with TPM_STORE_ASYM inside
ASYM(priv(x),xupass,SHA(tpmProof,z,SHA(pub(x))), pub(y)).

%-----------------------------------------------------
% Command: TPM_CMK_CreateKey (part II)
-CMK_CreateKey |
```

```
% specify the Migration Selection Authority / Migration Authority (z)
-CMKEY(x,xupass,NULL,xtype,MIG) | -U(z) |

% parent key must have type STORAGE, user presents correct password
-KEY(y,yupass,tpmProof,STORAGE,NMIG) | -U(yupass) |

% must also change the migration password for key x
KEY(x,xupass,SHA(tpmProof,z,SHA(pub(x))),xtype,MIG).

%*************************************************************
% Command: TPM_CMK_CreateTicket
-CMK_CreateTicket |

% user presents signed data (x), signature, verification key (y)
-U(x) | -SIG(x,priv(y)) | -U(pub(y)) |

% returns the sigTicket
U(SHA(tpmProof,SHA(pub(y)),x)).

%*********************************************************
% Command: TPM_CMK_CreateTicket2 (flawed implementation)
% same as CMK_CreateTicket except order of components is switched
-CMK_CreateTicket2 | -U(x) | -SIG(x,priv(y)) | -U(pub(y)) |
  U(SHA(tpmProof,x,SHA(pub(y)))).

%*********************************************************
% Command: TPM_CMK_CreateBlob (RESTRICT_MIGRATE)
%  also documented as migration directly to a MSA
-CMK_CreateBlob_Restrict_Migrate |
```

```
% output from TPM_AuthorizeMigrationKey, zmsa = MSA key = destination key
-U(SHA(zmsa,RESTRICT_MIGRATE,tpmProof)) |

% user inputs digest of the source key
-U(SHA(zsrc)) |

% TPM_KEY12 structure from CMK_CreateKey
-ASYM(priv(x),xupass,SHA(tpmProof,SHA(zmsa),SHA(zsrc)), pub(y)) |

% parent key must non-migratable, user presents correct password
-KEY(y,yupass,tpmProof,ytype,NMIG) | -U(yupass) |

% returns OAEP encoded TPM_MIGRATE_ASYMKEY structure
MIGBLOB(priv(x),xupass,SHA(tpmProof,SHA(zmsa),SHA(zsrc)),zmsa).

%-------------------------------------------------------
% Command: TPM_CMK_CreateBlob (APPROVE)
%  indirect migration that requires MA(zma) approval
-CMK_CreateBlob_Approve |

% proper TPM_MIGRATIONKEYAUTH from TPM_AuthorizeMigrationKey
-U(SHA(zdest,APPROVE,tpmProof)) |

% user-input restrictTicket, broken into three pieces
-U(SHA(zma)) | -U(SHA(zdest)) | -U(SHA(zsrc)) |

% TPM_KEY12 structure from CMK_CreateKey
-ASYM(priv(x),xupass,SHA(tpmProof,SHA(zma),SHA(zsrc)), pub(y)) |
```

```
% parent key must non-migratable, user presents correct password
-KEY(y,yupass,tpmProof,ytype,NMIG) | -U(yupass) |

% sigTicket from CMK_CreateTicket
-U(SHA(tpmProof,SHA(zma),RTICKET(SHA(zma),SHA(zdest),SHA(zsrc)))) |

% rewraps the TPM_KEY12 structure under zdest
ASYM(priv(x),xupass,SHA(tpmProof,SHA(zma),SHA(zsrc)), zdest).

