Investigating formal representations of PIN block attacks

Eirini Kaldeli

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2007

Abstract

Financial security APIs control the use of tamper-proof hardware security modules (HSMs) that are used in cash machine networks. The idea is that the API keeps the system secure even from corrupt insiders. Recently, several attacks have been found on these APIs, attracting the attention of formal methods researchers to the area. One family of attacks involves cracking PIN values by tweaking inputs to API functions away from their usual values and watching for errors. These so-called "PIN block attacks" affect many APIs. A framework has been proposed for modelling them as Markov Decision Processes, and analysing the resulting models using probabilistic model checking, in order to asses how vulnerable an API configuration is. One problem of this framework is that the models produced are very large, and thus it often takes considerable time to analyse them.

The objective of this thesis is to investigate and implement alternative ways of representing the models of PIN block attacks, aiming at increasing their compactness, and consequently making their analysis more efficient in terms of time and memory requirements. The great amount of symmetry inherent in the model is one of the main characteristics that will draw our attention, since it is responsible for a lot of redundant operations. We experiment with our approaches on a number of different security API configurations, and evaluate the results. We argue that the efficiency of the probabilistic model checker depends on a number of issues, that should be taken into account during the modelling of real-world systems, in order to achieve faster performance and avoid memory overloads.

Acknowledgements

First, I would like to thank my supervisor Graham Steel for his guidance and advice at all stages of this project, and for being there whenever I needed him, patiently listening to my anxieties and queries. Thanks also to Alan Bundy for his thorough reviewing of this dissertation and his valuable comments.

I am grateful to Gethin Norman and Dave Parker for their quick responses to my questions about several practical and theoretical aspects of PRISM.

There are also a number of persons, who may not have been directly involved in this project, but without whom things would have been much harder. I will never forget the usual habitues of the fifth floor in Appleton Tower, too many to mention individually, the tough and good times we shared. Most special thanks go to Giorgos Korfiatis for his willingness to help at all times, and the endless conversations we had through the cables of VoIP. Finally, to my family, who has provided me with continual support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Eirini Kaldeli*)

Table of Contents

Chapter 1

Introduction

Automatic Teller Machines (ATMs) are used by millions of customers every day to make cash withdrawals, check their account balances, deposit cash or transfer money between bank accounts. The security of these transactions is primarily maintained by the internationally understood and accepted notion of the *Personal Identification Number* (PIN), which has become a common feature of everyday life over the past decades, bringing cryptography into wide use. The security of the PIN is a principal requirement in order to ensure customers and financial institutions against fraud, since its acquisition entails full control over the corresponding bank account.

In a typical ATM transaction, the PIN typed in by a customer needs to be decrypted, re-encrypted and verified several times, as it travels through the different zones of the ATM network. The security of these sensitive cryptographic operations relies mostly on the integrity of tamper-resistant electronic devices, known as Hardware Security Modules (HSMs), whose aim is to prevent the revelation of sensitive data, such as cryptographic keys and clear (unencrypted) PINs. HSMs are governed by *security APIs*, which keep the system secure even from corrupt insiders, by specifying a strict set of legitimate operations on the PIN. For a long time, it was believed that the HSM transactions supported by typical commercial APIs did not pose any particular threat against the security of the PIN. It was not until a few years ago, that numerous attacks against security APIs were discovered, which manage to crack the customers' PINs. In this thesis we are concerned with a particular class of these attacks, the so-called *PIN block attacks*, which exploit flaws in the PIN processing scheme of banking security APIs. The common premise of these attacks is to execute unanticipated sequences of commands to deceive the API into revealing secrets in a manner that was not intended by the API designer.

The discovery of attacks against security APIs has attracted the attention of

formal methods researchers to the area. The aim was to develop automated tools that would facilitate the particularly arduous process of API design, by assisting them at spotting weaknesses and flaws in the key management and PIN processing schemes, and potentially help them quantify and set bounds on information leakage through API commands. In [Ste06], Steel has proposed a framework for modelling PIN block attacks, and determining the expected number of steps required by an intruder who follows the most effective available strategy to recover the PIN. The modelling of PIN block attacks requires handling both probability and non-determinism, giving rise to a *Markov Decision Process* (MDP), that represents all possible attacks. The quantitative analysis required for the assessment of the cost of the optimal attack included in the MDP is performed by the probabilistic model checker PRISM.

Our work builds upon the existing framework implemented by Steel, called *AnaBlock*. One problem with this system is that the models it produces are very large, and thus it often takes too long a time for the model checker to analyse them. Our area of focus is to look over the model from different perspectives in a systematic way, and investigate alternative representations, aiming at improving the efficiency of AnaBlock in terms primarily of time and secondarily of memory consumption. Our goals include reducing the symmetry inherent in the model, which is responsible for a great amount of redundant operations and makes the analysis particularly inefficient.

The work undertaken for this thesis can be grouped into three alternative approaches. The first one attempts to reduce the number of variables contained in the model representation. The second proposes a split of the representation into interacting components (modules) and investigates alternative ways of reducing the symmetric nature of digitwise attacks, i.e. attacks that involve operations that are performed independently on each digit. Finally, the third approach aims at breaking the duplication that characterises a particular kind of PIN block attack, known as the full decimalisation attack.

The contribution of our work is two fold. First, we have managed to obtain considerable improvement in the runtimes required for the analysis of specific categories of configurations. For the case of configurations that allow full decimalisation table attacks, a much more compact model was constructed, by removing unnecessary operations and states. The results of our attempts to handle the digitwise class of attacks were rather ambivalent: although a dramatic drop in runtime was recorded, this only applies to configurations for which the attack model does not include other families of attacks. Secondly, our experience with PRISM has indicated some important issues, pertaining to the factors that affect

the efficiency of model construction and model checking. The observations we have made can potentially act as a guide to obtaining representations that are more efficient with respect to the probabilistic model checker.

1.1 Layout of the thesis

The contents of this thesis are structured in such a way to allow several levels of reading. The aim was to make the thesis as self-contained as possible, including all the background material that is necessary for a non-specialist reader to understand the approaches attempted and interpret the results obtained. The information contained in the background chapters is actually what the writer had to learn in order to undertake the work in this thesis. However, for readers familiarised with the concepts that concern this dissertation, either from the security or the automated-reasoning perspective, a great amount of this information will be unnecessary to read, in which case chapters 2-4 should be skipped, or read selectively.

The remainder of this thesis is outlined as follows. Chapter 2 introduces the concept of financial security APIs, and explains how they are used in order to ensure the confidentiality of the PIN. The idea behind the API attacks is explained, and a detailed description of the three families of PIN block attacks considered in this thesis are presented. Chapter 3 discusses the contribution of formal tools in the analysis of security APIs and gives a review of related work in this area, identifying what research has already been done, and how it is related to the system we investigate in this thesis. In chapter 4 we describe the main features of the probabilistic model checker PRISM we have used to analyse our models. This chapter includes a presentation of the kinds of probabilistic models, a description of PRISM's formalisms, and an introduction to the principal data structures on which PRISM's algorithms rely. The symmetry reduction tools incorporated in PRISM are also presented. Chapter 5 describes in detail the AnaBlock system our work rests on: its aims, how the three families of PIN block attacks are modelled, the algorithms for specifying the steps of each attack, and the properties that are model checked. The basic features of the range of API configurations on which the system is tested are presented, as well as the results of the experiments performed on these configurations. In the same chapter, we highlight the need for more efficient approaches to modelling the PIN block attacks and make some general remarks about the evaluation methods we will follow. Chapters 6 to 8 contain the main contribution of this thesis. In chapter 6, we discuss our attempt to represent the model of PIN block attacks by

using only two variables. We present the runtimes we obtained for a number of configurations, and try to give a concise interpretation of the poor performance of this representation, based on the way PRISM works. Chapter 7 suggests an alternative way to handling the symmetry characterising the digitwise attacks, by dividing the model into multiple modules and making use of a symmetry reduction tool. A range of experiments are performed, and an evaluation of the results follows. In chapter 8 we describe how the duplication resident in the full decimalisation table attack can be reduced, and present some experimental results. Finally, in chapter 9 we outline the achievements and limitations of our contributions, discussing the lessons learnt throughout our work. This chapter also includes some suggestions about possible directions for future research.

Chapter 2

Financial security APIs and their flaws

Cryptographic systems have long been used to store and process sensitive information, such as passwords, personal data, PINs or military secrets, in a manner designed to prevent unauthorised parties from gaining access to it. The confidentiality and integrity of this information, which is encrypted under cryptographic keys, must be preserved not only in storage, but also when it is being manipulated. For this purpose, a well-defined policy is needed to determine who has access to the data and under what circumstances, as well as to regulate its flow during processing. This policy is enforced by the *security API*, a cryptographic Application Programming Interface which specifies a set of commands that define the allowed interactions between the user and the security system. Any data released by the system is encrypted under one or more secret keys, so that it has no meaning in the outside world, and can only be manipulated by feeding it back into the system under the specifications of the desired API command.

2.1 Banking security APIs

The *Personal Identification Number* (PIN) constitutes the primary security measure for protecting customers and financial institutions from fraud during an ATM transaction. The PIN is a secret numeric password shared between the card holder and the issuing bank, and is used to establish authentication during a financial transaction. In the following, we will assume that the PIN consists of 4 decimal digits P_1, P_2, P_3, P_4, as it typically does. Associated with the user's account is also an identifying account number known as the *Personal Account Number* (PAN), which is used by the bank to identify which account a transaction refers to.

When a PIN is entered at an ATM, it has to be sent to the issuing bank

or other authorised entity for verification. Since knowledge of the PIN gives the right to transact with the account, it is imperative that the PIN is kept secret. Therefore, the PIN is encrypted under a transport key, and the resulting *Encrypted PIN Block* (EPB) is then transmitted through the ATM network. Before arriving at its destination, the encrypted PIN passes through several nodes, each of which decrypts the EPB, verifies the resulting PIN block format, re-formats it if necessary, and re-encrypts it under the transport key shared with the next node.

All these sensitive operations on the PIN are handled within tamper-proof devices, known as *Hardware Security Modules* (HSMs), which are commissioned to physically protect cryptographic keys and PINs from eavesdroppers and malicious employees. They typically include a dedicated cryptoprocessor that carries out all the PIN processes, and a small amount of memory where the sensitive data is stored. Any detected malicious attempt to access the memory will cause the immediate erasure of the protected data. HSMs are regulated by security APIs, which are charged to carry out a strictly controlled set of PIN manipulation operations according to the user's commands, ensuring that the sensitive values are not revealed in an unencrypted form outside the HSM. Within the HSM, the PIN is formatted into a 64-bit clear PIN Block (PB). It should be noted that there exists a multitude of recognised formats for the clear PIN block, including the ISO-0 and the VISA-3 formats, which we will describe in the following section. Standard financial APIs implement functions to generate and verify PINs, translate PINs between different encryption keys and formats used in the different zones of the ATM network, and support a whole host of key management functions.

2.2 PIN block attacks

The security issues of the API design have until recently been overlooked by software engineering researchers, who mainly concentrated on avoiding implementation errors or verifying the correspondence of the API implementation to its specification. In the last few years, however, a raft of attacks have been discovered, which compromise the reliability of many widely-used security APIs. The common theme behind these attacks is to produce an unexpected sequence of transactions, which can trick a security module into revealing a secret in a manner contrary to the device's security policy. Although patches have been issued by manufacturers, API flaws continue to be found. Some of the discovered attacks relate to weaknesses in the key management architecture of financial HSMs ([Bon04],[BA01], [Bon01]), see also section 3.1). In this thesis we consider a spe-

cific class of attacks against financial security APIs: the *PIN block attacks*, also known as information leakage attacks, which arise from flaws in the PIN processing command set. These kinds of attacks involve the intruder tweaking inputs to API functions away from their usual values, and exploit the responses given by the API that leak information about the PIN. This information is usually small, concerning only a few bits, however when accumulated through repeated calls of specific functions, it can eventually reveal the entire PIN, or bring it within the scope of a brute-force search.

We are concerned with three groups of PIN block attacks, following the terminology used by Steel in [Ste06]. The categorisation is done according to the kind of operations that are required by each attack, so that all variations of attacks belonging to the same group can be described by a common general strategy. The first family of attacks, which is summarised under the title "ISO-0 attacks", was discovered by Clulow in [Clu03, §3.5.3-3.5.4]. The decimalisation table attacks, which constitute the second family of attacks, were detected independently by Bond and Zieliński [BZ02] and Clulow [Clu03, §3.5.5], while the description of how to perform brute-force guessing, which we will see last, is also thanks to Clulow [Clu03, §3.5.8]. In [BC04], Bond and Clulow discuss how difficult it is to successfully eradicate the vulnerabilities arising from the PIN processing operations, and present some new examples of information leakage that makes use of the statistical distribution of PIN blocks, exploiting non-uniformities resulting during PIN generation. The hazard of the several variants of PIN block attack strategies and the specific weaknesses of each function supported by the financial security API have been recently summarised in [BO06].

The PIN block attacks described below, require first of all the adversary having access to an HSM and being able to generate API calls (the necessary API functions depend on the attack). All API functions use cryptographic keys, which are usually kept outside the HSM, encrypted under a master key. Thus, in order to be able to obtain answers from the HSM, the adversary must have the necessary encrypted PIN encrypting/decrypting keys, to use them under the terms of the API. Finally, the adversary must also have a valid EPB. These requirements entail the help of a bank insider. It should be underlined that, as mentioned in [BO06], the malicious employee does not need to be an insider of the issuing entity, since the HSMs implementing PIN processing APIs usually separate logically and/or physically the functionality required for the issuing facility from the functionalities required for the verification of the PIN, which is considered to be less security sensitive. Until the PIN is finally verified by the issuing entity, it passes through several HSMs, which perform a series of operations on the PIN.

The PIN block attacks abuse weaknesses of functions involved in these operations, like the translation, change account number and verification functions. Hence, the malicious employee does not need to afford any special privileges required for handling the HSM in secure or authorised mode, or knowledge of issuer's keys. All he needs is to gain logical access to an HSM, which may be outside of the issuer's environment, and generate API calls. So, even if a financial organisation applies all necessary security measures to its own facilities and disables vulnerable functions, it may still be exposed to PIN block attacks, because it cannot guarantee that the other entities participating in the ATM network adhere to these measures.

2.2.1 ISO-0 attack family

The ISO-0 format, also known as ANSI X9.8, VISA-1, VISA-4 and ECI-1, is a format specified by the ANSI standards on retail banking. It is characterised by the fact that the 12 rightmost PAN digits are prepended with 4 zeros and exclusive-ored (XOR-ed) with the PIN to yield the clear PIN block:

$$\begin{array}{lcccccccccccccccc}
\texttt{B1} = & 0 & 4 & \texttt{P}_1 & \texttt{P}_2 & \texttt{P}_3 & \texttt{P}_4 & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} & \texttt{F} \\
\texttt{B2} = & 0 & 0 & 0 & 0 & \texttt{A}_1 & \texttt{A}_2 & \texttt{A}_3 & \texttt{A}_4 & \texttt{A}_5 & \texttt{A}_6 & \texttt{A}_7 & \texttt{A}_8 & \texttt{A}_9 & \texttt{A}_{10} & \texttt{A}_{11} & \texttt{A}_{12}
\end{array}$$

$$\texttt{ISO-0 PB} = \texttt{B1} \oplus \texttt{B2}$$

Each character in the above blocks corresponds to 4 bits. The `F`s are hexadecimal, the $\texttt{P}_\texttt{i}$s are the PIN digits and the $\texttt{A}_\texttt{i}$s are the digits of the PAN. The 0 at the beginning of `B1` is the control value, indicating that the block is in format 0. The 4 at the second position of `B1` specifies that the PIN consists of 4 digits. The XORing of the encrypted PIN block against the account number serves to diversify clear blocks that contain the same PIN. This way, even if two users have the same PIN number, the associated encrypted PIN blocks will bear no resemblance because of the different account numbers. This prevents the so-called "code book" attacks, which involve the adversary constructing a table listing every PIN (plaintext) and the encryption of that PIN under a given key (ciphertext). The PAN offers a variation that makes the code books required for a successful PIN recovery significantly larger.

The standard version of the ISO-0 attack is against functions to which the PAN is supplied, exploiting the error check performed by the HSM during the PIN extraction process. Under normal operation, the EPB is provided to the HSM, which decrypts it to find the corresponding clear block PB. Then the PB is XOR-ed with the supplied PAN to reveal `B1`, from which the PIN digits can be extracted. At this point, the HSM performs an error check in order to confirm

that the extracted PIN is valid, i.e. that all its digits are of decimal value. If the test fails, then the process exits by reporting an error. Suppose, now, that instead of typing in the correct PAN (effectively B2), the adversary provides a modified PAN $\mathtt{B2} = \mathtt{B2} \oplus \Delta$, where say $\Delta = \mathtt{0000x000000000}$, with $\mathtt{x} \neq 0$ having a properly chosen value. The PIN extraction process in this case will proceed like this (taken from [Clu03, p. 75]):

INPUTS(EPB, B2′)

1. $\mathtt{PB} = \mathtt{d_k(EPB)}$
2. $\mathtt{B1} = \mathtt{PB} \oplus \mathtt{B2}$
   $= (\mathtt{P1} \oplus \mathtt{B2}) \oplus (\mathtt{P2} \oplus \Delta)$
   $= \mathtt{P1} \oplus \Delta$
   $= 0\ 4\ \mathtt{P_1}\ \mathtt{P_2}\ \mathtt{P_3}\ \mathtt{P_4}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F}\ \mathtt{F} \oplus$
   $0\ 0\ 0\ 0\ \mathrm{x}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$
3. Extract PIN as $\mathtt{P_1P_2P_3P_4} \oplus \mathtt{00x0}$ and test to confirm if it is a valid PIN
4. If test fails then exit with an error
5. Process the rest of the function as normal

If for example the value of 'x' is 8, and $\mathtt{P_3}$ is 0, 1, 8 or 9, the error check will still pass, since the XOR-ing will give a decimal value. If, however, $\mathtt{P_3}$ is in the range 2-7, an error will be reported. By doing this in an iterative manner, and by observing the effects, the attacker can narrow down the third digit to a pair of possible values, 0 and 1, 2 and 3 etc. This is because P_i and $P_i \oplus 1$ result in identical patterns of passes and fails when XOR-ed against the values of 'x'. The same process can be followed for narrowing down $\mathtt{P_4}$, by putting the 'x' in the position of $\mathtt{A_2}$. However, it can't be applied on the first two digits, since these do not overlap with the PAN part of B2. It should also be noted that the same attack strategy applies to the ISO-3 format, which involves besides the account number the use of random data as well.

An extended version of the attack presented above, which exploits the translate and reformat functions of the HSM, allows the full recovery of the PIN. The extension involves the attacker lying not only about the PAN, but also about the format of the PIN block. More specifically, the attacker submits an ISO-0 formatted PIN to the reformat function, but specifies it as a PIN block of another format, namely a VISA-3 format. In this quite simple format, the PIN starts from the left most digit and ends by the delimiter 'F'. For a 4-digit PIN the VISA-3 format looks like this:

VISA-3 $\mathtt{PB} = \mathtt{P_1P_2P_3P_4FXXXXXXXXXXX}$

where 'X' are 4-bit hexadecimal pad digits, all of the same value. Assuming for simplicity a null PAN, the format masquerading will have the following effects on the reformat process (based on [Clu03, p. 75]):

INPUTS (EPB, B2, Format′)

1. $\mathtt{PB} = \mathtt{d_k(EPB)}$.
   Since the clear PIN block is actually in ISO-0 format:
   $\mathtt{PB} = \mathtt{B1} \oplus \mathtt{B2}$
   $= \mathtt{04P_1P_2P_3P_4FFFFFFFFFF} \oplus \mathtt{0000000000000000}$
   $= \mathtt{04P_1P_2P_3P_4FFFFFFFFFF}$
2. The HSM looks for a left-justified F-terminated string following the VISA-3 formatting rules, and extracts $\mathtt{04P_1P_2P_3P_4}$, assuming it to be a 6-digit PIN.
3. The PIN is then formatted into a new ISO-0 PIN block as:
   $\mathtt{B1} = \mathtt{0604P_1P_2P_3P_4FFFFFFFF}$
   $\mathtt{B2} = \mathtt{00000000000000000}$
   $\mathtt{PB} = \mathtt{B1} \oplus \mathtt{B2} = \mathtt{0604P_1P_2P_3P_4FFFFFFFF}$
4. Output : $\mathtt{EPB} = \mathtt{e_k(PB)}$

By following this strategy, we manage to expose the first two PIN digits to the previous ISO-0 attack. Consider now the effect of using a non-null PAN, by providing for example $\mathtt{B2} = \mathtt{00000x0000000000}$. The clear PIN block will be obtained like this:

$$\begin{array}{lcccccccccccccccc} \mathtt{PB} = \mathtt{B1} \oplus \mathtt{B2} & & & & & & & & & & & & & & & & \\ = & 0 & 4 & \mathtt{P_1} & \mathtt{P_2} & \mathtt{P_3} & \mathtt{P_4} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{F} & \oplus \\ & 0 & 0 & 0 & 0 & 0 & \mathtt{x} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \end{array}$$

According to the VISA-3 formatting rules, there are 3 possible outcomes for the extracted PIN. If the result of $\mathtt{P_4} \oplus \mathtt{x}$ is a decimal, then the PIN will be extracted as the 6-digit PIN $\mathtt{04P_1P_2P_3P_4} \oplus \mathtt{00000x}$. If $\mathtt{P_4} \oplus \mathtt{x} = \mathtt{F}$ then the PIN will be extracted as the 5-digit PIN $\mathtt{04P_1P_2P_3}$. If, however, $\mathtt{P_4} \oplus \mathtt{x} = \mathtt{F}$ is in the hexadecimal range A-E, then an error will be reported and the call will fail. This method allows the attacker to identify uniquely the value of all PIN digits.