%-----------------------------------------------------------
% Command: TPM_CMK_CreateBlob (APPROVE_DOUBLE)
%  requires MA(zma) approval
-CMK_CreateBlob_Approve_Double |

% proper TPM_MIGRATIONKEYAUTH from TPM_AuthorizeMigrationKey
-U(SHA(zdest,APPROVE_DOUBLE,tpmProof)) |

% user-input restrictTicket, broken into three pieces
-U(SHA(zma)) | -U(SHA(zdest)) | -U(SHA(zsrc)) |

% TPM_KEY12 structure from CMK_CreateKey
-ASYM(priv(x),xupass,SHA(tpmProof,SHA(zma),SHA(zsrc)), pub(y)) |

% parent key must non-migratable, user presents correct password
-KEY(y,yupass,tpmProof,ytype,NMIG) | -U(yupass) |

% sigTicket from CMK_CreateTicket
-U(SHA(tpmProof,SHA(zma),RTICKET(SHA(zma),SHA(zdest),SHA(zsrc)))) |

% does the OAEP encoding
```

```
MIGBLOB(priv(x),xupass,SHA(tpmProof,SHA(zma),SHA(zsrc)), zdest).

%-----------------------------------------------------------
% Command: TPM_CertifyKey
-CertifyKey |

% certifying key and authorization
-KEY(y,yupass,ympass,ytype,ymig) | -U(yupass) | -TPM(y) |

% key-to-be-certified and authorization
-KEY(x,xupass,xmpass,xtype,xmig) | -U(xupass) | -TPM(x) |

% creating the signature
SIG(SHA(pub(x)),priv(y)).

%------------------------------------------------------------
% Command: TPM_CertifyKey2
-CertifyKey2 |

% migrationPubDigest
-U(xmigpubdigest) |

% certifying key and authorization
-KEY(y,yupass,ympass,ytype,ymig) | -U(yupass) | -TPM(y) |

% key-to-be-certified and authorization
-KEY(x,xupass,xmpass,xtype,SHA(tpmProof,SHA(xmigpubdigest),SHA(pub(x)))) |
-U(xupass) | -TPM(x) |

% creating the signature
```

```
SIG(SHA(pub(x)), priv(y)).

end_of_list.
% --- END OF USABLE ---

% ---- START OF SOS _____
list(sos).

%------------------------------------------------------------
%        Command Enabling: put a "-" in front to turn off the command
% by default, we turn off the flawed commands
%------------------------------------------------------------
CreateKey.
LoadKey.
-LoadKey2.
Sign.
Seal.
Unseal.
Unbind.
AuthorizeMigrationKey.
CreateMigrationBlob.
ConvertMigrationBlob.
CMK_CreateKey.
CMK_CreateTicket.
-CMK_CreateTicket2.
CMK_CreateBlob_Restrict_Migrate.
CMK_CreateBlob_Approve.
CMK_CreateBlob_Approve_Double.
CertifyKey.
CertifyKey2.
```

```
%-------------------------------------------------------
%                             User Knowledge
%--------------------------------------------------------

% user generated key
UKEY(KEY1).
KEY(KEY1,PASS,MPASS,STORAGE,NMIG).
U(pub(KEY1)).
U(priv(KEY1)).

% user generated key
UKEY(KEY2).
KEY(KEY2,PASS,MPASS,STORAGE,MIG).
U(pub(KEY2)).
U(priv(KEY2)).

% TPM Storage Root Key
TPMKEY(SRK).
KEY(SRK,SRKPASS,tpmProof,STORAGE,NMIG).
TPM(SRK).
U(pub(SRK)).

% STORAGE1 = non-migratable storage
TPMKEY(STORAGE1).
KEY(STORAGE1,PASS,tpmProof,STORAGE,NMIG).
U(pub(STORAGE1)).

% STORAGE2 = migratable storage
 TPMKEY(STORAGE2).
```

```
 KEY(STORAGE2,PASS,MPASS,STORAGE,MIG).
U(pub(STORAGE2)).

% TPM generated Certifiable-Migration Key
CMKEY(CMK, PASS, NULL, SIGN, MIG).
U(pub(CMK)).

% One Migration Authority (MA) key:
% this predicate is commented out because we do not want the
% MA key to be generated. rather, it just exists
% KEY(MA,NULL,NULL,STORAGE,NMIG).
U(pub(MA)).

% One Migration Selection Authority (MA) or destination key
U(pub(TPM2)).

% MA certifies TPM2 with a signature
SIG(RTICKET(SHA(pub(MA)), SHA(pub(TPM2)), SHA(pub(CMK))), priv(MA)).

% User-known passwords
U(PASS).
U(MPASS).
U(SRKPASS).
% make user the TPM owner
U(TPMowner).

% keys that the TPMOwner trusts (for migration)
Trust(TPM2).
Trust(KEY1).
```

```
end_of_list.

% --- END OF SOS ----

%-----------------------------------------------------
%                          Goals
%-----------------------------------------------------

list(passive).

% Insecure_Implementation_Y
% Turn on CMK_CreateTicket2
% the CMK should only be migrated to TPM2 (as certified by the MA)
-MIGBLOB(priv(CMK),PASS,SHA(tpmProof,SHA(pub(MA)),SHA(pub(CMK))),pub(KEY1)).

% Insecure_Implementation_Z
% Turn on TPM_LoadKey2
% Loading a user-generated non-migratable key into the TPM
-TPM(KEY1).
% Cannot extract private halves of non-migratable keys
-U(priv(MA)).
-U(priv(STORAGE1)).
-U(priv(SRK)).

% Ultimate goal
-U(tpmProof).

end_of_list.
```

Bibliography

[1] Trusted Computing Group Architecture Overview Revision 1.2. http://www.trustedcomputinggroup.org.

[2] R.J. Anderson. Why cryptosystems fail. *Communications of the ACM*, November 1994.

[3] Mihir Bellare and Phil Rogaway. Optimal assymetric encryption. *Advances in Cryptology - Eurocrypt*, pages 92–111, 1994.

[4] Eli Biham. Cryptanalysis of multiple modes of operation. *Technical Report, Computer Science Department, Technion*, (CS0833), 1994.

[5] Eli Biham. Cryptanalysis of triple-modes of operation. *Technical Report, Computer Science Department, Technion*, (CS0885), 1996.

[6] M. Bond. Attacks on cryptoprocessor transaction sets. *Workshop on Cryptographic Hardware and Embedded Systems*, 2001.

[7] Jolyon Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, 2003.

[8] A.C. Yao D. Dolev. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29, 1983.

[9] Y. Novilov E. Goldberg. Berkmin: A fast and robust SAT-solver, 2002.

[10] Software Design Group. http://alloy.mit.edu/beta/, 2004.

[11] T. Hoare. Communicating sequential processes. *Communications of the ACM*, 1978.

[12] IBM Inc. *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide for the IBM 4758-001, Release 1.31*, 1999.

[13] Daniel Jackson. *Alloy 3.0 Reference Manual.* Software Design Group, May 2004.

[14] John Arnold Kalman. *Automated Reasoning with Otter.* Rinton-Press, 2001.

[15] Gavin Lowe. An attack on the Needham-Schroder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[16] P. Zielinski M. Bond. Decimalisation table attacks for PIN cracking. *Technical Report, University of Cambridge*, (560), 2003.

[17] R.J. Anderson M. Bond. API-level attacks on embedded systems. *IEEE Computer*, 34:67–75, 2001.

[18] R. Needham M. Burrows, M Abadi. A logic of authentication. *ACM Transactions in Computer Systems*, 8:18–36, February 1990.

[19] C. Meadows. A model of computation for the NRL protocol analyzer. *Proceedings of the Computer Security Foundations Workshop VII*, pages 84–89, 1994.

[20] Nilsson Nils Michael Genesereth. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, 1987.

[21] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[22] Sean Smith. *TrustedWorthy Computing and Applications.* Springer, 2005.

[23] M.E. Hellman W. Diffie. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, June 1977.

[24] Paul Youn. The analysis of cryptographic APIs using the theorem prover otter. Master's thesis, Massachusetts Institute of Technology, 2004.