The important observation about all versions of the ISO-0 attack family is that they are digitwise, i.e. each digit is determined consecutively and independently of the other digits.

2.2.2 Decimalisation table attacks

This family of attacks is applied against PIN verification functions. It exploits PIN generation algorithms, like the broadly used IBM 3624 scheme, where the customer's PIN is calculated by encrypting the PAN under a secret key called a *PIN Derivation Key* (PDK). The resulting ciphertext is converted into hexadecimal, and all but the first four digits are discarded. In order to convert these digits into a PIN which can be typed onto a standard numeric keypad, a *decimalisation table* or *dectab* is used, which maps each hexadecimal value to a decimal. A typical decimalisation table looks like this:

Hex. value	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec. value	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

To permit the cardholders to change their PINs, some PIN generating schemes use an *offset*, calculated by subtracting modulo 10 the original PIN (generated by the above described process) from the customer's chosen PIN. This offset is not regarded to be secret, and is stored along with the PAN in the database of customer account records.

The verification process conducted by an HSM is analogous to the PIN generation process. It first encrypts the provided PAN under the PIN verification key, which is equivalent to the PDK, decimalises the corresponding ciphertext, which is termed as Intermediate PIN (IPIN), and adds to it the offset modulo 10. The result is then compared with the supplied EPB. For one reason or the other, the dectab is usually not stored within the HSM, but is instead passed as a parameter along with the EPB, the PAN and the offset. Now, suppose that the attacker supplies a modified dectab, by changing the ith entry mapping $h_i \in [0..16]$ to $d_i \in [0..10]$, so that h_i is mapped to a new decimal value $d'_i \neq d_i$. If the IPIN contains the value d_i, then the verification will fail, otherwise the EPB will still pass the verify call. By repeating this procedure, each time altering a single entry of the dectab, the attacker can determine which digits constitute the PIN. Ten calls to the verification function are enough, since we only care about the digits of the PIN after the decimalisation.

However, this tactic can't reveal the positions of the digits, nor which digits are repeated and how many times, if there are any digits that appear more than once. This can be achieved by altering the value of the supplied offset for the altered version of the dectab that led to a hit. By advancing the offset by one at each position, and then at every combination of positions, until the offset passes, the attacker can identify the location of the digits present in the PIN.

The strategy of the decimalisation table attack can be better illustrated through an example. Suppose that the decimalised, original PIN is 8353. The attacker starts by trying modified decimalisation tables, each time altering a single entry d_i, starting from $i = 0$. For the following dectab, corresponding to $i = 3$, the verification will fail:

Hex. value	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec. value	0	1	2	**4**	4	5	6	7	8	9	0	1	2	3	4	5

At this point, the attacker knows that there is at least one 3 somewhere in the PIN. He can now try to determine the position and the number of the present 3s, by iterating through the set of possible offset values and checking for a match. For simplicity, we assume that the correct offset is 0000 (the account holder has not changed his PIN). The sequence of the offset inputs supplied by the attacker and the corresponding HSM responses will be the following:

```
Attacker set offset:  0001 → Verification:  fail
Attacker set offset:  0010 → Verification:  fail
Attacker set offset:  0100 → Verification:  fail
Attacker set offset:  1000 → Verification:  fail
Attacker set offset:  0011 → Verification:  fail
Attacker set offset:  0101 → Verification:  pass
```

Now the attacker knows that the PIN is of the form $x3x3$. By repeating the same procedure, he can further determine the value of the rest of the digits, denoted by 'x'.

2.2.3 Brute force guessing attacks

Because the PIN space is relatively small (10^4), it is crucial that security APIs are set up in such a way that it would be infeasible for an adversary to carry on an attack based on exhaustive trial and error. However, many API functions have weaknesses that allow the performance of brute force guessing. Although not very efficient by themselves, exhaustive search attacks can pose considerable threats when combined with other kinds of attacks, which have managed to reduce the search space, so that the likelihood of a correct guess can become dangerously high.

The similarity in operation between the verification and the check value function supported by many APIs can be exploited for the performance of a brute force attack. The check value command is used to encrypt a 64 bit binary block of zeros under the supplied encryption key. The output is called the check value

of the encryption key. The role of the check value function is to ensure that the encryption key has been imported correctly. Consider now the case, where the attacker can supply a block of 0s as the PAN to the PIN verify function described in the previous section. The result of the encryption stage of the PAN will be the same as the result from the check value call, when the PDK is given as input. Since the decimalisation table is known, it is trivial to calculate the decimalised version of the output of the check value function, and thus we know the IPIN value. The PIN can now be calculated as IPIN + OFFSET (mod 10), where the OFFSET value can be recovered by exhaustive search: starting with 0000, we increase the OFFSET by one, until we find a value that verifies successfully. This process will require at most 10^4 calls of the verification function, as many as the values of possible offsets, plus a call to the check value function to obtain the check value of the PDK.

2.3 Summary

We have introduced the notion of security APIs and how they can control the use of tamper-proof hardware security modules (HSMs) that are used in cash machine networks. Since in a typical ATM transaction the PIN needs to be decrypted, re-encrypted and verified several times, it is essential that all these manipulations are performed in a secure and precisely controlled manner, protecting the sensitive data from corrupt employees. We have seen however, that the commands of the APIs can under certain circumstances be used in unintended ways to conjure information about the secret data. In section 2.2, we have discussed in detail the case of PIN block attacks, which are attacks against the PIN processing operations of security APIs. More specifically, we have described the strategy behind three families of PIN Block attacks. The ISO-0 attacks, which are a family of digitwise attacks, exploit the reformat function of the HSM, which allows the encrypted PIN block to be translated into another format. The decimalisation table attacks, which are based on the ability of the attacker to modify the decimalisation table used by the HSM during PIN verification. Finally, brute-force guessing attacks, which take advantage of the check value function supported by many APIs, and can be helpful for finishing off the cracking of the PIN.

Chapter 3

Formal analysis for security issues – Literature review

Security protocols are notoriously hard to design due to the range of unintended interactions between principals that can arise, and which are very hard to predict. Many published protocols have been implemented and deployed in real applications, only to be found flawed years later. The detection of possible attacks requires detailed and laborious manual inspection, which cannot guarantee that a novel flaw, that escaped the current analysis, will be discovered in the future. As Dijkstra put it, such an analysis "can only show the presence of errors, not their absence" [Dij70]. There is therefore a need for formal automated techniques to assist the design process of security protocols.

The two main approaches to formal verification are *model checking* and *theorem proving*. Model checking starts with a model described as a state transition system, and discovers whether certain asserted hypotheses, encoded as properties in the specification language of the model checker, are valid in the model. If they are not, it can produce counterexamples. The properties to be checked can refer to qualitative requirements, like "is my bank web-service secure?", or to quantitative ones, like "how reliable is my voice-over-IP service?", assuming a measure of reliability. Model checkers are able of identifying trends or anomalies on a system, and can be also used to examine worst-case (or best-case) scenarios. Theorem proving, on the other hand, starts from a set of initial axioms and searches at a higher level of abstraction for a chain of inference steps, which can prove that a particular statement always holds. Typically, a theorem prover requires some level of human guidance in order to find which of the available rules of inference to apply at each stage.

Formal verification methods have been broadly used for aiding human analysis of security protocols, and some have led to the discovery of new attacks. These

methods can also be useful for analysing security APIs, which are a very close friend of security protocols: the individual calls of commands to the security device controlled by an API can be thought as runs of a security protocol between two agents: the user and the device, which plays the role of the trusted third party. However, despite their similarity to security protocols, there are some features inherent to the API commands, that make the tools which work well for analysing protocols to perform significantly worse when applied to the analysis of security APIs. First of all, the non-determinism, resulting from the fact that there is no enforced ordering on the interactions between the intruder and the security component, can cause a large blow-up to the size of the search space. Moreover, the bit-by-bit operations specified by the API, as opposed to the more abstract level of security protocols, are characterised by non-trivial algebraic properties, which can also lead to a massive increase of the search space. Therefore, security APIs have been shown to pose new challenges to formal analysis, demanding the modification of existing tools and the special focused treatment of bitwise operators.

3.1 Formal analysis of security APIs

There are several papers that address automated formal analysis techniques of APIs for financial ATM network processing, regarding attacks against their key-management scheme. Bond and Anderson ([BA01],[Bon04]) have discovered manually a number of attacks on the IBM 4758 Common Cryptographic Architecture (CCA), based on exploiting the way the API constructs keys of different types supported by the CCA. In his PhD thesis, [Bon04] Bond attempted to rediscover an attack he had already detected by hand on the IBM 4758 CCA, by using the theorem prover SPASS. The attack in question exploits of the Import command of the API, which allows the adversary to change the type of a key, taking advantage of the properties of the XOR operator. However, this formal analysis approach failed to lead to a fully automatic genuine rediscovery, mainly because of the infinite branches of the search space resulting from the repeating combinations of terms being XOR-ed.

In [GSJ+05] a model-checking-based tool for finding API-level attacks, that can compromise the security of software components, is presented. What discriminates the proposed framework from previous tools is that, instead of just identifying potential vulnerabilities, it generates exploits against them, i.e. sequences of operations that attack a vulnerability. This is particularly helpful in order to confirm if a localised vulnerability poses a real threat or not. Emphasis

is given to capturing the low-level details of the operations supported by the API, so that the produced model is close to the concrete system. Two case studies are presented: the first shows how format-string exploits can be modelled as API-level exploits, and the second addresses the IBM 4758 CCA . In the latter case, it is considered how an adversary can use a set of specified inference rules to enhance his knowledge about the key. The Key Import attack against a rather restricted version of IBM 4758 CCA, including only 2 of the available commands, was rediscovered by using the exploit generating tool.

In [Ste05] Steel introduces the development of XOR constraints, which can provide an efficient handling of bitwise XOR operations employed by many security systems. As we have already seen, the combinatorial blow-up caused by the algebraic properties of XOR prohibited Bond from formalising even a small subset of the IBM 4758 CCA. The framework proposed by Steel uses a modified version of the daTac theorem prover [Vig94], that deals with constraints specifying equality of terms modulo XOR. These constraints build upon several properties of the XOR operator, like its associativity and commutativity, or the fact that XOR-ing a term with itself produces the identity value. By applying the framework on the command set of the 4758 API, Steel was able to rediscover the Key Import Attack, as well as some other kinds of attacks.

A team of researchers at Cambridge University, including Bond, have also tried to formally rediscover the attacks on the IBM CCA, using a methodology for modelling security APIs with the theorem prover Otter[1], which takes input in first-order logic [YAB+05]. User knowledge of a message or value is represented via a predicate. To avoid the production of infinite numbers of terms, a number of restrictions are imposed on the ways in which the intruder can deduce new terms from existing knowledge, so that only meaningful, well-defined forms are allowed. The applied optimisation techniques manage to reduce the state space size without impeding any attacks to be discovered. Following this methodology, Youn et al. were able to rediscover all known attacks on IBM's CCA API.

In his MSc thesis [Kei06], Keighren has also managed to rediscover all known attacks on the 4758 API, by using the constraint-logic-based model checker CL-AtSe[2], which allowed the modelling of a greater set of commands than the previous attempts, which were based on the use of theorem provers. The formal analysis of the IBM recommendations showed that there may be situations where they cannot prevent a newly discovered variant of attack. CL-AtSe only checks security for a small number of runs of the protocol, usually only three, which

[1]Available at www.cs.unm.edu/ mccune/otter
[2]Available at www.loria.fr/equipes/cassis/softwares/AtSe

means that there is no guarantee of security if more sessions are executed. In [CKS07] a new class of well-formed security protocols using XOR, including IBM CCA API, is defined and proved to be decidable for an unbounded number of sessions.

The AnaBlock system investigated in this thesis, presented by Steel in [Ste06], addresses the case of PIN block attacks we have already discussed, as opposed to the above presented work which focuses on flaws in the API operations for the management of cryptographic keys. The literature related to the manual discovery of PIN block attacks and the threats they pose has already been presented in section 2.2. As far as we are aware, there has been no other attempt of formally analysing attacks based on flaws in the PIN-processing commands supported by financial HSMs. The framework proposed by Steel, focuses on modelling known attacks rather than searching for new ones, although the modelling of API commands allowed the discovery of a previously unknown attack variation. However, AnaBlock does not guarantee that there is not an unknown attack that can compromise the security of the API under investigation. The aim is to assess how vulnerable a set of API commands is, based on the knowledge of already detected attacks. The assessment of some measure of risk or security requires the performance of quantitative analysis, while the papers mentioned above were only concerned with checking a safety condition. A detailed description of the AnaBlock system is presented in section 5.1.

Another important distinction between the PIN block attacks and the previously mentioned kinds of attacks, is that the analysis of the former requires the introduction of likelihood to represent the probabilities of the possible HSM outcomes, according to the distribution of PINs. Thus, the model of PIN block attacks has to be endowed with both non-determinism and probability, which gives rise to a Markov Decision Process, in contrast with the models presented above, which had to deal only with non-determinism. The MDP is analysed by the probabilistic model checker PRISM. Probabilistic formal methods have been broadly used for the analysis of many security protocols. Some of the work done in this area is discussed in the following section.

3.2 Probabilistic analysis of security protocols

Sheyner et al. presented in [SHJ+02] an automatic technique for constructing and analysing attack graphs, i.e. graphs modelling a series of exploits against network vulnerabilities, based on a modified version of the NuSMV symbolic model checker. The network is modelled as a finite state machine, where the

state transitions correspond to attacks, such as a buffer overflow or a remote login attack, without any insight into the lower-level operations that constitute these attacks (contrast this to the API models of the previous section, where the transitions correspond to the performance of atomic operations like XOR-ing). The analysis of the graphs is concerned with finding the most cost-effective way of guaranteeing the security of the network. The graph is annotated with transition probabilities, according to the likelihood of each attack to be detected by the security mechanisms. Thus, the graph can be interpreted as an MDP. A reward assigning mechanism is employed, by giving all nodes where the intruders goal has been achieved the benefit value 1, and all other nodes the benefit value 0. Then an iteration algorithm is applied to extract the optimal attack selection policy for the intruder and the expected likelihood of this policy to succeed. The goals of the AnaBlock system are quite similar.

In [Shm04] Shmatikov uses PRISM (see chapter 4) to formally analyse the Crowds protocol for maintaining anonymity during communication with a Web server. The fundamental characteristic of the protocol is that it provides protection against groups of collaborating local eavesdroppers, called crowds. The protocol relies on probabilistic message routing, so that even if some group members share collected information with the adversary, the latter is not likely to distinguish between true senders and randomly selected forwarders. The model of the system has to capture the relative probabilities of certain observations by the adversary. Since there is no non-determinism in the protocol specification, the system is represented as a DTMC. PRISM is used to check the validity of a number of anonymity properties. The analysis revealed potential vulnerabilities of the Crowds system, including a previously unknown flaw, lying on the fact that the adversary's confidence in a user's identity increases with the increase of crowd size.

In [NS06] the probabilistic model checker PRISM is used for the verification of a number of properties on three contract signing protocols. The resulting models combine non-determinism, corresponding to the actions of a misbehaving participant, and probability, corresponding to the behaviour prescribed by the protocol specification, and are hence represented as MDPs. The main property that needs to be verified with respect to these protocols is fairness, which, informally speaking, implies that, at the end of the signing process, either both parties have the counterpart's signature or none of them does. PRISM is used to calculate the probability that each protocol terminates in a fair state as a function of the number of messages that must be exchanged. Additional properties, like timeliness, are also evaluated. The experiments on the 3 case-studies demonstrate that prob-

abilistic model checking can be a useful tool for the quantitative characterisation of probability-based properties of security protocols.

For completeness, we also mention the work done in analysing protocols subject to off-line guessing attacks in [DMV04]. Off-line guessing attacks are those for which the intruder doesn't participate in any communication with agents during the guessing phase, in contrast with on-line guessing where the intruder employs guesses while interacting with other agents (in the case of an API the honest agent the intruder can consult is the security module). As we have already seen, the PIN block attacks involve the intruder revising his knowledge about the PIN according to the effects of particular commands typed in the HSM, and thus they belong to the category of on-line attacks. The deduction system for modelling off-line guessing proposed in [DMV04] is only concerned with expressing whether a vulnerability exists or not, without caring about the probability of the intruder to guess correctly. Although this formalisation does not include the notion of probability and only considers the case of off-line guessing, it resembles the AnaBlock system in the sense that it is an attempt at providing a general model for verifying whether a protocol is vulnerable, capturing a generic class of attacks.

3.3 Summary

We have highlighted how significant the use of formal automated methods, based on model checkers or theorem provers, is in the analysis of security protocols. Security APIs, although similar to security protocols, are different enough to demand the adoption of more subtle approaches to conventional techniques. In section 3.1, we have seen that the main attempts of researchers to apply formal analysis to security APIs aimed at the automatic discovery of attacks against their key management scheme. To achieve this, researchers had to modify the existing formal tools, in order improve their performance and meet the specific requirements of security APIs. We have outlined how these attempts differ from the AnaBlock system for formal analysis of PIN block attacks. The modelling of the latter requires the introduction of probability, which has to be properly handled by an applied formal tool during the analysis phase. In section 3.2, we have briefly described several papers that deal with the formal verification of security protocols with probabilistic behaviour and their relation to the AnaBlock system.

Chapter 4

PRISM: A probabilistic model checker

PRISM is the tool we use for analysing the models of PIN block attacks, and a basic understanding of its theoretical foundations, the techniques it applies and its incorporated facilities is a sine qua non for our goal to investigate alternative representations. The following presentation of PRISM's features and capabilities is selective, with respect to the issues that arose throughout our work. Thus, some topics are discussed in more detail than others, because they pertain to problems we faced when using the model checker for the analysis of our models. It should be made clear that the majority of the information presented in the followings has been sourced from Dave Parker's PhD thesis [Par02], with additional material drawn from various papers published by the PRISM developing team, comprising Marta Kwiatkowska, Gethin Norman and Dave Parker, in collaboration with other researchers in the field ([HKN+03], [Kwi03], [KNP02]). Most of the figures were taken from a set of lectures on PRISM available at www.prismmodelchecker.org/lectures. Some information was taken from the book [HR04] and Wikipedia[1].

4.1 Overview of the probabilistic model checking process

Probabilistic model checking is an automatic, formal procedure for establishing the correctness, performance and reliability of systems that exhibit random behaviour. Such systems are often encountered in a wide range of diverse domains, including communication and multimedia protocols, biological process modelling,

[1] www.wikipedia.org

security, distributed algorithms, game theory and others. The common feature of all these systems is that they involve unpredictable characteristics, whose representation requires the encoding of the probability of making a transition between states, instead of simply the existence of such a transition. As opposed to conventional model checkers, which can only specify the validity of a given formula in temporal logic, probabilistic model checking additionally involves the calculation of likelihood of the occurrence of a certain event. Thus, it allows the expression of quantitative statements, such as "the probability of cost exceeding an upper bound is 95%".

PRISM is a state-of-the-art symbolic probabilistic model checker (see section 4.4.1 for the meaning of symbolic). The tool takes two inputs: a finite-state model description written in the high-level PRISM language, described in section 4.3.1, and a properties specification, consisting of a number of temporal formulae, as presented in section 4.3.1. PRISM parses the description of the model, constructs a probabilistic model of the appropriate type (see section 4.3.2), computes the set of reachable states, and identifies any deadlock states, i.e. states which are reachable but with no outgoing transitions. Deadlock states are regarded to constitute an error, therefore self-loops must be added to these states to resolve the situation. The properties specification is also parsed, and then model checking is performed to determine which states of the model satisfy each property. This procedure is illustrated in Figure 4.1. For more information on the tool and its capabilities you can visit PRISM homepage in www.prismmodelchecker.org.

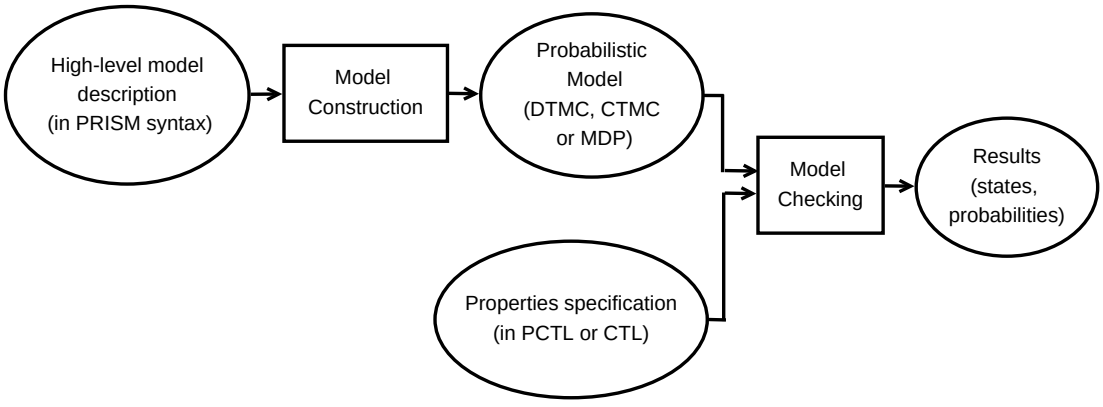

Figure 4.1: Overview of PRISM architecture.

4.2 Probabilistic models

PRISM supports three types of stochastic models: Discrete-Time Markov Chains (DTMCs), Continuous-Time Markov Chains (CTMCs) and Markov Decision Processes (MDPs). They all satisfy the *Markov property*, which implies that, if the current state of the model at time t is known, transitions to a new state at time $t' > t$ are independent of all previous states. *DTMC*s are typically used to represent synchronous systems where the probability of making transitions between states is given by discrete probability distributions, i.e. each transition corresponds to a discrete time step. They are defined by a function $P : S \times S \to [0, 1]$ satisfying $\sum_{s \in S} P(s, s') = 1, \forall s, s' \in S$, where S is a finite set of states. The function P can be thought as a matrix, whose entry $P(s, s')$ gives the probability of making a transition from state s to state s'. This square matrix is known as the *transition probability matrix*. *CTMC*s extend DTMCs by allowing transitions to occur in real time. They are defined by a *transition rate matrix* $R : S \times S \to \mathbf{R}_{\geq 0}$, where $R(s, s')$ gives the rate at which transitions between states s and s' occur. The probability of a transition from s to s' to be triggered before t time units is $1 - e^{R(s,s') \cdot t}$.

In addition to choices with probabilistically determined outcome, it is often helpful or even mandatory to include non-deterministic decisions, whose outcome is left completely unspecified, in cases where it is unrealistic or impossible to associate the possible outcomes with concrete probabilities. DTMCs and CTMCs are unable to model such non-deterministic aspects of systems. *MDP*s generalise DTMCs by supporting both probabilistic and non-deterministic choices to be made, and are particularly useful for modelling decision-making in situations where outcomes are partly random and partly under the control of the decision maker, like for example when modelling the strategy of an attacker against a security protocol. They are defined by a transition function $Steps : S \to 2^{Dist(S)}$ that maps each state $s \in S$ to a finite, non-empty set $Steps(s)$ which represents the non-deterministic choices between several discrete probability distributions over the state space S. As far as we are aware PRISM is the only fully developed tool to provide quantitative model checking of MDPs.

Like DTMCs and CTMCs, MDPs can also be represented by a matrix. For fully probabilistic models, the corresponding matrices are square, having as entries either discrete probabilities in the case of DTMCs, or the parameters of the exponential distribution in the case of CTMCs. In an MDP, each state contains several non-deterministic choices, each of which specifies a discrete probability for every state. In terms of a matrix, this equates to each state being represented by several different rows, where each row corresponds to a single non-

deterministic choice. Thus, the matrix representing the transition function will have $|S|$ columns and $\sum_{s \in S} |Steps(s)|$ rows. An example, taken from the PRISM lecture slides, is illustrated in the following figures. Figure 4.2 shows an MDP consisting of 4 states $s1, s2, s3$ and $s4$, which are drawn as circles. The transitions are labelled with their associated probabilities, and are joined by an arc if they correspond to the same non-deterministic choice. In Figure 4.3 the matrix that represents the MDP is shown. For clarity, the choices for each state are separated with a horizontal line.

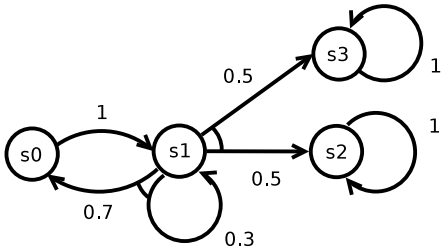

Figure 4.2: A 4 state MDP.

$$\mathbf{Steps} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ \hline 0.7 & 0.3 & 0 & 0 \\ 0 & 0 & 0.5 & 5 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

Figure 4.3: The corresponding **Steps** matrix.

4.3 Model description and property specification in PRISM

In this section, we describe how the probabilistic models discussed in the previous section are specified using PRISM, and how the properties we want to evaluate on these models can be expressed to PRISM formalism. More details about the the PRISM language can be found in the PRISM manual[2].

4.3.1 The PRISM language

PRISM provides a high-level, state-based modelling language, based on the Reactive Modules formalism suggested in [AH99]. The two basic elements of the PRISM language are *modules* and *variables*. A model is specified by the parallel composition of several modules, which can be seen as partner processes that interact with each other. Each module contains a number of *local* variables, whose values determine its local states. The local variables can be read by all modules, however they can only be changed by the module they belong to. *Global* variables, that can be updated by all modules, are also supported. The global states

[2]Available at www.prismmodelchecker.org/manual.

of the whole model are defined as the valuation of the variables for all modules. The behaviour of each module is described by a set of state transition rules, which are specified using guarded commands of the form:
`[] <guard> -> prob_1 : <update_1> + ... + prob_n : <update_n>;`
where `<guard>` is a predicate over all model variables, and `<update_i>` corresponds to one of the possible probabilistic transitions that can be executed if the guard condition evaluates to true. The updates use primed variables of the form `(x_i′=some_value)` to denote the next values of variables. The nature of `prob_i` depends on the type of model described, reflecting either a probability, in the case of DTMCs and MDPs, or a rate in the case of CTMCs. A transition of the whole model comprises transitions for one or more of its component modules. These transitions can be performed in an asynchronous manner, where a single module makes a transition independently, while the others remain in their current state, or synchronous, where two or more modules make a transition simultaneously. The commands can be labelled with *actions*, which can be exploited to synchronise one or more modules.

PRISM can be used to reason not just about the probability that a model behaves in a certain fashion, but also about a wider range of quantitative measures relating to model behaviour, that include notions like "expected time", "expected steps" or "expected power consumption". This is enabled through the *reward* mechanism, which allows the assignment of rewards, i.e. real values, to transitions indicated by some specific action labels and states that satisfy some desired expressions. At this point we would like to mention a limitation of the PRISM reward techniques we discovered when experimenting with the tool, and that had not been realised by the PRISM team before. PRISM does not allow the assignment of action labels to commands that update global variables, in order to avoid situations where two synchronising commands from two different modules assign different values to a single global variable. However, this constraint imposes unnecessary restrictions to the assignment of transition rewards, since the latter is also implemented through the use of action labelling. To overcome these restrictions, the involved global variables must be turned into local variables, and this conversion often demands considerable changes in the representation of the whole model to be made.

4.3.2 Property specification in PCTL

Like most model checkers, PRISM uses temporal logic to identify the properties that can be evaluated. Properties are expressed in a language based on *Probabilistic Computational Tree Logic* (PCTL), which is interpreted over DTMCs and

MDPs, and *Continuous Stochastic Language* (CSL) for reasoning on CTMCs. PCTL was defined in [HJ94] as an extension of non-probabilistic CTL. It introduces a new probabilistic operator $P_{*\lambda(\phi)}$, where $*$ is a comparison operator, λ a probability threshold and ϕ a formula in CTL. So, $P_{>0.5(\phi)}$, for example, states that the probability of reaching a state that satisfies ϕ is greater than the bound 0.5. CSL can be seen as an extension of PCTL, that introduces additional operators that allow reasoning about the real-time behaviour of CTMCs.

PRISM property specification language is very powerful and supports the model checking of a wide range of features in a model. The principal operators of the language are `P`, `S` and `R` which refer, respectively, to the probability of an event occurring, the long-run probability of some condition being satisfied and the expected value of the models rewards. The role of these operators can be better illustrated through some examples. For clarity, we recall here some basic temporal operators: `F`, which means "finally, somewhere in the subsequent path", `G`, which means "globally, in the entire subsequent path" and `U` which is the strong until. `p U q` holds on a path if `p` holds continuously until `q` holds, with the additional requirement that `q` does hold in some future state.

- `P<0.3 [ true U (cost > 100) ]` – "the cost eventually becomes higher than 100 with probability at most 0.3"
- `P=? [ true U (cost > 100) ]` – "what is the probability that the cost eventually becomes higher than 100?"
- `S=? [ profit > cost ]` – "what is the long-run probability that the profit is higher than the cost?"

Note that it is possible to either determine whether a probability or expected quantity satisfies a given bound, or obtain the actual value. However, probabilities for an MDP can only be computed once the non-determinism has been resolved. Hence, PRISM actually computes either the minimum or maximum probability of a path property being satisfied, quantifying over all possible resolutions, i.e. the best and worst cases. Therefore, for MDPs there are two possible types of properties `Pmin` and `Pmax` (see also section 5.1.2 for an example).

As we have already seen, PRISM models can be augmented with the assignment of rewards to states and transitions. The tool can analyse properties which regard the expected values of these rewards, by the use of the `R` operator. Cumulative reward properties refer to the reward accumulated along a path, by summing the state rewards for each state and the transition rewards for each transition between these states. The following is an example of a reachability

reward property:
`R=? [ F profit > 100 {cost < 100} ]` – "what is the expected time taken to reach a state where the profit is higher than 100, starting from a state where the cost is less than 100?".

4.4 Efficient model storage and manipulation techniques

The reason that model checking has been so successful in the real world is that an enormous amount of work has been put into developing space and time efficient techniques for the storage and analysis of models, by incorporating state-of-the-art data structures and algorithms. The principal challenge one has to wrestle with, when developing any kind of model checker, is the notorious state space explosion problem, i.e. the tendency for the number of states in a model to grow exponentially as the size of the system being represented is increasing linearly. In practise, this means that models of even the most trivial real-world systems can end up having many millions of states, as a result of combinatorial considerations. Moreover, the numerical computations that must be performed on these models during verification are often memory consuming and require considerable time to be accomplished. Below, we will focus on two approaches that aim at combating these problems, and on which the PRISM implementation relies: the symbolic and the explicit approach.

4.4.1 Symbolic approach

One of the most successful approaches to tackling the state explosion problem is the adoption of *symbolic* techniques, which rely on a data structure called a *Binary Decision Diagram* (BDD), that provides a graph-based representation of Boolean functions. A *BDD* is a directed acyclic graph with a unique initial node, where all terminal nodes are labelled with 0 or 1 and all non-terminal nodes, called *decision* nodes, are labelled with a Boolean variable. Each non-terminal node has exactly two outgoing edges, one dashed line and one solid. Starting from the root, a dashed line is taken whenever the value of the variable at the current node is 0; otherwise the solid line is traversed. The function value is the value of the terminal node that is reached. Figures 4.4 and 4.5 illustrate a simple example of a BDD representing the function $f(x, y) = xy$.

The suitability of BDDs and similar symbolic encodings pertains to the extremely compact representations that they can provide, even for very large mod-

els. Efficient manipulation algorithms then allow model checking to be performed on these space efficient BDD-based data structures, that avoid the storing of redundant information. This can be illustrated through an example. Considering the BDD in Figure 4.5, the representation can be optimised by having pointers to just one copy of 0-terminal and one copy of 1-terminal, resulting in the structure depicted in Figure 4.6. There is further scope for optimisation, leading from the observation that the right hand y is unnecessary, because we end up in the same node whether it is 0 or 1. Therefore, the structure can be further reduced to the one shown in Figure 4.7. By removing duplicate terminals and non-terminals, as well as structurally identical (isomorphic) subBDDs, we result in a reduced form of the BDD, which offers a more compact representation of the Boolean function $f(x, y) = xy$ than the truth table in Figure 4.4. If this is the case for such a simple example with only two variables, it becomes clear how efficient storage BDDs can provide for more complex systems.

x	y	xy
0	0	0
0	1	0
0	1	0
1	1	1

Figure 4.4: Truth Table.

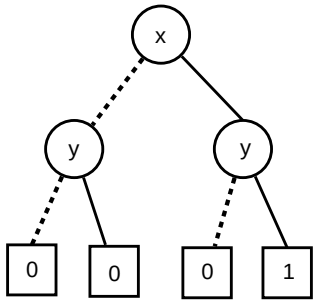

Figure 4.5: BDD representing $f(x, y) = xy$.

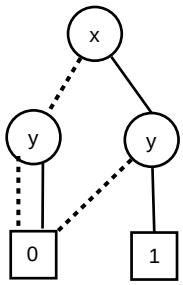

Figure 4.6: Sharing the terminal nodes of the BDD in Figure 4.5.

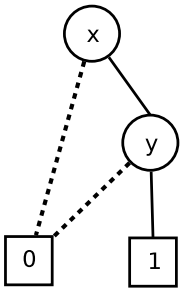

Figure 4.7: Removing a redundant decision node.

In order to exploit the advantages of symbolic storage in the case of probabilistic model checking, a natural extension of BDDs, *Multi-Terminal binary Decision Diagrams* (MTBDDs), is employed by PRISM. MTBDDs extend BDDs by representing functions which can take any value, rather than just 0 or 1, and

thus allow numerical computations required by probabilistic model checking. The adoption of MTBDDs enables the manipulation of models that contain up to 7 billion states, on which model checking would be infeasible if using conventional techniques. In the following section we will go through the basic theory behind MTBDDs in more depth, emphasising issues related to the operation of PRISM. The understanding of these rather specific theoretical issues is necessary for interpreting PRISM behaviour in chapters 6 and 7. The features of MTBDDs we will discuss in the followings also apply to BDDs, since the latter are a special case of the former.

4.4.1.1 Multi-Terminal Binary Decision Diagrams

An *MTBDD* M is formally defined as a directed acyclic graph, whose nodes represent a set of Boolean variables $\underset{\sim}{x} = \{x_1, ..., x_n\}$. It represents a real-valued function $f_M(x_1, ..., x_n) : \{0, 1\}^n \rightarrow \mathcal{D}$, where $\mathcal{D}$ is an arbitrary range. Here, we will consider that $\mathcal{D}$ is the set of real numbers $\mathbf{R}$. We define a *minterm* of an MTBDD as a valuation of its Boolean variables which results in a non-zero value. For clarity in the graphical representation, the terminal nodes with value 0 and any edges which lead directly to it are often omitted. Like the BDD in Figure 4.5, MTBDDs can be reduced to a more compact form, by applying a set of reduction rules to maximise sharing between the nodes of the graph. An MTBDD is said to be reduced if it contains no *shared* nodes, i.e. non-terminal nodes labelled with the same variable and with identical children, or terminal nodes labelled with the same value. Additionally, it contains no *redundant* nodes, i.e. non-terminal nodes having two identical children (we don't care about the variable of the node). Any MTBDD can be reduced by repeatedly eliminating, in a bottom-up fashion, any instances of shared and redundant nodes. It is important to note that the reduction rules that can be applied to MTBDDs have no effect on the function being represented, but do typically result in a significant decrease in the number of MTBDD nodes.

The application of the reduction rules can lead to distinct reduced MTBDDs, with different structure and number of nodes, that represent the same function f, and cannot be further optimised. In order to guarantee that the MTBDD representing a given function is unique, we have to impose an ordering on the variables occurring along any path through the MTBDD. Let $\underset{\sim}{x} = \{x_1, \ldots, x_n\}$ be a set of Boolean variables totally ordered as follows: $x_1 < x_2 < ... < x_n$. An MTBDD M over $\underset{\sim}{x}$ is said to be *ordered* if for every occurrence of x_i followed by x_j along any path in M, we have $x_i < x_j$. For a fixed ordering of Boolean variables, the reduced data structure can be shown to be *canonical*, meaning that

there is a one-to-one correspondence between MTBDDs and the functions they represent. In other words, if two reduced ordered MTBDDs M and M' represent the same function, then they have identical structure. Thereafter, when we refer to MTBDDs we will assume that they are ordered and reduced, unless stated otherwise. The canonicity property has crucial implications for efficiency. For example, it makes checking whether two MTBDDs represent the same function simply a matter of checking whether they have the same structure.

The choice of the variable ordering can have a tremendous effect on the size of the MTBDD for a given function. Consider the function $f = (x_1 \wedge y_1) \vee \ldots (x_n \wedge y_n)$. If we choose the ordering $x_1 < y_1 < x_2 < y_2 < \ldots < x_n < y_n$, then we can represent the function as an MTBDD with $2n+2$ nodes. If we opt instead for the ordering $x_1 < \ldots < x_n < y_1 < \ldots < y_n$ we derive an MTBDD that requires 2^{n+1}. The MTBDDs resulting from these two alternative orderings for $n = 3$ are shown in figures 4.8 and 4.9, which are taken again from the PRISM lecture slides. The sensitivity of the size of an MTBDD to the particular ordering is the price we pay for the advantage of achieving a canonical form. The problem is that finding the optimal ordering is known to be an NP-hard problem [BW96]. However, there are numerous heuristics that usually produce a fairly good ordering. The choice of an alternate ordering of variables as shown in Figure 4.8 is an example of such a heuristic, and will be discussed later in section 6.2.1 in more detail.

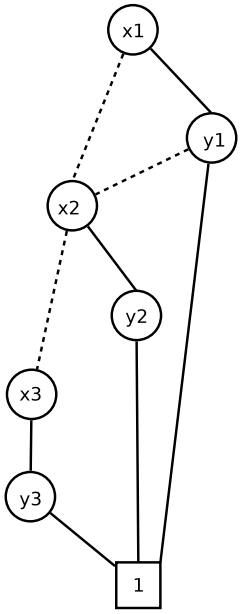

Figure 4.8: MTBDD with ordering $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$.

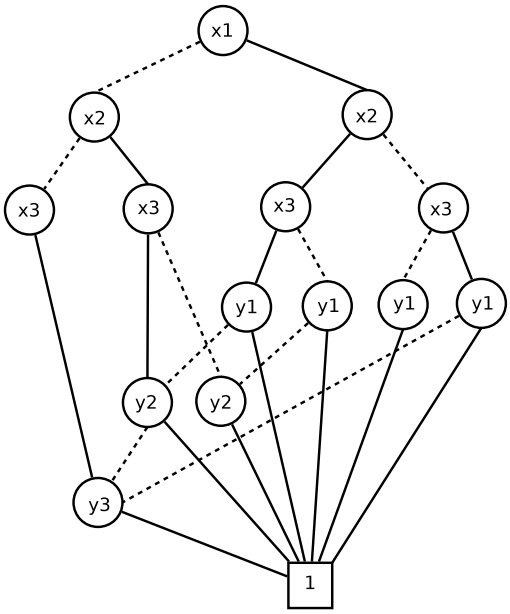

Figure 4.9: MTBDD with ordering $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$.

We will now show how MTBDDs can be used to represent a set of states

and transition relations. In section 4.2 we have seen how DTMCs, CTMCs and MDPs are expressed by real-valued matrices. Thus, our task is reduced to representing vectors and matrices as MTBDDs. Consider a real-valued $2^n \times 2^n$ matrix $\mathbf{M}$, representing a DTMC or CTMC. $\mathbf{M}$ can be interpreted as a mapping from pairs of integer indices to reals, i.e. $f_M : \{0, \ldots, 2^{n1}\} \times \{0, \ldots, 2^{n1}\} \rightarrow \mathbf{R}$. Given a suitable encoding of these integer indices into n Boolean variables, f_M can instead be expressed as a function of the form $f_M : \mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathbf{R}$. Hence, f_M can be represented as an MTBDD M over $2n$ variables, n of which encode row indices and n of which encode column indices. If the row position x is encoded by Boolean variables x_i and the column position y by Boolean variables y_i, where in both cases $i = 1, \ldots, n$, then the MTBDD M representing the function $f_M(x_1, \ldots, x_n, y_1, \ldots, y_n) = M(x, y)$ is a canonical representation of matrix $\mathbf{M}$. Recall that in the case of MDPs the transition matrix is non-square, since there can be multiple row entries for each state. Therefore, we have to consider one more set of Boolean variables corresponding to the non-deterministic choices. It should be noted that the encoding of matrices with sizes which are not powers of two is simply a matter of padding them with extra elements, rows or columns, all set to 0.

The process of representing matrices as MTBDDs can be better understood through an example, sourced from the PRISM lecture slides. Figure 4.11 shows the 4×4 transition rate matrix M for the 4 state CTMC depicted in Figure 4.10. The table in Figure 4.12 demonstrates how the matrix entries are represented by the MTBDD in Figure 4.13. Since the matrix is 4×4 we use 2 variables x_1 and x_2 to encode row indices and 2 variables y_1 and y_2 to encode the column indices. In both cases, the standard binary encoding for integers is used. Each minterm of the MTBDD represents a non-zero entry in the matrix. So, for the matrix entry $(0, 1)$, for example, $x_1 = 0$ and $x_2 = 0$ encode the row index 0 and $y_1 = 1$ and $y_2 = 0$ encode the column index 1. Observe also that we have chosen the interleaved ordering $x_1 < y_1 < x_2 < y_3$.

4.4.2 Explicit and hybrid approach

In contrast with the symbolic approach, explicit or enumerative techniques are those where the entire model is stored and manipulated explicitly. In the context of probabilistic models, *sparse* matrices are the most obvious and popular explicit storage method. The real-valued matrices that are used to describe probabilistic models are often very large but contain a relatively small number of non-zero entries (as confirmed by figures 4.3 and 4.11). There are several types of sparse storage schemes, all of which rely on the idea that only the non-zero entries of

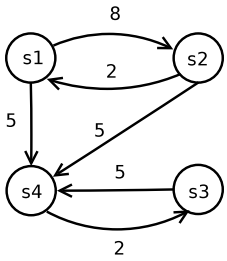

Figure 4.10: CTMC.

$$\mathbf{M} = \begin{pmatrix} 0 & 8 & 0 & 5 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 \end{pmatrix}$$

Figure 4.11: Matrix $\mathbf{M}$.

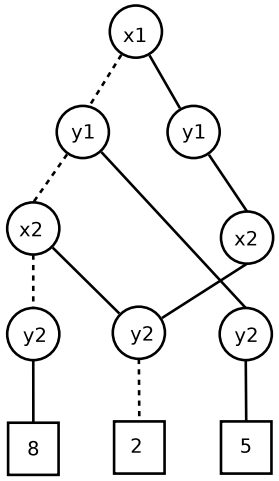

Figure 4.12: MTBDD M.

Entry in M	x_1	x_2	y_1	y_2	$x_1y_1x_2y_2$	f_M
(0,1)=8	0	0	0	1	0001	8
(1,0)=2	0	1	0	0	0010	2
(0,3)=5	0	1	1	1	0101	5
(1,3)=5	0	1	1	1	0111	5
(2,3)=5	1	0	1	1	1101	5
(3,2)=2	1	1	1	0	1110	2

Figure 4.13: Mapping table.

the matrix need to be explicitly stored.

Sparse storage schemes provide compact storage of matrices, proportional to their non-zero entries. Moreover, matrix entries can be accessed in a convenient and fast way. The main disadvantage of sparse formats is that they are not efficient for making modifications to the matrix, e.g. adding or deleting matrix entries. For large stochastic models, the memory requirements for the matrix storage can become unwieldy, and in many cases model checking fails to be accomplished. MTBDD-based data structures afford in these cases a successful alternative, since they can increase the size of models which can be handled, given the same hardware constraints, by some orders of magnitude. However, the symbolic implementation is usually outperformed by its sparse-based explicit counterpart in the case of smaller models. Experiments on a wide range of case-studies showed that MTBDD-based numerical computation, which is required for checking quantitative properties, performs poorly in comparison to explicit

alternatives, in terms of both time and space efficiency. It should be noted that computation based on reachability, which is sufficient for model checking qualitative properties, can be implemented efficiently with BDDs.

PRISM is a symbolic model checker, which means that the underlying data structures are BDDs and MTBDDs. For numerical computation, however, the tool provides three distinct engines. The first is a pure MTBDD-based implementation and the second a conventional explicit version using sparse matrices. The third engine is based an hybrid approach that will combine features of both the symbolic and explicit implementations, which aims at a compromise between the speed of sparse matrices and the size of the model that can be tackled. For typical examples, this technique is many times faster than using MTBDDs alone and can almost match the speed of the sparse matrix based implementation. At the same time, it provides a significant reduction in the amount of memory required for verication, and thus increases the size of model on which model checking can be successfully performed.

4.5 Symmetry reduction in PRISM

Symmetry reduction techniques can be effective at combating the state space explosion problem for model checking. Replication in the structure of a well-defined model can give rise to symmetries, or automorphisms. A permutation $\pi : S \rightarrow S$ on the state space is called an automorphism when it preserves the transition relation R, i.e. if $(s, s') \in R$, then $(\pi(s), \pi(s')) \in R$. A group of automorphisms gives rise to a partition of the states of a structure into equivalent classes, or orbits. Given a group G of automorphisms, we can choose a set $\bar{S}$ containing a unique representative state for each orbit. Then we can define a function $rep : S \rightarrow \bar{S}$ which selects the corresponding unique representative $rep(s) \in S$ for each state $s \in S$, and use this to induce a new transition relation $\bar{R} = \{(rep(s), rep(s'))|(s, s') \in R\}$. Since all permutations in G preserve the transition relation R, this quotient transition system $(\bar{S}, \bar{R})$ is bisimilar to the original transition system (S, R). Any representative function which maps all elements of an equivalence class on to the same unique representative can be used. The most common type of automorphism is component symmetry, in which any pair from a set of symmetric components in a model can be exchanged with no effect on the overall behaviour. Both symmetry reduction approaches supported by PRISM address the case of component symmetry. Thereafter, when we refer to symmetry we will mean component symmetry, unless stated otherwise.

To better understand the essence of component symmetry let us consider a

simplified example: an MDP representing two symmetric process-components which run in parallel. Suppose we are interested in the values of two variables $x1, x2 \in [0..1]$. The first process is responsible for manipulating variable $x1$, and the second for manipulating $x2$. Each state is indicated by the range of the corresponding digit. Figure 4.14 shows the MDP in question. The states are denoted by nodes labelled $i|j$, where i and j represent the state of process 1 and 2, respectively. Initially, both processes are in state 0..1. Each of the possible probability distributions from a state is denoted by a set of probability-labelled transitions (arrows), grouped by an arc. For simplicity, we assume that for each process there is only one probabilistic transition available at each step (this is analogous to having only one digitwise operation available). Figure 4.15 gives the function *rep* which maps each state from the MDP to a unique symmetric representative, and Figure 4.16 shows the emerging quotient model over the reduced state space.

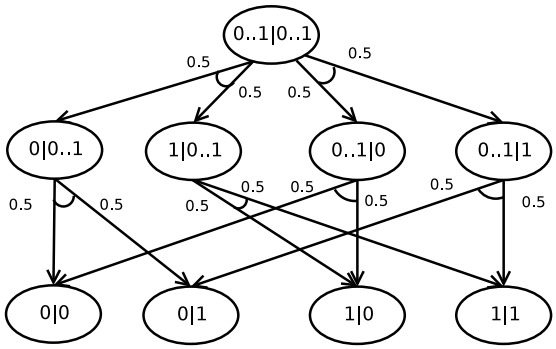

Figure 4.14: Full MDP.

$rep : S \to \bar{S}$
$0\|0..1 \to 0\|0..1$
$1\|0..1 \to 1\|0..1$
$0..1\|0 \to 0\|0..1$
$0..1\|1 \to 1\|0..1$
$0\|1 \to 0\|1$
$0\|0 \to 0\|0$
$1\|0 \to 0\|1$
$1\|1 \to 1\|1$

Figure 4.15: Representative function.

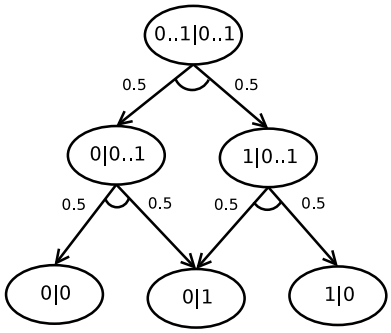

Figure 4.16: Quotient reduced MDP.

Clearly, it would be desirable to combine symmetry reduction with the use of symbolic data structures such as MTBDDs, which have proven very successful for improving the efficiency of model checking. The symmetry reduction method

described in [KNP06], PRISM-symm[3], considers the problem of constructing for a DTMC, CTMC or MDP exhibiting component symmetry the corresponding quotient model, on which probabilistic model checking can instead be performed. The approach builds upon the symbolic model checking framework already implemented in PRISM. In terms of the PRISM language, the component processes are expressed as modules. The algorithm first constructs a symbolic representation for the full model and then reduces it to the quotient model by computing a unique representative for any state it is possible. The unique representative function *rep* in this case simply sorts the values of symmetric components, according to some kind of ordering. The MTBDD corresponding to the quotient model is then computed by appropriately permuting and summing the elements of the transition matrix. The experimental results presented for four large case-studies demonstrate that this approach can offer not only a significant increase in the size of probabilistic models that can be successfully verified, but also a considerable improvement in the corresponding overall runtimes. It should be mentioned that no automated procedure of detecting and exploiting symmetry is supported by the current version of the tool, and thus symmetry has to be identified manually.

An alternative approach to symmetry reduction for systems exhibiting full component symmetry, which is also amenable to the existing symbolic model checking techniques integrated in PRISM, is introduced in [DM06] and relies on the use of *generic representatives*, or *counter abstraction*. Here, the *rep* function is obtained by counting the number of process-components in each possible local state, ignoring the individual component identities. The proposed method, GRIP, involves translating a fully symmetric source program into a reduced program with an isomorphic state space. A state in the translated program indicates how many processes are in each local state. The basic limitations of the GRIP version presented in [DM06] are that it only applies to fully symmetric systems, and that it requires a restrictive input language, which doesn't allow multiple local variables in modules, multiple module types or communication by synchronisation. Recently, in June 2007, new features have been added to GRIP[4] [DMP07], which overcome some of the previously mentioned lacunae and yield better results for many case-studies.

An important distinction between GRIP and PRISM-symm, as demonstrated by the results of a series of experimental comparisons between simple PRISM, PRISM-symm and GRIP on five case-studies (3 MDP models and 2 CTMCs) presented in [DMP07], is that PRISM-symm performs better than GRIP for

[3]Available at www.prismmodelchecker.org/download.php/symm

[4]The new version of GRIP is available at www.prismmodelchecker.org/download.php/grip

models that include a large number of processes with a relatively small number of local states. If the processes, however, contain many local states, the exponential blow-up in the number of variables required by counter abstraction makes GRIP very inefficient. On the other hand, an advantage of GRIP is that, unlike PRISM-symm, there is no need to first construct the full unreduced model, and thus the resulting MTBDD size for the symmetric model is smaller. However, the experimental results presented in [DMP07] demonstrate that this reduction in MTBDD size is not translated in any noticeable improvement in runtime, and for one of the models GRIP requires much longer time. Another limitation that characterises both tools is that they can only support one block of symmetric modules. This means that it is not possible to state in a single model e.g. that process 1 is symmetric to process 2, and at the same time that process 3 is symmetric to process 4.

4.6 Summary

In the first section of this chapter we gave a general overview of the PRISM probabilistic model checker. In section 4.2 we have introduced the stochastic models which can be manipulated by PRISM, and in section 4.3.1 we have shown how these models can be described in PRISM language and how the properties we want to model-check on them can be expressed. Section 4.4.1.1 presented the techniques used by PRISM for model construction, with emphasis on the MTBDDs that constitute the core data structure of PRISM. The final section introduced the notion of component symmetry and the two tools incorporated in PRISM for reducing this kind of symmetry.

Chapter 5

Formal analysis of PIN block attacks

While designers traditionally try to achieve security through systematic application of rules of thumb for fulfilling a set of assumptions, this process is usually convoluted and laborious. We have seen that formal tools can provide considerable aid to the human analysis of security APIs, by making it less engineered-driven and more automated. In this chapter we describe such a formal system, called AnaBlock, on which our work has built on. In the first section, we will discuss the main goals of the system, describe in detail how it works, and present the results of experiments on a series of API configurations. All this information has been sourced from Steel's paper [Ste06]. In the second section, we will see how certain limitations of the original AnaBlock framework indicate the directions of our work, and present the main facilities we used for the evaluation of the modifications we have attempted, which are discussed in the next chapters.

5.1 The AnaBlock system

We have seen that PIN block attacks take advantage of certain API commands, which can provide hints to the attacker about the values of the PIN digits. These attacks are especially hard to thwart, even when the strategy behind them is known in advance. The difficulty of their prevention relies mainly on the fact that many common commands that may be exploited by an attacker are associated with functions that are essential for the convenient operability of the HSM. At first glance, the functions that are potential sources of leaked information, like the decimal digit integrity checks or the check value function, seem harmless and are indeed useful, since they can detect unintended errors made by a benign user. However, we have already seen that the repetitional call of these functions

may reveal information that can help the attacker to reason about the PIN value. Hence, API designers are confronted with two opposing goals: the regulation of information flow and leakage on the one hand, and the maintenance of convenience in use on the other. The fulfilment of these two tasks becomes even more complicated if we consider that HSM APIs are typically designed for flexibility, so that the development cost associated with individual customisation can be limited by allowing the construction of many different configurations from a given generic set of API calls. Thus, each customer usually enables a different list of commands, uses other parameters and settings or extends the functionalities of the HSM's API by developing its own calls. For each of these configurations, we want to know if the specification resulting from the choices of the customer is secure, since the addition of a single, innocent-looking option, such as the check value computation, may entail a significant compromise to the API's security.

The AnaBlock system is a formal framework that can assist API designers in their decision making about what functionalities to support in the security APIs, ensuring at the same time that the risk of PIN cracking attacks, which may be enabled by these functionalities, doesn't exceed a certain bound, under which the API is considered to be secure. The motivation for creating the system came about from discussions Steel had with API designers at nCipher plc, a well-known manufacturer of HSMs. The API designers indicated the need for a formal tool, that, given an API specification and a particular set of options set by the customer, would be able to determine the expected number of steps required by an attacker to recover the PIN using the best of the known PIN block attacks available. Such a tool would be very helpful for experimenting with different configurations, allowing API designers to add and remove commands and vary several parameters, such as key settings and PIN block formats, while being aware of their effects on the security degree of the API. This is particularly difficult to be achieved by manual analysis, because of the complexity and fine granularity of API calls, each of which can potentially encapsulate many atomic parametrised operations, that may be responsible for a dangerous information leakage. It is also important that the tool is simple enough to allow changes to be made in a convenient way as new types of attacks are discovered. AnaBlock is a prototype of a system whose aim is to meet these requirements prescribed by the API designers. It should be underlined that it only deals with the three families of PIN block attacks described in 2.2, and can't guarantee that an unknown attack exists, which can compromise the security of the API.

The AnaBlock system comprises two stages:

- **PIN cracking model generation**: A program written in SICStus PRO-

LOG[1], which uses constraint logic programming, generates a model in PRISM syntax for all possible PIN block attacks, given a specific API configuration. This model assumes a uniform distribution of PINs and is represented as an MDP.

- **Probabilistic model checking**: The model of available PIN block attacks is given as input to PRISM, which returns the expected number of steps required to determine the PIN, if the optimal strategy is followed by the attacker. If full recovery of the PIN is not possible, then PRISM returns the probability of the most effective attack to reduce the range of PIN values to a particular size.

At a third stage, a script can be used to post-process the state, transition and reward matrices generated by PRISM, and produce the best attack step by step, i.e. find the exact set of operations-transitions the intruder has to apply (i.e find the shortest Markov Chain).

5.1.1 Model generation

The PROLOG program takes two inputs: the configuration file, chosen by the customer, and the API file, provided by the API designer. The configuration file specifies the exact list of commands a customer intends to enable in the HSM's API, including some options declared during installation. The API file defines the set of rules which specify which operations are available to an intruder when particular commands and options are supported by the configuration. The rules can be easily modified by the API designer to cover new variations of the three families of PIN block attacks.

The model generated by PROLOG is represented as a state transition system. The states can be seen as tuples consisting of 7 variables, which reflect the current situation for the intruder. The first elements of the tuple P_1, P_2, P_3, P_4 are constrained integers in the range [0..9] and convey the knowledge of the attacker about the values of the corresponding PIN digits. In the initial state, these four variables are constrained to [0..9], while their range decreases monotonically as the intruder makes successive calls to API commands and reasons about the PIN. It should be noted that the expression and manipulation of the constrained ranges of PIN digits is allowed thanks to the constraint solver embedded in SICStus PROLOG (see [JM87] about the theoretical foundations of the scheme, [COC97] for the particular implementation clp(FD)), which greatly facilitates this task.

[1] www.sics.se/sicstus/

The rest of the elements in the tuple record information for the decimalisation table attack and their meaning will be described later in section 5.1.1.2.

The transitions correspond to state changes that are due to both probabilistic and non-deterministic choices. For each state that is not an endpoint, there are n sets of outgoing transitions. Each set represents a particular decision made by the attacker about his next move, and consists of one or two transitions. Obviously, if $n = 1$, there is only one option for the intruder. A set with two transitions represents the intruder calling an API command: the one transition corresponds to the case where the HSM signals an error, and the other corresponds to the case where the command is executed successfully. Each transition is associated with a probability, which is calculated under the assumption that PIN values are distributed uniformly. These two probabilities sum to 1. The computation of the transition probabilities is based on the range of possible PIN values reasoned by the attacker after an HSM command call (see section 5.1.1.1). This way it is ensured that the calculated possibility will be correct, independently of what particular operation was performed. This is an important property of AnaBlock, because it allows commands to be combined in any order, and thus the analysis of different configurations does not require any changes to the Anablock system. A set with one transition represents a brute-force guessing, and the associated probability is 1. All probabilities assigned to transitions depend only on the current state, i.e. the model satisfies the Markov Chain property, and thus it is an MDP. Transitions are also weighed with a cost, which indicates how many calls of API commands are required to move from the current state to the next state. A transition corresponds to a single API call, except for the case of brute-force guessing, whose cost depends on the range of possible PIN values represented by the current state (see 5.1.1.3).

The above described MDP for the possible PIN block attacks must be expressed in terms of the PRISM language. Because PRISM can not explicitly handle constraints, the four PIN digit ranges are represented by a set of 40 Boolean variables `P1_could_be0`, ..., `P1_could_be9`, ..., `P4_could_be0`, ..., `P4_could_be9`. Thus, a value assignment where `P3_could_be0` and `P3_could_be1` are false, while the other 38 variables are true, corresponds to the situation where $P_1, P_2, P_4 \in [0..9]$ and $P3 \in [2..9]$. An additional variable `PIN_Possibilities` is used to hold the number of possible PINs represented by a particular state. Four more Boolean variables `P1_guessed`, ...,`P4_guessed` are added, which are set to true when the corresponding PIN digit has been determined, or when brute force guessing is used to guess all digits. As we will see later, the `PIN_Possibilities` and the `Pi_guessed` variables are necessary to express the properties we want to

check on the model, e.g. the probability of the optimal available attack to reduce the value of `PIN_Possibilities` under a certain bound. It should finally be noted that all possible state transitions are described in a single PRISM module. The costs of the transitions are specified using the reward mechanism supported by PRISM.

The sequence of steps prescribed by each family of PIN block attacks is defined recursively by the `determine` predicate in the PROLOG code, which takes as arguments the 7 variables that define a state in the MDP. The clauses of the `determine` predicate are called according to the rules included in the API file, using the meta-programming mechanism provided by PROLOG, which allows from within a program the writing or manipulation of data structures that represent programs. Meta-programming in PROLOG exploits the fact that terms and predicates have identical syntactic structure, which entails that predicates can be used as arguments of other predicates. Each clause of `determine` corresponds to an API command, with particular values for its input parameters. A suitable call of the `findall` predicate over `determine` leads to the generation of all possible attacks. However, the meta-programming of `determine` is done in such a way, so that an expedient overall strategy results, by performing first the less expensive attacks, i.e. the ones that can lead quicker to the determination of PIN digits. Thus the following pattern is followed:

1. Perform the allowed digitwise operations prescribed by the attacks of the ISO-0 family, until no more such operations are available. If the PIN has been fully determined, stop. Otherwise continue to 2.
2. Perform the allowed whole-PIN operations prescribed by the decimalisation table attacks, until no more such operations are available. If the PIN has been fully determined, stop. Otherwise continue to 3.
3. Perform the allowed brute-force guessing steps.

Since the order of execution of the digitwise attacks doesn't matter, first all digitwise operations on the first digit of the PIN are tried, then on the second, then on the third and then on the fourth. This way, the bunch of symmetric states resulting when allowing all possible orders of execution is removed. For completeness, we include in the following the detailed algorithms for the three clauses of the `determine` predicate, each one corresponding to a family of PIN block attacks, as described by Steel in [Ste06].

5.1.1.1 Digitwise attacks

A `determine` clause for a digitwise operation takes as arguments the variables that define the current state, and produces two probabilistic transitions: the

one corresponds to the HSM command reporting an error, and the other to successful execution. Both transitions lead to states where the intruder has gained new knowledge about the PIN. For each of these two new states the `determine` predicate is called recursively. The detailed algorithm of executing the digitwise commands given in [Ste06, §3.1] runs like this:

1. **If** this command has already been applied on the same digit in this state
   stop.
2. Calculate the overlap between the range of PIN values accepted by this command, and the current range of possible PIN values. This is called `Accept_Range` and represents the next state, if the command has returned successfully.
3. Calculate the overlap between the range of PIN values rejected by this command, and the current range of possible PIN values. This is called `Error_Range` and represents the next state, if the command has returned an error.
4. **If** `Accept_Range`$= \emptyset$ or `Error_Range`$= \emptyset$.
   stop (the application of this command, doesn't tell us anything useful).
5. Count the total number of possible PINs that satisfy `Accept_Range`. This is called `ASize`.
6. Count the total number of possible PINs that satisfy `Error_Range`. This is called `ESize`.
7. Write output in PRISM syntax: the probability of moving into a state representing `Accept_Range` is `ASize`$/($`ASize`$+$`ESize`$)$, and the probability of moving into a state representing `Error_Range` is `ESize`$/($`ASize`$+$`ESize`$)$.
8. Recursively call `determine` on `Accept_Range`.
9. Recursively call `determine` on `Error_Range`.

5.1.1.2 Decimalisation table attacks

As mentioned above, the first 4 arguments of the determine predicate are the constrained integers representing the PIN digits. The `determine` clauses taking care of the decimalisation table attacks also make use of the last 3 arguments. The 5th argument is an integer called `Last_Dectab`, which indicates the last dectab tried against the PIN. A value i means that the last dectab tried was the standard table modified by adding 1 to all decimal elements i. The 6th argument is the integer `Last_Dectab_Hit`, which records the value of the dectab last time the PIN failed to verify, i.e. when a dectab hit occurred. The 7th is the integer `Last_Offset`, which records the last offset value which was tried.

The decimalisation table attacks are modelled by two `determine` clauses, one responsible for trying the next modified dectab, and one for stepping through the offsets. The first clause succeeds only when the value of `Last_Dectab_Hit` is greater than `Last_Dectab`, in which case the dectab is advanced and two new states are produced, one for a a dectab hit and one for a miss. `determine` is recursively called on these states, with the appropriate updated values for `Last_Dectab_Hit` and `Last_Dectab`. The second clause succeeds only when `Last_Dectab_Hit` is equal to `Last_Dectab`, i.e. when a dectab hit has just occurred. In this case the offset is increased by one and two new states are again produced, one for an offset hit and one for a miss. Again a recursive call to `determine` is made with the arguments corresponding to `Last Dectab Hit`, `Last_Dectab` and `Last_Offset` set to the proper values. In the following we present the algorithm that describes in detail how the dectab attack is executed: (taken from [Ste06, §3.2]):

1. Start with `Last_Dectab=-1`, `Last_Dectab_Hit=11`, `Last_Offset=0`.
2. **If** all PIN digits are determined
   stop
3. **If** `Last_Dectab` > `Last_Dectab_Hit`
   Increase `Last_Dectab` by 1
   If the dectab hits
   set `Last_Dectab_Hit` to `Last_Dectab`
   goto 4
   else adjust digit constraints and goto 2
4. Increase offset to next suitable value
5. **If** offset hits
   set `Last_Offset` to 0
   set the digits hit by the offset to `Last_Dectab`
   set `Last_Dectab_Hit` to 11
   goto 2
6. Goto 4

It is also worth mentioning how the use of the propositional constraint mechanism of PROLOG allows the convenient expression of the newly created states, yielding a concise and elegant code. Suppose, for example, that we are at a state where $P_1 \in [0..9] \wedge P_2 \in [0..9] \wedge P_3 \in [0..2] \wedge P_4 \in [0..2]$. and a dectab miss occurs, with the decimalisation table modified at position 5. All we have to do, is to remove value 5 from the the range of possible values for each digit, resulting in a state where $P_1 \in ([0..4] \vee [6..9]) \wedge P_2 \in ([0..4] \vee [6..9]) \wedge P_3 \in [0..2] \wedge P_4 \in [0..2]$. The

idea behind the adjustment of constraints in the other cases of hits and misses is the same.

5.1.1.3 Brute-force attacks

The `determine` clause modelling the guessing attacks is added as a last option, and succeeds only if guessing is allowed by the API specification. It produces a single transition with probability 1, that leads to a state where all PIN digits have been guessed (an endpoint). It should be noted that the actual values of $P1, P2, P3, P4$ are not set, but only the four Boolean variables `P1_guessed`, ...,`P4_guessed` are updated to true. The `determine` clause responsible for the guessing steps is called once for each of the intermediately "final" states generated by the previous attacks, having each time as arguments the variables that determine this "final" state. The cost associated with the transition from the penultimate to the final state depends on the range of the PIN values at the current state and the specific kind of brute force guessing attack, according to what operations are available. Thus, the formula that calculates the cost is specified in the API file. For the brute-force attack described in 2.2.3, the cost assigned to a transition outgoing from a state representing n possible PIN values will be $n/2 + 1$, where the 1 corresponds to the single call to the check value function, and $n/2$ are on average the required calls to the verification function.

5.1.2 Model checking

Once the model representing all the options available to the attacker is generated, it is passed to PRISM, together with a file containing the specifications of the properties we want to verify or evaluate. First, we would like to check if the attacks allowed by a configuration can lead to full recovery of the PIN. This is achieved by asking PRISM about the probability of eventually arriving at a state where all digits are known. In PRISM property specification language, this is expressed like this (the meaning of the operators has already been explained in section 4.3.2):

$$\texttt{Pmax} =? \left[\texttt{true}\, \texttt{U}\, (\texttt{P1\_guessed} \wedge \texttt{P2\_guessed} \wedge \texttt{P3\_guessed} \wedge \texttt{P4\_guessed}) \right]$$

Because the model is an MDP, there may be multiple paths that lead to the satisfaction of the desired property. Thus, PRISM can only compute the probabilities once the non-determinism has been resolved. Since we are interested in the worst case scenario, we request the maximum probability $Pmax$ of the property being verified, after PRISM has quantified over all possible resolutions.

For models where the attacks manage to determine the PIN with probability 1, the property we are interested in, is the expected number of operations required. For this purpose the following property is model checked:

$$\texttt{Rmin} =? [\, \texttt{F}\, (\texttt{P1\_guessed} \wedge \texttt{P2\_guessed} \wedge \texttt{P3\_guessed} \wedge \texttt{P4\_guessed})\, \{\texttt{"init"}\}]$$

As we have explained in section 4.3.2, the reachability reward property $\texttt{F}\,(\texttt{prop})$ corresponds to the reward accumulated along a path until a state satisfying property `prop` is reached. In this context, the `F` operator specifies that the property must hold at some future state, starting from the initial state. By setting $\texttt{Rmin} =?$, we are requesting the minimum value from the set of expected rewards corresponding to the different non-deterministic paths, i.e. we are looking for the resolution that corresponds to the best case for the intruder. For the configurations that don't allow attacks that can determine the PIN exactly, we are interested in the attack with the highest probability of reducing the range of PINs to some particular size `k`. This can be obtained by model checking the property:

$$\texttt{Pmax} =? [\, \texttt{true}\, \texttt{U}\, \texttt{PIN\_Possibilities} <= \texttt{k}\,]$$

5.1.3 Experimental results

To evaluate the AnaBlock system Steel has conducted a series of experiments, all based on a generic API following the command set presented in [Clu03, §3.3]. All the experiments were conducted using a 2.11ish version of PRISM. The configuration files include 13 settings (6 commands, 4 block formats, and 3 extra options for locking-down the PAN, dectab and offset). For the purposes of evaluation, seven typical congurations were used, selected in such a way as to allow different kinds of attack. The basic characteristics of each configuration are briefly the following[2]:

(1) All options are enabled, thus this is the most insecure configuration.

(2) The PAN is locked down, thus no digitwise operations can be performed.

(3) Does not support the VISA-3 block format, thus no masquerading is allowed.

(4) The translation function is blocked, but IBM 3624 PIN verification is allowed. This scheme for the verification function allows PIN calculations

[2]For the exact set of options enabled by each configuration see http://homepages.inf.ed.ac.uk/gsteel/anablock/anablock-0.11/examples/

that are based on validation data supplied by the user, e.g. an ASCII encoding of the PAN digits, rather than on the PAN itself. The check value function is also supported.

(5) No VISA-3 format is supported and the dectab is locked down. The check value function is not enabled, neither clear PIN encryption is allowed in the verification box, thus brute force guessing attacks are also prevented.

(6) The offset is locked down, the translation function is not supported, neither IBM 3624 PIN verification is allowed. Thus no digitwise operations can be performed.

(7) The translation function is enabled and the PAN is allowed to vary, but IBM 3624 PIN verication is not supported, and neither is the VISA 3 format. The offset is locked down.

Table 5.1 summarises the results of the experiments on the configurations for which full recovery of the PIN is possible. We can see for each configuration what the most effective attack strategy is, and the expected numbers of steps $E(steps)$ required by this strategy to determine the PIN. As expected, for configuration (1) the attacker can recover the PIN in the least number of steps. The highest number of steps is required for attacking configuration (4). That's because the attack tactic in this case relies highly in brute-force guessing, which is very expensive to perform. Table 5.2 shows the results for the configurations on which the available attacks cannot determine the value of the PIN uniquely. In this case, we are interested in the probability of these attacks to reduce the PIN range to k possible values. We can see from the probabilities corresponding to each value of k that configuration (7) is particularly vulnerable, even if it has the offset locked down and does not allow the full ISO-0 attack.

Configuration	Attack	$E(steps)$
(1)	ISO-0 (full)	13.6
(2)	Dectab (full)	16.145
(3)	ISO-0 restricted & Dectab (full)	15.275
(4)	ISO-0 / IBM 3624 PIN & Check Value	57.8

Table 5.1: Best attack and expected number of steps for the configurations that allow full determination of the PIN.

It should be highlighted that the attack to configuration 4 is a novel variation of attack, that had not been realised before. The original IBM 3624 algorithm

Configuration	Attack	k=400	k=36	k=24	k=14	k=1
(5)	ISO-0 restricted	1	0	0	0	0
(6)	Dectab no offset	1	1	0.568	0.064	0.001
(7)	Dectab no offset & ISO-0 restricted	1	1	1	1	0.001

Table 5.2: Best attack and probability of narrowing down the possible PIN values to k for the configurations that don't allow full determination of the PIN.

offers the intruder the ability to vary not only PANs as input to the verification function, but also supply customised validation data. By exploiting this potential, the intruder can follow the following strategy: first XOR 1 against A_1, then increase by 1 the third digit of the offset, and repeat the call to the PIN verification command. If the verify call succeeds, then he knows that the first bit of P_3 is 1. If the call fails, he knows that the first bit of P_3 must be 0. The intruder can proceed by increasing the offset by 2, by 4, and then by 8, so that he can reason about all bits of P_3. Then he can repeat the whole procedure for P_4, thereby determining the last two digits of the PIN. After having determined the last two digits of the PIN by following this strategy, the attacker can guess the other two digits by brute-force, by exploiting the check value command.

5.2 Alternative approaches to representing PIN block attacks

One limitation of AnaBlock is that the produced MDP of all possible attacks is very large, and thus it takes a considerable time for the model checker to analyse it. As we will see in the following, the experimental results illustrate that for many configurations the system consumes a great amount of memory, and that the runtimes can be quite long. High memory and time requirements put restrictions on the applicability of Anablock. Any improvements in the efficiency of the system in respect to these requirements would obviously make it more convenient to use and more attractive to API designers, hopefully facilitating detailed analysis of more complex, and thus more time and memory consuming, financial APIs. Our objective is to investigate and implement alternative ways of representing the model for the three families of PIN block attacks, hopefully yielding more compact MTBDDs, and consequently reducing the corresponding runtime and memory usage. More precisely, we attempt three different approaches, which are presented in the three following chapters. The first one focuses on decreasing

the number of variables used for the representation of the model, the second investigates alternative ways of decomposing the model into multiple modules and exploiting the symmetric nature of digitwise operations, and the third addresses the duplication that characterises the full decimalisation attack.

To evaluate the representations that arise from these alternative approaches to modelling PIN block attacks, we conducted experiments based on the generic API used in [Ste06]. Besides the seven configurations that were used for the experiments with original AnaBlock, we also considered three additional configurations, to test some more combinations of attacks. It should be highlighted that we have used PRISM version 3.0[3] for all our experiments, while the runtimes reported in [Ste06] were measured using the 2.11 distribution of PRISM, which is currently not publicly available. To add new functionality to PRISM, version 3.0 implements model construction in a slightly different way, which may cause in certain cases an increase in the respective data structure size, and consequently in model construction and model checking times. The reverse is also true in some cases as well. Unfortunately, many of our models fall into the first category, and thus the respective times required for model construction are significantly longer than the ones reported in [Ste06]. However, this does not constitute any serious problem with respect to the evaluation of the system, since what we are interested in is the relative comparison between the original AnaBlock and the modified versions resulting from the different approaches we have adopted. The comparison between the two will be reliable as far as the measurements are taken using the same PRISM distribution and the same machine.

In Table 5.3 we present the runtimes for the experiments mentioned in section 5.1.3, when using the two PRISM versions in question. The measurements in the second and third column represent the PROLOG and PRISM 2.11 runtimes, respectively, as reported by Steel in [Ste06]. They were recorded on a 3.6GHz Pentium IV machine running Linux 2.6.12 and Sun JRE 1.5.0. The fourth column corresponds to the time required by PRISM version 3.0, recorded on the same machine. Regarding memory consumption, Steel only reports that the model requiring the most memory was that for experiment (3), which peaked at around 1.2Gb. According to our measurements, when using PRISM version 3.0, the maximum memory usage for configurations (1), (2) and (3) was at around 1.1Gb, for configuration (7) at around 400M, while the rest of the configurations required 70-150Mb. It should be noted that all the time and memory values reported in the next chapters, both for the original AnaBlock and the modified versions we have attempted, were recorded on a 1.8GHz Intel Core 2 computer running

[3]Available in www.prismmodelchecker.org/download.php

Configuration	Time		
	Anablock	PRISM 2.11	PRISM 3.0
(1)	50 min	38 min	1.1 h
(2)	71 sec	7 min	48 min
(3)	56 sec	11 min	54 min
(4)	3 sec	43 sec	46 sec
(5)	3 sec	14 sec	27 sec
(6)	10 sec	2 min	2 min
(7)	25 sec	16 min	13 min

Table 5.3: Runtimes for the experiments: the 2nd and 3d column correspond to the measurements presented in [Ste06], the 4th to the runtime required by PRISM version 3.0 on the same machine.

Linux 2.6 and Sun JRE 1.6.0, which gave approximately the same measurements as the previously mentioned machine. All experiments were conducted using the MTBDD computation engine for model checking, since reachability reward properties for MDPs can at the moment be analysed only by this engine.

At this point, we would also like to make some rather technical remarks about PRISM memory usage. As we have already mentioned (see 4.1), the PRISM process consists of two coarse phases: model construction, during which the PRISM language description is converted to an MTBDD, and model checking, during which the MTBDD is analysed in order to identify the set of states which satisfy a desired formula and/or compute probabilities. Both phases may consume considerable amounts of memory. As remarked on the PRISM site[4], it is common that PRISM becomes very slow or crashes because of excessive memory requirements. The principal cause of memory usage during the model construction phase is the underlying (MT)BDD package CUDD, which has an upper memory limit that can be set according to the physical resources available in the working machine. However, if the CUDD memory limit is increased too high just for model construction, then it is likely that there will be not enough memory for subsequent model checking operations, since the memory required for storing the model will be kept bound until all PRISM processes have finished. The memory consumption during the model checking phase depends highly on the computation engine (symbolic, sparse or hybrid) that is used. The size of the model under investigation, i.e. the number of its states and transitions, is a critical factor for the efficiency of performing probabilistic model checking on it. However, because of

[4]www.prismmodelchecker.org/manual/FrequentlyAskedQuestions/MemoryProblems

the symbolic techniques used by PRISM, the amount of memory required to store the probabilistic model can vary, sometimes unpredictably, according to several factors, such as the order in which the variables appear in the model file. Finally, it should be noted that sometimes model parsing, during which PRISM reads the model file and checks it for correctness, can also be resource intensive. A memory overload during model parsing can occur if the description of the model in the PRISM language is very large, as the models produced by AnaBlock are. In this case, the upper memory limit of the Java virtual machine (JVM) used to execute PRISM must be increased.

Unfortunately, the exact amount of memory used by CUDD or by the computation engine cannot be determined from within PRISM. So, the most accurate solution to the problem of measuring memory consumption is simply to analyse the total memory usage of the PRISM process over time, e.g. by using the "top" command. This is the approach we have followed in our experiments.

5.3 Summary

In the first section of this chapter we have presented the AnaBlock system. We have seen how this system can assist engineers at the design process of security APIs, and at assessing the vulnerability of a specific choice of API commands against the three families of PIN block attacks. AnaBlock takes as input the configuration of an API together with a set of rules, and by using constraint logic programming in PROLOG it generates an MDP that represents the group of possible attacks against the API configuration. This MDP is passed to PRISM, which performs model checking on it, extracting the optimal attack and evaluating a number of requested properties. We have described in detail the algorithms that specify the steps each family of attacks comprises, as well as the properties that are model checked. The AnaBlock system has been evaluated on seven configurations, based on a generic API. The experimental results have given some new insights into the available attacks and how they can be combined in a way convenient for the intruder. They also revealed a new variation of attack against one of the configurations under investigation. In the second section we have seen how the motivation of our work is prompted by some limitations of AnaBlock, arising from the large size of the generated models, whose analysis often requires long runtimes and memory usage. Finally, we have defined our area of focus, and made some important remarks about the tools we will use for our experiments and the evaluation methods we will follow.

Chapter 6

A 2-variable representation

The representation of PIN block attacks in the original AnaBlock model comprises 44 Boolean variables, 7 integers with small range (0..12 or less) and the integer variable `PIN_Possibilities` with range 0..10000. In order to get rid of the large number of variables that are used to identify each state –and are redundant as far as the construction of the MDP is concerned– we can go for a more succinct model that will represent the same state transition system by using fewer variables.

The alternative representation we propose consists of only two variables: the `PIN_Possibilities` variable and an integer variable `State_Id`, which uniquely represents each state. `State_Id` is a counter that counts the number of different states, and its range is identical to the state space size for each configuration. More precisely, we observe that each state is uniquely determined by an ordered list of variables [P1, P2, P3, P4, `Dectab, Dectab_Hit, Offset`], where Pi is the integer corresponding to the 10 bits that constitute the ith PIN digit, according to the standard binary encoding. Starting from 1, PROLOG increases `State_Id` by one for every new list that has not occurred before. This way, each list-state is mapped to the appropriate `State_Id`.

6.1 Motivation

We have opted for a more laconic representation setting out from the idea that it would be more convenient for the model checker to construct a model with only 2 variables than having to deal with 49 variables. This reduction in the number of variables does not entail any change in the size of the model itself, understood as the number of states it comprises, however it was expected that the 2-variable representation would be translated into a more compact MTBDD. As we have already seen in section 4.4.1.1, the size of MTBDDs has a major impact on the efficiency of a symbolic model checker, because it affects both

the amount of memory required for storage and the amount of time required for model construction and manipulation (the complexity of most operations that are performed on MTBDDs is polynomial in their size).

The encoding scheme used by PRISM for translating the high-level PRISM language description into a data structure ensures a close correspondence between PRISM and MTBDD variables. Each PRISM variable is encoded with its own set of Boolean MTBDD variables, following the standard binary representation. Considering configuration (3) for example, the range of the `State_Id` variable is about 1..21000, and thus 14 MTBDD Boolean variables are required to represent it. Taking into account the `PIN_Possibilities` variable, the total number of MTBDD variables in the case of the 2-variable representation becomes 27, while the original multiple-variable model requires 85 MTBDD variables (we don't consider the additional variables that are required to encode the non-deterministic choices). Hence, it sounds reasonable that PRISM would take benefit from the significantly lower number of variables, yielding a more compact storage of the model, and consequently lower memory and time requirements.

It is evident that, when using the 2-variable representation, we loose track of the exact value of the PIN at each state and can't ask PRISM about the likelihood of the PIN being of a particular value or range. However, what really matters is the ability to calculate the probability of eventually arriving at a state where all digits are known, as well as the number of steps that are required in the best case (for the intruder) to fully determine the PIN, if this is possible. To achieve this, we slightly change the properties we want to evaluate, which take the following form in PRISM syntax for PCTL:

```
Pmax=? [true U (PIN_Possibilities=1)]
Rmin=? [F (PIN_Possibilities=1){PIN_Possibilities=10000}]
```

6.2 Experimental results and evaluation

We ran PRISM for configurations (2), (3) and (5) using the 2-variable representation. Table 6.1 summarises the runtimes we obtained for the original AnaBlock representation and the runtimes resulting from the 2-variable representation. The total runtimes are also plotted in the histogram of Figure 6.1, to comparatively illustrate the time requirements of the two representations for each configuration.

As we can see, against our expectations, the results we obtained were particularly disappointing. For the heavy configurations (2) and (3), the overall PRISM runtimes were about 4 times higher in comparison to the original model. The time for model checking was somewhat less, but that doesn't make any differ-

Example	Time			
	AnaBlock	Model Construction	Model Checking	Total
(2)-original	1 min	49 min	2 min	52 min
(2)-2var-repr	2 min	4 h	7 sec	4 h
(3)-original	1 min	52 min	6 min	59 min
(3)-2var-repr	2 min	3.5 h	4 min	3.5 h
(5)-original	5 sec	27 sec	<1 sec	38 sec
(5)-2var-repr	11 sec	1 min	<1 sec	1 min

Table 6.1: Experiments with the 2-variable representation. (n)-original/2var-repr indicates the experiments done on configuration (n) using the original and the 2-variable representation model, respectively.

ence since the bottleneck is construction time. AnaBlock time is higher because of the additional computations that are needed to determine the right value of `State_Id`. It is also worth mentioning that memory usage by PRISM remained at the same levels as for the original AnaBlock version: about 1.1Gb maximum

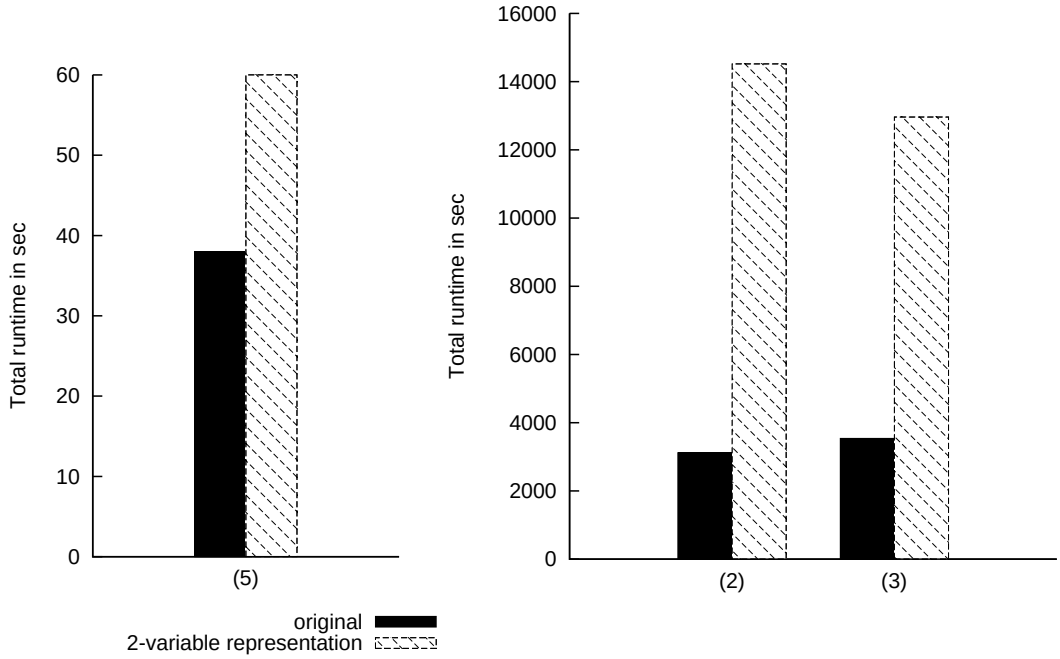

Figure 6.1: Total runtimes for each configuration.

for configurations (2) and (3), and about 70Mb for configuration (5).

The reasons for the increase in model construction time must be looked for in the way the core data structures of PRISM, MTBDDs, are built. The observation about the decrease in the number of MTBDD variables disregards another key issue that dramatically affects the size of the resulting MTBDD and the time required for its construction: the major impact the inherent structure of the model can have on the corresponding MTBDD size and the process of its construction. In the following section we discuss some rather specialised theoretical issues about the way the PRISM model is converted into an MTBDD, based on Parker's PhD thesis [Par02]. This theory is necessary to understand the reasons why the 2-variable representation led to longer model construction times, as well as for the interpretation of the results obtained by the approaches discussed in the subsequent chapter.

6.2.1 Factors that affect MTBDD construction

In [HMKS99] Hermanns et al. note that, in order to derive compact MTBDD encodings of probabilistic models, one should try to preserve structure and regularity from the high-level description of the system in its MTBDD encoding. The more regularity the low-level MTBDD representation exhibits, the higher the number of shared and redundant nodes (see 4.4.1.1), and subsequently the smaller the size of the data structure is. This can be illustrated by reconsidering the example of representing a matrix as an MTBDD in section 4.4.1.1, which we display again here in Figures 6.2 and 6.3. We see that the repeated submatrices (2, 0) give rise to a shared node, while the identical adjacent submatrices (0, 5) are responsible for the removal of a redundant node.

The correlation between PRISM and MTBDD variables ensures that the structural information of the high-level PRISM formalism is reflected in the corresponding MTBDD. The ordering of the variables is also a matter that pertains to the structure of the model. In section 4.4.1.1 we have mentioned that the ordering of the MTBDD variables is of major importance. Finding the optimal ordering is a computationally expensive problem, and hence one has to resort to heuristics. An example of such an ordering heuristic has already been presented: when building the MTBDD to represent the matrix in Figure 4.11, the MTBDD resulting from the interleaving of the Boolean variables contained fewer nodes than the one resulting from a non-interleaved ordering. Considering the Boolean vectors $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$, which encode the source and target states, respectively, the interleaving approach will opt for the ordering $(x_1, y_1, \ldots, x_n, y_n)$. In general, it has been shown ([HMKS99], [EFT93]) that the alternate variable or-

$$\mathbf{M} = \left(\begin{array}{cc|cc} 0 & 8 & 0 & 5 \\ \mathbf{2} & \mathbf{0} & 0 & 5 \\ \hline 0 & 0 & 0 & 5 \\ 0 & 0 & \mathbf{2} & \mathbf{0} \end{array}\right)$$

Figure 6.2: Matrix $\mathbf{M}$.

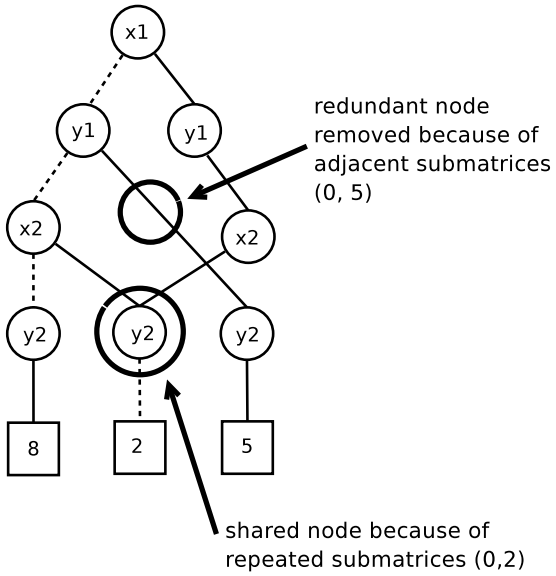

Figure 6.3: MTBDD M.

dering is beneficial for an MTBDD representing a transition matrix. This can be explained by the following reasoning: in a typical transition, only a few PRISM variables will actually change value, the rest remaining constant (such as happens, for example, in the case of the digitwise operations). Due to the direct correspondence between PRISM and MTBDD variables, this implies that each target-column variable y_i is most closely related to the corresponding source-row variable x_i. The MTBDD variables encoding a single PRISM variable are also considered to be closely associated with each other, and at a second level, the same applies for those referring to PRISM variables in the same module. In general, the heuristic used by PRISM imposes that MTBDD variables which are believed to be closely related are placed as near to each other as possible in the ordering.

In line with this heuristic, PRISM implements the construction of the final MTBDD that represents the entire module in a compositional manner. Instead of simply encoding a given, monolithic transition system as an MTBDD, the construction proceeds in steps, starting from the MTBDDs for the lowest level components, and using MTBDD-based parallel composition in every construction step. The superiority of this gradual construction is due to the fact that it exploits structure and regularity, and thus automatically yields good state encodings. According to PRISM's compositional approach, a separate MTBDD is first constructed for each guarded command in the model, which describes the local behaviour of a particular module for some subset of the global state space. Hence, it is represented by an MTBDD over the row variables for the entire system and

the column variables for that module. In terms of the PRISM language, the row and the column variables correspond to the non-primed and primed variables of a guarded command, respectively. The MTBDD describing the behaviour of an entire module is computed by summing those for all of its commands. The final MTBDD representing the whole model, either a DTMC or CTMC, is then obtained by combining the MTBDDs for all its modules. The procedure is roughly the same for the case of MDPs, except that some extra variables that encode the non-deterministic choices are introduced.

Besides minimising MTBDD size, the alternate ordering of the row and column variables has also the advantage that it accelerates the compositional construction of MTBDDs. A proper choice of representation and variable ordering can highly facilitate the convenient division of the transition matrix represented by the MTBDD and its expression as an algebraic composition of small-scale submatrices, corresponding to the components (commands and at a second stage modules) of the system being modelled. Regularity in the high-level formalism not only entails fast access to the partner submatrices, but can also be beneficial for implementing matrix operations, which can be expressed recursively, and are constantly used during the compositional construction of the final MTBDD.

6.2.2 Comparing the 2-variable with the multi-variable representation

Having understood the basic aspects of the MTBDD construction procedure, we can now explain why PRISM needs longer time in the case of the 2-variable representation. The key lesson learnt with respect to the above discussed issues is that maximising structure and regularity in a model is usually the best way to get a more compact MTBDD and a quicker construction process. The original multi-variable representation of the PIN-block attacks offers a high-level description that retains much of the inherent structure of the model, while the 2-variable representation distorts any sense of regularity. For example, let's consider the case of an XOR command that conveys knowledge about the range of digit P_3. In the original representation, a typical guarded command would be of the form (see section 4.3.1 for the meaning of the symbols):

```
[] P3_could_be0=true & P3_could_be1=true & P3_could_be2=true &
   P3_could_be3=true & P3_could_be4=false &...& P3_could_be9=false
-> 1/2: (P3_could_be0′=false) & (P3_could_be1′=false)
 + 1/2: (P3_could_be2′=false) & (P3_could_be3′=false)
```

The fact that only 2 of the variables change value after the transition is a feature that must be preserved in the low-level representation, so that it can be properly exploited by the interleaving ordering heuristic used by PRISM. When adhering to the 2-variable representation, the guard-source states and the update-destination states are indicated by arbitrary values, and consequently the correlation between the respective row and column MTBDD variables is lost. In general, because of the high number of variables that remain unchanged during typical transitions in the multi-variable case, the overlapping of the variable values between different PRISM states is more often. This fact can lead to a high reduction in the size of the ultimate composite data structure, due to the occurrence of repeated submatrices that can be exploited in a similar way as in Figure 6.3. Thus, even if the number of variables is higher, the expected increase in the size of the final MTBDD, in comparison with the 2-variable representation, is counterbalanced by the removal of a great amount of redundant nodes.

Equally important is the effect of each representation on the decomposition of the transition matrix into submatrices. Recall that, at a first stage, a separate MTBDD is constructed for each individual command. In the 2-variable version, these small MTBDDs will involve all of the MTBDD Boolean variables that encode the 2 PRISM variables, while in the multi-variable case the component MTBDDs will only contain a subset of the variable space (e.g. in the above digitwise command the corresponding MTBDD would only include 10 variables), and will therefore be more efficient to construct. It should be underlined that this fact does not affect the size of the final data structure, however it significantly reduces the time required for the construction of each individual MTBDD. Thus, the larger the model is, i.e. the more guarded commands its PRISM representation includes, the more significant the impact of this time factor becomes.

6.3 Summary

In this chapter, we have attempted an alternative representation of the original Anablock model, by keeping only 2 out of the 52 PRISM variables used in the original model description. Though we would expect that the reduction in the number of variables would yield smaller MTBDDs, which would thus be quicker to construct, the experimental results demonstrated a considerable increase in the corresponding model construction runtimes. After having a closer insight to the way MTBDDs are built in PRISM, we concluded that it is more profitable to adhere to a more comprehensive description, that is closer to the high-level

abstraction of the model, even if this implies that more variables must be used. A representation that reflects the inherent structure of the model will beneficial from the heuristics used by PRISM, as will the compositional process of constructing the final MTBDD. This is because it is more likely that in a richer description the guarded expressions will involve only a subset of the variable space, and thus it will be simpler to construct the individual MTBDD corresponding to a single command.

Chapter 7

A multiple module approach

We have already mentioned in 5.1.1 that the digitwise operations, performed in the case of the ISO-0 attack family, are characterised by a lot of symmetry, since the result of applying the proper XOR operations on a digit is independent of the range or value of the other digits. To break this symmetry, AnaBlock executes the digitwise commands in order, trying first all operations on the first digit of the PIN, then the second, then the third and then the fourth. An alternative approach to dealing with this kind of symmetry would be to take advantage of a symmetry reduction technique that would cut off the redundant states of the resulting model.

As we have already seen, PRISM provides two tools for symmetry reduction, PRISM-symm and GRIP, whose main aspects have been introduced in section 4.5. Though GRIP version 2006 would not allow the representation of our model in terms of symmetric processes given the constraints set by its input language, the additional features incorporated in the recently published optimised version of GRIP make this possible. The language, however, remains restrictive, and a different representation of the original model would be required as far as the variables (the parser sets too many constraints) and the implementation of the interactions between the modules are concerned, since GRIP still doesn't support communication via synchronisation labels. We have already mentioned that PRISM-symm out-performs GRIP when applied to a specification consisting of a small number of complex modules, whereas GRIP wins out when applied to a large number of simpler modules. As we will see, the model for PIN block attacks, expressed in terms of symmetric components, consists of a small number of processes which include a relatively massive number of local states and local variables. Therefore, the exponential blow-up in variables required by counter abstraction (see section 4.5) would make GRIP inefficient. Moreover, the experimental results presented in [DMP07] on three case-studies for MDP models

do not demonstrate any improvement in runtimes when using GRIP instead of PRISM-symm, while in one case GRIP takes a longer time to build the model. For the above mentioned reasons and because of time limitations, we have not attempted to use the GRIP tool for symmetry reduction, although this might have been helpful for comparison purposes.

7.1 Synchronisation of digitwise modules

AnaBlock constructs a PRISM model comprising a single huge module for all applicable operations. Since, however, the digitwise operations are applied separately on each digit, there is scope for modelling the PIN block attacks using multiple modules. Under this perspective, we can go for a one-module-per-digit representation, where each module is responsible for a single digit. After the execution of the digitwise modules, a final module can take care of the all-PIN attacks, if any are allowed by the API. In this context, we can exploit another important feature of PRISM, *module renaming*, which allows duplication of modules. Considering for example the standard ISO-0 attack, we observe that the operations which can be applied on digits P_3 and P_4 are the same. Thus, assuming that module M_3 is responsible for manipulating P_3 and module M_4 for manipulating P_4, the entire definition of M_4 can be replaced by a renaming of the form: `module M4=M3 [P3_could_be0=P4_could_be0, ...] endmodule`

The digitwise modules can be executed either in order, following AnaBlock's convention for breaking symmetry, or in parallel, so that digitwise symmetry is not removed from the model, since all possible orders of applying the available operations will be considered. In both cases, there will be 4 digitwise modules M_i, $i \in [1..4]$, where M_i is responsible for digit P_i. In the first case, each module M_i, $i > 1$, starts executing its commands only when there are no more operations that can be carried out on digit $i-1$, i.e. when module M_{i-1} has reached a locally final state. We have to keep in mind that, since the modules are non-deterministic, there are multiple locally final states in every module, each of which unblocks the next module when reached. If all-PIN attacks are supported, these locally final states will constitute the multiple initial states of a last module, which will carry on with the non-digitwise operations. To achieve this behaviour, the modules must be properly synchronised through the use of guard variables and action labels.

We also observe that for all possible combinations of digitwise attacks the operations on digit P_1 are always identical to the ones applied on digit P_2, and that the operations on digit P_3 are identical to the ones on digit P_4. This follows

when considering the possible combinations of all kinds of digitwise attacks: the restricted ISO-0 and the IBM 3624 PIN verification attacks involve operations only on P_3 and P_4, while the support of VISA-3 format allows XOR-ing with P_1 and P_2 as well. Thus, in all cases, modules M_2 and M_4 can be just defined as a renaming of modules M_1 and M_3, respectively. If these digitwise modules are executed in order, the resulting model will be identical to the one constructed by AnaBlock with the use of a single composite module. If the modules, instead, are executed in parallel, the PRISM-symm tool can be exploited to remove the symmetric states, by explicitly stating that the pairs of renamed modules are symmetric. A potential merit of the concurrent execution is that it doesn't require any interaction and sharing of variables between the symmetric modules, as the consecutive execution does. Intercommunication through the use of action labels and the reading of local variables that belong to other modules can turn out to induce considerable computational overhead on the model checker for some models. This can be avoided when the modules run in parallel, at the cost of much greater state space, which can, though, be noticeably reduced by applying symmetry breaking.

Regarding the parallel execution, the models were produced by hand. Since the experiments that were conducted demonstrated that this approach doesn't give good results (see the next section), there was no point in automating their construction. The ordered version was generated by a variant of AnaBlock, in which the `determine` predicate responsible for the digitwise operations has been properly altered, so that it performs all possible operations only on a specific digit, without recursive calls to subsequent digits. The consecutive execution of the modules is achieved through the use of the $done\_previous\_P_i$, $i \in [2, 4]$ variable, which is set to true in every state where digit P_{i-1} has been determined. The predicate `symmetric_module` takes care of the renaming of modules M_1-M_2 and M_3-M_4, so that there is no need for PROLOG to repeat the same computations. All variables referring to the PIN digits are declared to be local in each of the corresponding modules. The model produced by this multi-module variant of AnaBlock applies only to configurations that involve exclusively digitwise attacks, since the experiments showed that it is impossible to combine all-PIN with digitwise attacks using multiple modules (see the next section).

7.2 Experimental results and evaluation

Table 7.1 summarises the results we obtained for the configurations that allow only digitwise operations to be applied, i.e. configurations (5) and (1). We also

performed experiments on two new, made-up configurations. Configuration (8) allows solely IBM 3624 PIN verification to be performed, which only enables the recovery of digits P_3 and P_4, narrowing down the possible PIN values to 100. Thus, it is like configuration (4), but without the support of brute force guessing. In configuration (9) the translation function is activated, but IBM 3624 PIN verification is not allowed. Digitwise operations lead to the full determination of the PIN in 13.6 steps, the same as for configuration (1). The 'x' entries in the

Example	# states	Time			
		AnaBlock	Model Construction	Model Checking	Total
(1)-original	210849	24 min	42 min	30 min	1.6 h
(1)-ordered	210849	1 min	13 sec	53 sec	2 min
(1)-parallel	1041933841	–	13 sec	x	x
(1)-par-symm	523694496	–	60 sec	x	x
(5)-original	121	11 sec	27 sec	<1 sec	38 sec
(5)-ordered	123	1 sec	<1 sec	<1 sec	2 sec
(5)-parallel	441	–	<1 sec	<1 sec	2 sec
(5)-par-symm	231	–	8 sec	<1 sec	9 sec
(8)-original	452	5 sec	1 sec	31 sec	37 sec
(8)-ordered	454	1 sec	<1 sec	<1 sec	3 sec
(8)-parallel	1764	–	< 1 sec	<1 sec	2 sec
(8)-par-symm	903	–	9 sec	<1 sec	10 sec
(9)-original	186649	16 min	53 min	19 min	1.5 h
(9)-ordered	186649	38 sec	12 sec	30 sec	1 min
(9)-parallel	815730721	–	10 sec	x	x
(9)-par-symm	410278765	–	44 sec	x	x

Table 7.1: Experiments with multiple modules. '(n)-original/ordered/parallel' indicates the experiment on configuration (n) using the original, the ordered and the parallel multiple-module versions of Anablock, respectively. 'par-symm' indicates that the modules were executed in parallel, but that symmetry reduction was applied. 'x' denotes that PRISM failed to perform the corresponding stage, and '–' that no runtime measurements are available, since the relative model was written by hand.

table indicate that PRISM was unable to perform the corresponding stage: after constructing the MTBDD and calculating the number of states, PRISM aborts immediately if the number of states is excessively large, without being able to compute the state and transition matrices. This is because, although memory

consumption remains in acceptable levels, space requirements are prohibitive for models that exceed 400 million states, like configurations (1) and (9). It should also be noted that for these configurations, which allow operations on all four digits, it is not possible to declare both pairs of symmetric components, since PRISM-symm can only support one block of symmetric modules. Hence, only M_3 and M_4 were stated to be symmetric, as in all cases they involve equal or more operations than M_1 and M_2. In the following we present an analysis of the experimental results we have obtained, with insight in how each approach affects the model construction and model checking processes.

Ordered execution of digitwise modules

The results summarised in Table 7.1 demonstrate an immense drop in PRISM runtime when splitting the model into symmetric modules which are executed consecutively, as both model construction and model checking become a matter of some seconds. AnaBlock time decreases significantly too, since PROLOG has to perform only half of the required operations, because of module renaming. Especially for configurations (1) and (9), for which AnaBlock time constitutes an overhead that is not negligible, the decrease in time required by PROLOG to build the respective MDP is a noteworthy achievement. The amount of the decrease in runtimes is also apparent in Figure 7.1, where the total runtimes are plotted as histograms on a logarithmic scale. Regarding memory usage by PRISM, configurations (1) and (9), which are the heaviest ones, and require about 1.1Gb maximum memory when the original AnaBlock is used, peak at around 700Mb and 300Mb, respectively, when the ordered multiple-module version is used.

The impressive improvement in runtime is justified by a rationale similar to the one argued in section 6.2.1. There, we have seen how state space explosion can be circumvented by compositionally constructing symbolic MTBDD-based representations of complex systems from small-scale components, and that compactness for the representation can only be achieved if heuristics are applied with insight into the structure of the system under investigation. The one-module-per-digit model conforms better with this compositional approach, allowing PRISM to generate small, compact MTBDDs for the lower-level components, which efficiently reflect the underlying structure, characteristic of the operations on a specific digit. It turns out that the interaction among the 4 modules is not that heavy (they only have one variable in common for synchronisation reasons, while all other updates are performed independently), and hence the composition of these module-based MTBDDs proceeds particularly fast.

The results shown in Table 7.1 confirm that it is wise to invest in an optimal

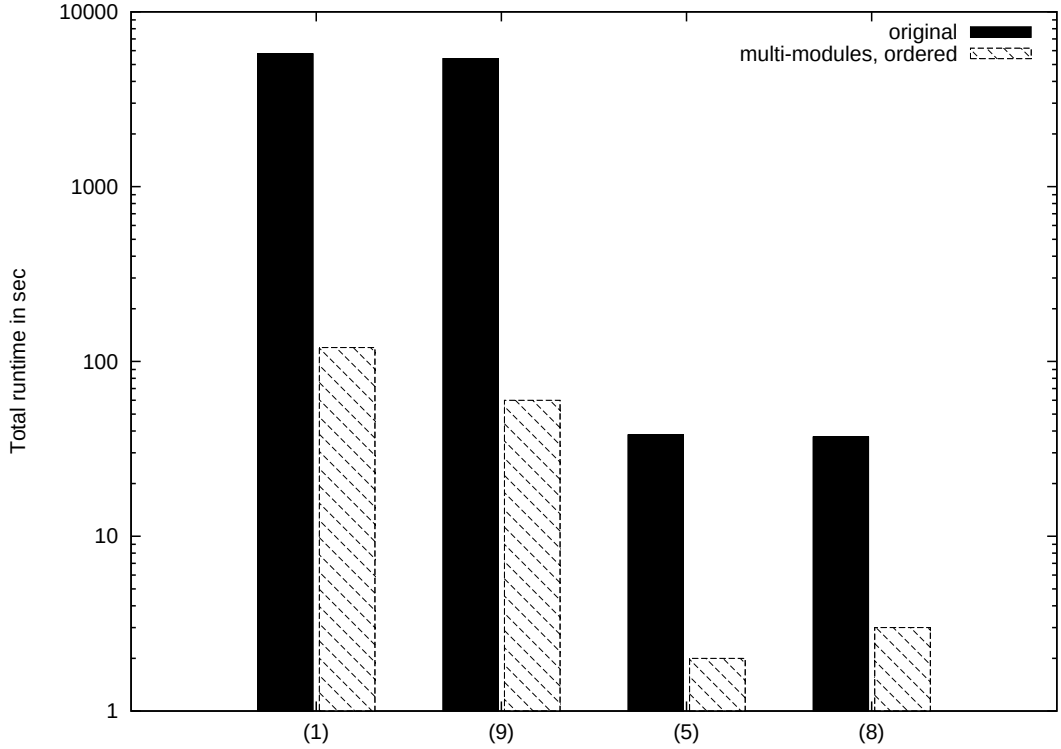

Figure 7.1: Total runtimes for each configuration.

encoding of the lowest level components, so that closely related commands and variables remain near to each other. Conversely, in the original one-module representations there is an interpolation between commands that concern different digits. To see how important it is for affined variables to be placed close to each other within the boundaries of a module, we performed one more experiment: we "tidied up" the digitwise commands in the original aggregate module, so that commands referring to a specific digit appear together. However, no improvement was recorded in the corresponding runtimes. This result implies that grouping variables by modules is what makes the difference.

Parallel execution of digitwise modules (without PRISM-symm)

As we can see in Table 7.1, as far as model construction time is concerned, the parallel approach is equally fast as the ordered approach (or even a little faster for some configurations), despite the fact that the former has to deal with many more states. This is possibly because the interaction between the modules required for their consecutive execution adds some small load to the model checker. In the case of concurrent execution, however, the performance of model checking is infeasible for the heavy configurations (1) and (9) (see the 'x' entries in Table 7.1),

because of the blow up in state space size as already has been explained. In general, it is not an uncommon situation for a model checker to be able to build the MTBDD representing extremely large models, but fail to conduct model checking, either in a fully symbolic fashion (because of irregularities introduced during the numerical computation, which cause an increase in the intermediate vectors that are used) or with the use of explicit data structures (because of excessive memory requirements).

Symmetry reduction with PRISM-symm

When applying PRISM-symm, the state space size is reduced almost 50% with respect to the number of states arising from the parallel execution of the digitwise modules. For configurations (1) and (9) that produce particularly large models, however, it remains 3 to 4 orders of magnitude higher than the number of states in the case of ordered execution. This is partly due to the fact that PRISM-symm can only deal with the one pair of symmetric modules, leaving many symmetric states unresolved. In addition to that, the interleaved execution of commands in different modules gives rise to extra states, in which the range of a digit has been narrowed down, while there are still pending operations on the previous digits. A unique representative of each of these additional states is included in the quotient model. As a result, the reduction in the number of states is not enough to allow model checking to be carried out. Moreover, we note that the application of symmetry reduction imposes some additional time to model construction, since the sorting process of the PRISM-symm algorithm (see section 4.5) requires some seconds to be conducted.

Another worth-mentioning observation is that, despite the reduction of state space size achieved after applying PRISM-symm, the size of the MTBDD representation of the quotient model increases. So, for example in the case of configuration (9), the MTBDD resulting from the concurrent execution of modules consists of 90595 nodes, while after the removal of symmetry the MTBDD comprises 1010429 nodes. This is not a peculiar phenomenon, since in [KNP06] a rise in the MTBDD size is also observed for one of the models under experimental investigation. In general, there are two contrasting factors which influence the size of the MTBDD representing the quotient model. Firstly, the removal of a large number of states decreases MTBDD size, while on the other hand the permutation of a large number of matrix elements during sorting destroys a great deal of the matrix's regularity, and hence increases MTBDD size. Thus, it follows that the more regularity a model exhibits, the more it will suffer from MTBDD size increase after the application of PRISM-symm. The experimental results imply

that structure plays a determinative role in our model, and therefore the size of the constructed MTBDD is unavoidably very large, with detrimental effects on model checking, which then can't be accomplished.

Module for all-PIN operations

Regarding the configurations that involve a combination of digitwise and all-PIN operations, PRISM runs out of memory whilst trying to construct the corresponding MTBDD. This memory overload is due to the final module that takes over the all-PIN operations after the completion of the digitwise modules. More precisely, the bottleneck lies on the initialisation of the variables of the last module to appropriate values, as determined by the locally final states of the previously executed digitwise modules. In general, heavy interaction between modules adds excessive complexity and burden to the model checker. Recall that PRISM groups variables by module, and the heuristics are based on the hypothesis that variables from different modules are more likely to be unrelated. Therefore, it is computationally very expensive to deal with a module that has many local variables whose value assignments depend on the values of local variables in other modules. Experiments with made-up models consisting of a limited number of states showed that, even when the state space is very small, the memory requirements for the necessary variable initialisations remain prohibitive. It should be mentioned that the use of global variables, in order to overcome the drawback of the initialisation phase of the last all-PIN module, allows PRISM to finish without running out of memory, but negates any reduction in runtime achieved by the multiple modules approach.

It can be asserted that given the inability of combining the digitwise and the all-PIN operations, the multiple module approach constitutes a compromise against the generality of the AnaBlock model, since the splitting into individual modules can only be applied in situations where we know in advance that the attack strategy includes only the execution of digitwise commands. Thus, if we want to take advantage of the tremendously short runtimes achieved by the multiple module approach, a discrimination into exclusively-digitwise and not-exclusively-digitwise APIs emerges, each of which requires a different treatment. It is evident that such a discrimination, which is not always easy to be done before the actual model checking takes place, is not desirable.

7.3 Summary

We have attempted to split the representation of original AnaBlock into multiple modules, having one module responsible for each digit as far as the ISO-0 family of attacks is concerned. These digitwise modules can be executed either in order or in parallel. In the latter case, we have tried to apply symmetry breaking by using the PRISM-symm tool. The experimental results demonstrated that the ordered application of digitwise operations in a multiple module manner is the best way to deal with the digitwise operations, decreasing the overall runtime to a few seconds (only 2 minutes were required by PRISM for the heaviest configuration). Regarding model construction time, all approaches yielded approximately the same results. In the case of the parallel approach, though, model checking turned out to be infeasible for heavy configurations, because of the blow up in the number of states. Symmetry breaking led to a reduction of the state space size to a half, which however was still not adequate to allow model checking to be accomplished. A major limitation of the ordered execution of multiple modules, which has proved to be so successful, is that it cannot be applied for configurations that require the performance of non-digitwise operations. It was impossible to avoid a memory overload when adding a final module to carry on with all-PIN operations.

Chapter 8

Breaking symmetry in the decimalisation table attack

Besides the symmetry in the case of digitwise operations, the subtrees resulting from applying the full decimalisation table attack (by full, we mean that the offset is allowed to vary) are also characterised by much replication. To illustrate this, consider the case where we get a dectab hit that reveals the presence of at least one 8 in the PIN block. By trying several offsets, the intruder will find out that the PIN is of the form $8xxx$, $x8xx$, $xx8x$ or $xxx8$, with $x \neq 8$, plus the combinations where there are two or three 8s in the PIN, giving 15 possible situations. The xs here indicate the digits whose value is yet unknown (we don't care about the range that has been reasoned for the unknown digits at this stage). The key observation is that the operations applied to the rest of the digits are independent of the position of the specied digit: first, the decimalisation table will be advanced, either revealing the existence of a specific value among the unknown digits (dectab hit), or just narrowing their range (dectab miss). In the case of a dectab miss, the next operation will be to advance the decimalisation table once more, in the case of a dectab miss the next operation will be to try the first of the set of suitable offsets. The subsequent steps will follow the same pattern. Thus, the sets of operations that make up the subtrees under these four states with only one digit determined will be roughly the same, and will bring about symmetric states, considering the variable space defined by the four digits (i.e. ignoring the values of the tried offsets). Eventually the subtrees produced under the 15 possible states will be symmetric, grouped according to the number of repeated 8s in the PIN. This will be better understood in the following, through an example.

8.1 Description

In order to break the symmetry characterising the full decimalisation table attack, we can take advantage of the fact that the position of a specified digit doesn't really matter: each time one or more digits are set to a specific value, they are shifted to the left, taking the first places after the last already known digit, which are shifted to the right. This rearrangement of the digits is performed by the `arrange_order` predicate. Care must be taken so that `arrange_order` is called every time a digit becomes known. This usually occurs after a correct offset has been discovered, but a dectab miss can also give rise to a full determination of a digit, when all possible values of the unknown digit besides one have already been excluded. PROLOG calculates the next states and operations by considering at each step the shifted digits of the PIN.

The rationale of this approach can be better illustrated through an example. Consider that we are at a state where the PIN is of the form $5xxx$, and a dectab hit occurs with the decimalisation table modified at position 8. If the original model of AnaBlock is adhered to, the subsequent operations will give rise to the states represented in the tree of Figure 8.1, where we have abstracted the model by disregarding the variables pertaining to the dectab and the offset. The subtrees $T1.1$, $T1.2$, $T1.3$ will be symmetric to each other, since only the order of the digits is different. The same applies to subtrees $T2.1$, $T2.2$ and $T2.3$ under the states where the existence of two 8s in the PIN has been revealed. Now, if the reordering algorithm described in the previous paragraph is followed, the resulting graph will have the structure illustrated in Figure 8.2. For each group of symmetric states and subsequent subtrees in Figure 8.1 there is only one representative, thus a lot of redundancy has been removed. Regarding a final state of the reduced model, which represents an exact value of the PIN, the first digit will always correspond to the value revealed by the first dectab hit that occurred in the paths that lead to this final state.

It is worth mentioning that this approach is more straightforward in comparison to the previous ones in the sense that we don't leave it up to PRISM to reduce the number of states or perform any other optimisations in regard to the resulting MTBDD structure, but rather feed an already reduced model into the model checker. Of course, when considering this decreased model, we can't follow the actual value and range of the digits through the states, since the reordering process distorts the form of the PIN. However, as we have already mentioned, our prime concern is to evaluate the degree of security of each API configuration in terms of the number of steps that are required in the worst case (and the best for the intruder) to reduce the range of PIN to some particular size.

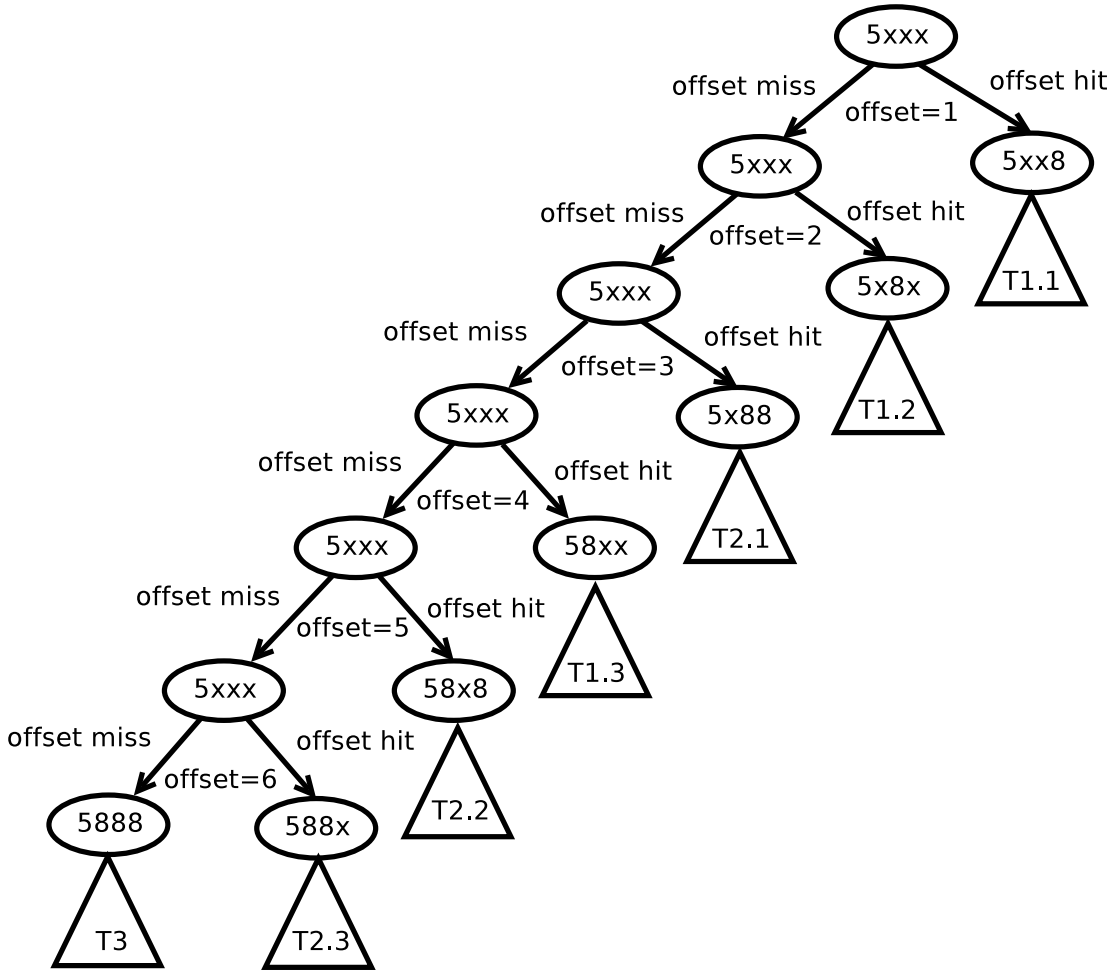

Figure 8.1: Part of the tree resulting when advancing the offset after a dectab hit at position 8, according to the original model of AnaBlock. For clarity we don't indicate the probabilities associated with each transition. The triangles indicate the subtrees under the relevant states. `offset=value` indicates the value of the tried digit for each pair of probabilistic transitions.

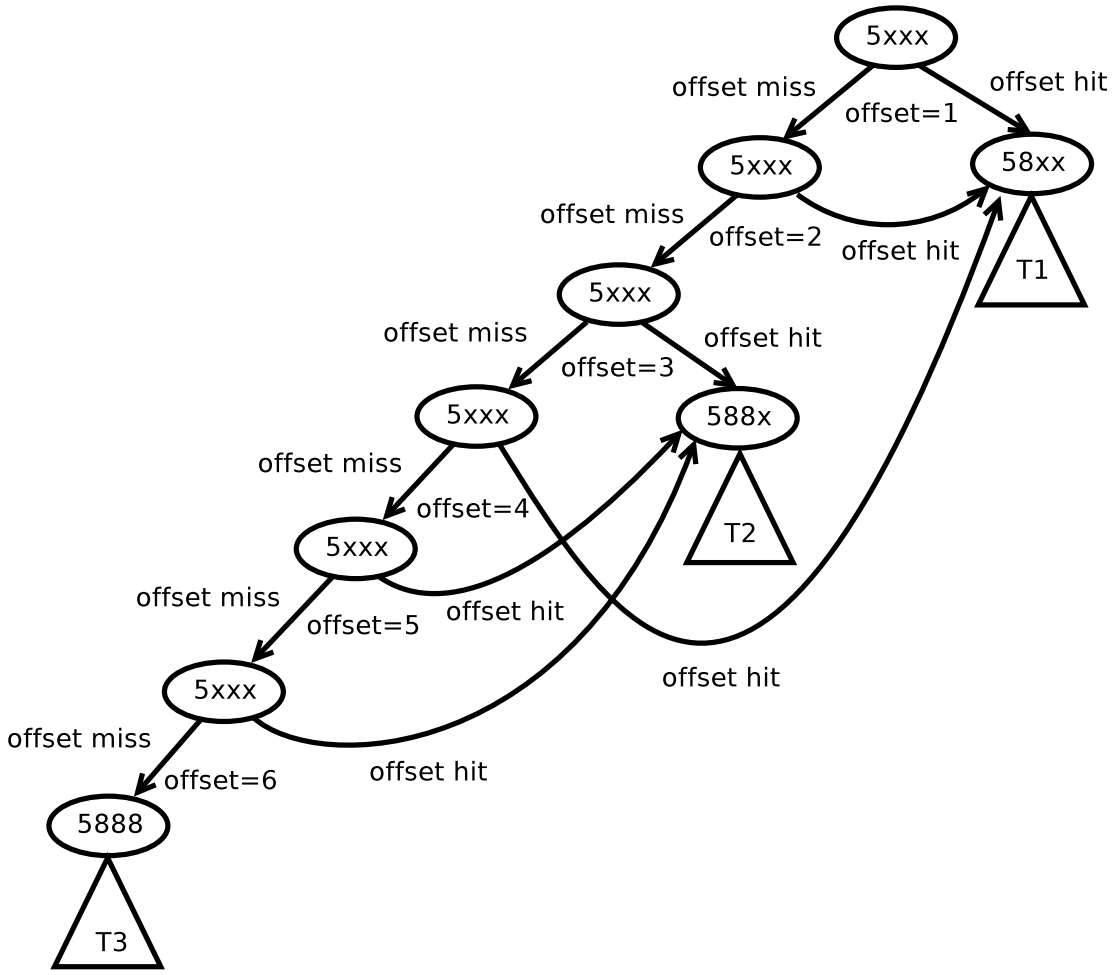

Figure 8.2: Part of the tree resulting when advancing the offset after a dectab hit at position 8, if the reordering algorithm is applied. For clarity we don't indicate the probabilities associated with each transition. The triangles indicate the subgraphs under the relevant states. `offset=value` indicates the value of the tried digit for each couple of probabilistic transitions.

8.2 Experimental results and evaluation

We have carried out experiments for configurations (2) and (3), which involve the full decimalisation table attack. We have also tested the model on a new configuration (10), which has no translation function, but allows IBM 3624 PIN verification. The PIN can be recovered in 15.3 steps in the best case for the intruder, the same number as for configuration (3), using a combination of the full decimalisation table attack and the IBM 3624 PIN attack. Digits P_3 and P_4 are determined by the digitwise operations, while the decimalisation attack further discovers digits P_1 and P_2. We recall that all configurations, for which decimalisation table operations are available and the offset is not locked down, allow full recovery of the PIN. The results of the experiments are summarised in Table 8.1, where we have also included the results for the original models, so that comparison can be made.

Example	# states	Time			
		AnaBlock	Model Construction	Model Checking	Total
(2)-original	19999	1 min	49 min	2 min	52 min
(2)-reduced	1871	14 sec	5 min	4 sec	5 min
(3)-original	20615	1 min	52 min	6 min	59 min
(3)-reduced	4191	30 sec	15 min	35 sec	15 min
(10)-original	21791	1 min	2.3 h	47 min	3 h
(10)-reduced	971	1 min	36 min	10 sec	38 min

Table 8.1: Experiments with symmetry reduction for the decimalisation table attack. (n)-original denotes the experiment on configuration (n) using the original AnaBlock model.'reduced' indicates that the model resulting after the removal of redundant states, resident in the full dectab attack, was used.

As we can see, the reduction of duplication leads to a massive decrease of the number of states. For configuration (2), which only allows the full decimalisation table attack, we note that the state space has been reduced 90%. The decrease of state space entails a remarkable drop in both model construction and model checking times. This relation between the state space size and the runtime is illustrated in Figure 8.3, where we have plotted the total runtimes (y-axis) against the number of states (x-axis) corresponding to the original and the reduced model, for each of the three configurations. The amount of the decrease in the total runtimes for each configuration becomes evident in the histogram of Figure 8.4.

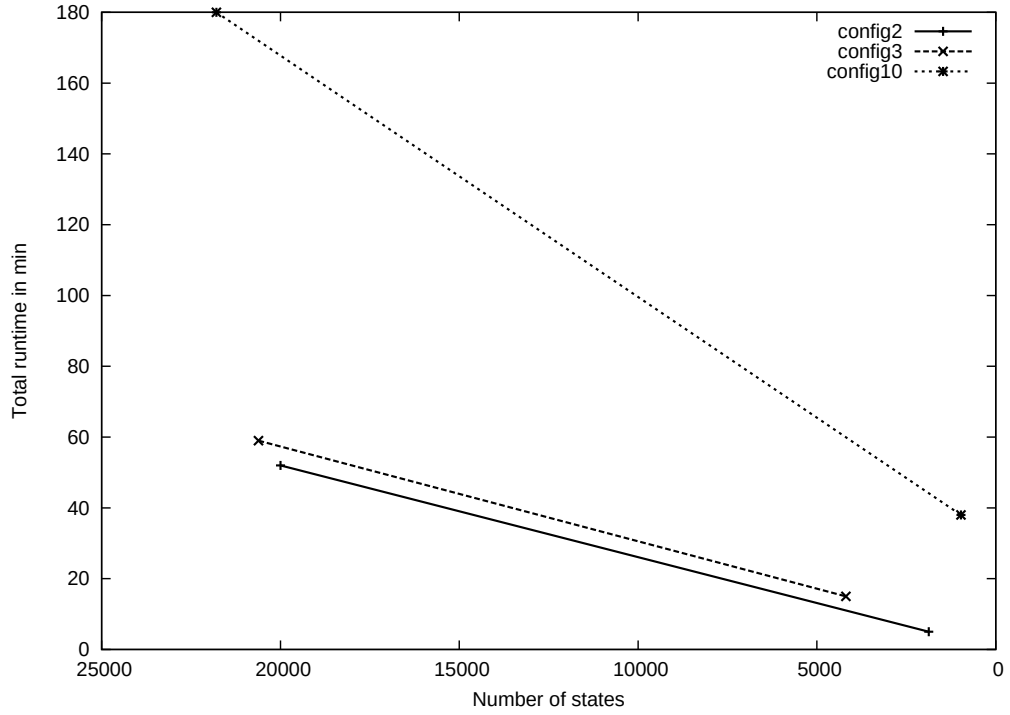

Figure 8.3: Runtime vs. state space size for the three configurations involving the full decimalisation table attack. The left point corresponds to the original model, the right to the reduced one.

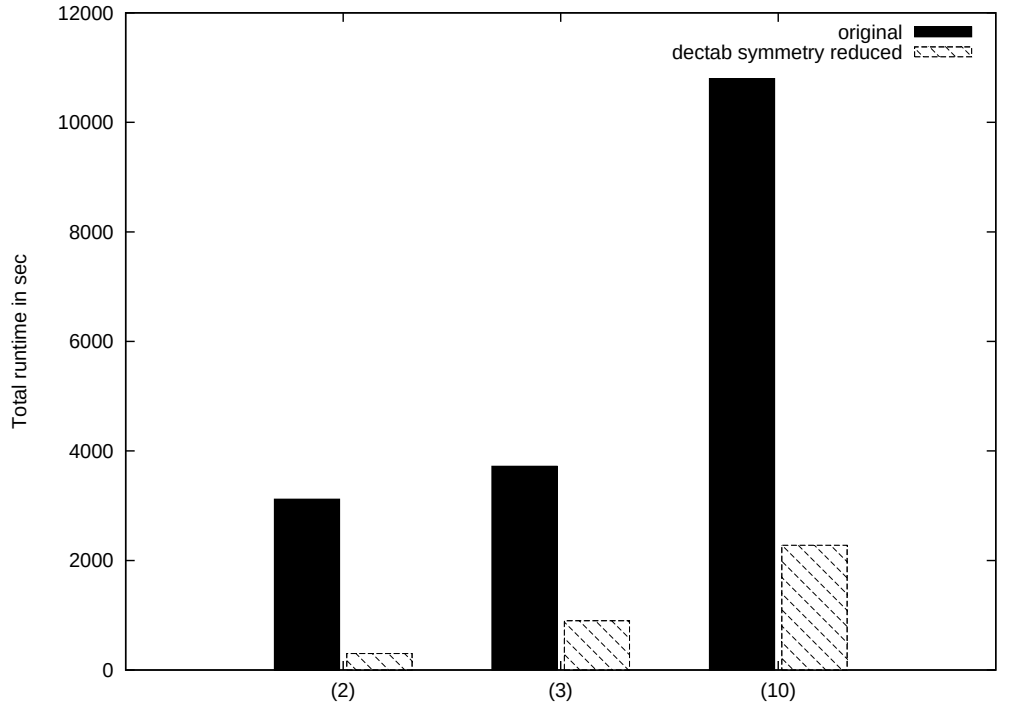

Figure 8.4: Total runtimes for each configuration.

Regarding memory consumption, when using the original version of AnaBlock, PRISM required for configurations (3) and (2) 1.1G memory maximum. When using the reduced model, memory usage was decreased to 400Mb and 700Mb respectively. Configuration (10) had a peak at around 1.3Gb when the original model was used, and at 1Gb after the redundant states were removed. Thus, the reduction of dectab duplication yielded improvement in memory consumption as well.

8.3 Summary

In this chapter we have explained how the application of the full decimalisation table attack gives rise to a lot of replication. We proposed an algorithm to get rid of the redundant operations, yielding a more compact model, which has no duplications and consists of noticeably fewer states. This reduced model led to considerable drop in the overall runtime. The time required for the heaviest configuration was reduced from 3 hours to 38 minutes.

Chapter 9

Discussion

The first section of this chapter summarises the contributions of our project, presenting its basic achievements and the key lessons learnt by our experience with the probabilistic model checker PRISM. Even in the cases where the results we got failed to meet our expectations, the experiments we conducted highlighted some significant issues that should be taken into consideration when representing a model in PRISM. The choice of an efficient representation demands thorough knowledge of the techniques PRISM uses, and the rules and heuristics that govern its behaviour, although the exact impact of the combination of factors that influence PRISM performance is hard to predict. In the last section, we present some directions that future research on improving the AnaBlock system can follow.

9.1 Conclusions

The first two approaches discussed in chapters 6 and 7, the 2-variable representation and the decomposition into digitwise modules, do not interfere with the actual model itself, but rather address some alternative ways of expressing the same state space and transition relations in PRISM syntax. The experimental results showed how dramatically PRISM representation can affect the way the corresponding symbolic data structures are constructed and manipulated, and consequently the performance of the model checker in terms of time and memory requirements. The experimental evaluation of the 2-variable approach indicated that a laconic PRISM representation can be highly inefficient, if it ignores essential features of the high-level abstraction of the corresponding model. The parallel version of the multiple modules representation (see section 7.1) was adopted as an alternative route to dealing with the digitwise symmetry. Though the model resulting from the concurrent performance of digitwise operations includes considerably more states, the ultimate goal was to minimise the state space size by

applying symmetry reduction, obtaining a smaller model that might possibly be faster to manipulate. However, this alternative approach to handling digitwise symmetry did not succeed. On the other hand, the third approach described in chapter 8, which deals with the duplications that arise from the full decimalisation table attack, concentrates on changing the model itself, by removing redundant operations and states. The result is a reduced model, which lacks some information contained in the original model, but still conveys all the knowledge that is necessary for finding the best attack and determining the expected number of the required steps.

The main two achievements of our work, with respect to enhancing the compactness of our models and the time and memory required for their construction and analysis, are the following:

- **Digitwise attacks**: Regarding the models that only involve the operations that constitute the ISO-0 attack family, the ordered execution of digitwise modules bore a striking drop in the overall runtime. This was especially apparent for heavy configurations, whose generation by PROLOG and analysis by PRISM took almost two hours when using the original AnaBlock representation, and became a matter of a few seconds when the one-module-per-digit representation was adopted. Memory consumption was reduced too. However, the applicability of the successive execution of digitwise modules suffers from a major limitation, since it was infeasible to combine digitwise with all-PIN operations using separate interacting modules. Hence, the utility of the one-module-per-digit approach, that led to such good results, is restricted only to configurations for which all-PIN operations are locked down or for which we know in advance that the allowed digitwise operations are enough to fully recover the PIN.

- **Decimalisation table attack**: For models that include the full decimalisation table attack, we have managed to obtain a more compact model with no duplications, which is much faster to construct and manipulate. The overall runtime was decreased from about one hour or more to a few minutes, while memory usage became less as well.

It is evident that the scope of improvements in time, memory and storage requirements for formally modelling and analysing attacks on security APIs is determined by the potentials and limitations of the available model checking techniques. The model resulting from the representation of all possible combinations and variations of the three families of PIN block attacks we have considered is a Markov Decision Process, which means that it includes both probabilistic transi-

tions and non-determinism. To our knowledge, PRISM is the only model checker that supports probabilistic quantification of described properties for systems that exhibit random or probabilistic behaviour. Therefore, the efficiency of analysing our model is restricted by the constraints set by PRISM, which we have covered in detail throughout the thesis. In future though, as PRISM constantly improves and new tools may arise, while computing resources become more powerful and possess greater amounts of memory, the restrictions we have faced in our project may be soon overcome. Concerning the model itself, irregardless of the model checker in use, as it is defined by its set of states and transition relations, we believe that we have managed to achieve a high degree of compactness, having carefully investigated the kinds of symmetry resident in the model.

All the changes we attempted in the modelling of PIN block attacks, both in terms of the model itself and its various representations in PRISM syntax, were based on observations specific to the inherent characteristics of the three families of attacks we considered. The AnaBlock system is simple enough to allow changes to be made as new types of attack are discovered. However, the incorporation of new attacks may lead to much larger models, whose analysis by PRISM demands higher memory and storage usage and longer runtimes, or even turns out to be infeasible. In such a scenario, one has to investigate alternative representations, that will exploit the particular features of the new kinds of attacks in an efficient way. It is at this stage that some general rules based on our experience over the course of this work can prove to be helpful. Below, we present some rules of thumb, that should be taken into account when developing models in PRISM. Although our remarks are based on the experiments with the different versions of the AnaBlock model, we believe that they are generally applicable to a wider range of models, because they are in accordance with the way PRISM algorithms work. The rules we suggest don't provide a definitive answer to what is the best representation (recall that many PRISM algorithms rely on heuristics, which don't always act as expected), but can act as a guide to enhancing the efficiency of model construction and model checking and avoiding bad design choices, which may appear to be beneficial at first glance. The following tips should always be considered in respect with the model under investigation, since their effects are closely interwoven and often contradictory, while their significance depends on the particular features of the specific model:

1. The structure and regularity resident in the high-level abstraction of a model are the chief sources of MTBDD efficiency and must be preserved by all means, even at the cost of a more verbose description. Reducing the number of MTBDD Boolean variables and maintaining structural information are

two opposing goals, but the experimental results of the 2-variable approach showed that the advantage of using fewer variables is outweighed by the loss of regularity.

2. The ordering of variables has a critical effect on the way the transition matrix is decomposed into submatrices and on the size of the emerging MTBDD. The choice of the optimal ordering is a very hard problem, but a general rule that should be followed is that variables and commands related to each other should be placed close together. Two variables are related if the value of one influences the value of the other, or if they appear in the same expression. The results we got when experimenting with the multiple modules approach suggest that it is very beneficial to delegate each set of similar commands to a dedicated module, so that the grouping of the variables becomes clear, in line with the partitioning of the model components.

3. Interaction between modules should be kept as low as possible. Interdependencies between local variables of different modules can introduce excessive load to the performance of the probabilistic model checker and possibly lead to a memory overload.

Concerning the symmetry reduction tool PRISM-symm, it should be underlined that its applicability is based on the assumption that the additional time it imposes to model construction will be less significant than the decrease achieved in model checking time. Thus, PRISM-symm is not suitable for models for which morel construction and not model checking time constitutes the bottleneck. Moreover, the permutation of a large number of matrix elements during the sorting phase of the algorithm partly twists the structure of the model, hence increasing MTBDD size in some degree. This implies that models for which regularity plays a major role, as the models produced by AnaBlock, may eventually be translated into a larger data structure, despite the removal of symmetric states (see section 7.2). It should also be mentioned that PRISM-symm can only handle one group of symmetric processes-modules, thus it is not able to resolve different kinds of component symmetry that may reside in the same model.

9.2 Future work

The aim of this project was to investigate and implement ways of exploiting the particular characteristics of the three families of PIN block attacks, in order to yield a model that would be more efficient to construct and manipulate. Within

this framework, given the limits set by PRISM functionality and by the model itself, we would be surprised if an alternative approach would bring about results that would cause us to radically revise our conclusions. Regarding the evaluation of our model, there is potential for more experiments to be made on new APIs, like for example the one that controls the HSM RG7000. These experiments might possibly reveal novel variants of attacks. Unfortunately, there was not enough time for this additional work to be done.

Beyond the strict area of our focus, there are other directions for development in order to enhance the applicability of the AnaBlock system. As remarked in [Ste06], one of the main weaknesses of the current AnaBlock framework is that the API specification file, which is given as input to AnaBlock, has to be written completely by hand. The definition of the rules, that specify what operations are available to an intruder when particular commands have been enabled, requires detailed hand-coding for each API, with no support of re-using the work that has been done for previous APIs in some way. The experience gained from the modelling of APIs suggests that there is potential for partly automating the generation of the API definition file. This would involve choosing a specification language that would allow the API designers to describe the operation of API commands, and then analysing these specifications to produce a set of all possible operations the intruder can call. The implementation of such an automated support demands considerable effort and good knowledge of the common characteristics of API specifications, but can make the analysis of available operations a more comprehensive and convenient task, and further facilitate the discovery of more variations of attacks.

Moreover, in [Ste06] it is indicated that there is scope for addressing some more functionalities that are supported by many HSMs, by adding more detail to our model. For example, a typical HSM doesn't allow an attacker to use an account number digit which is hexadecimal, as required in the ISO-0 attack. As a consequence, more operations may be required to construct the modification table, depending on the account number in question. To analyse this, we would also have to build a model of uniformly distributed account number digits, which seems too complex, but can lead to a richer description of real-world security APIs.

There is also potential for extending the quantitative analysis principles of AnaBlock to a more general framework for estimating the cost, in terms of time and computational resources, of an attack or combination of attacks to successfully meet its goal. Such a framework would be concerned with the question "How secure is this API?" rather than "Is this API secure?", considering attacks

whose success is subject to a probability bound and require considerable effort by the attacker. Besides the PIN recovery attacks we have already discussed, there are several attacks against cryptographic keys for which the time and resources available to the intruder are of prime concern (e.g. the parallel key search attack discovered in [RB03] for cracking DES keys and the meet-in-the-middle attacks discussed in [Bon01]). In the case of brute-force guessing attacks we have seen that the cost of a guessing action was assigned by the corresponding rule in the API specification file as a function of the number of possible PIN values at the current state. This approach can be generalised to cover a wider range of attacks and consider more parameters: each deduction rule of the form "$antecedent \rightarrow consequent$" (or "$current\_state \rightarrow next\_state$" in the context of a state transition model) modelling a specific attack action can be associated with a cost function. This would take as input the knowledge of the intruder represented by the *antecedent* and, depending on a number of parameters, such as the number of API calls or cryptographic operations required for a guessing action, would return the expected cost to obtaining the knowledge represented by the *consequent*. This idea could be seen as an adjustment of the formalisms proposed in [Cer04] and [AMRV06] for the quantitative analysis of security protocols to the special needs of security APIs.

9.3 Summary

In the first section of this chapter, we have made a comparative presentation of the three approaches to alternative representations of the PIN block attack model, outlining the main benefits of our modifications to the original Anablock systems and the limitations we were confronted with. We have also given some rules of thumb, epitomising the conclusions we have reached throughout our experience with PRISM. In the second section we have suggested some possible areas future research can focus on.

Bibliography

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, Jul. 1999.

[AMRV06] P. Adão, P. Mateus, T. Reis, and L. Viganó. Towards a quantitative analysis of security protocols. *Electronic Notes in Theoretical Computer Science*, 164(3):3–25, 2006.

[BA01] M. Bond and R. Anderson. API-level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, 2001.

[BC04] M. Bond and J. Clulow. Encrypted? Randomised? Compromised? (When cryptographically secured data is not secure). In *Cryptographic Algorithms and their Uses*, pages 140–151, Queensland, Australia, Jul. 2004.

[BO06] O. Berkman and O.M. Ostrovsky. The unbearable lightness of PIN cracking. Technical report, Tel Aviv University, Nov. 2006.

[Bon01] M. Bond. Attacks on cryptoprocessor transaction sets. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 220–234, London, UK, 2001. Springer-Verlag.

[Bon04] M. Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge, Jan. 2004.

[BW96] Bollig B and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[BZ02] M. Bond and P. Zieliński. Decimalisation table attacks for PIN cracking. Technical Report TR-560, University of Cambridge, Jan. 2002.

[Cer04] I. Cervesato. Fine-grained MSR specifications for quantitative security analysis. In *Fourth Workshop on Issues in the Theory of Security–WITS'04*, pages 111–127, Barcelona, Spain, 2004.

[CKS07] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes. In O. Grumberg and M. Huth, editors, *TACAS 2007*, number 4424 in LNCS, pages 538–552, 2007.

[Clu03] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[COC97] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics and Programs*, page 191206, Southampton, UK, September 1997.

[Dij70] E.W. Dijkstra. Notes on structured programming, 1970. Available at http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[DM06] A. Donaldson and A. Miller. Symmetry reduction for probabilistic model checking using generic representatives. In S. Graf and W. Zhang, editors, *Proc. 4th Int. Symp. Automated Technology for Verification and Analysis (ATVA'06)*, volume 4218 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 2006.

[DMP07] A. Donaldson, A. Miller, and D. Parker. GRIP: Generic representatives in PRISM. In *Proc. 4th International Conference on Quantitative Evaluation of Systems (QEST'07)*, 2007. To appear.

[DMV04] P. Drielsma, S. Mödersheim, and L. Viganò. A formalization of off-line guessing for security protocol analysis. In *Proceedings of LPAR'04, LNAI 3452*, page 363379, ETH Zürich, 2004. Springer Verlag.

[EFT93] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[GSJ+05] V. Ganapathy, S.A. Seshia, S. Jha, T. W. Reps, and R.E. Bryant. Automatic discovery of API-level exploits. In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pages 312–321, New York, NY, USA, May 2005. ACM Press.

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HKN+03] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, 56(1-2):23–67, 2003.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to represent and analyse Continuous Time Markov Chains. In *3rd Int. Workshop on the Numerical Solution of Markov Chains*, pages 188–207, Prensas Universitarias de Zaragoza, 1999.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, second edition, 2004.

[JM87] J. Jaar and S. Michaylov. Methodology and implementation of a clp system. In *Proceedings 4th international Conference on Logic Programming*, volume 1, pages 196–218, Cambridge, MA, 1987. MIT Press.

[Kei06] G. Keighren. Model checking security APIs. Master's thesis, University of Edinburgh, 2006.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66, Grenoble, Apr. 2002. Springer.

[KNP06] M. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In T. Ball and R. Jones, editors, *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4114 of *Lecture Notes in Computer Science*, pages 234–248. Springer-Verlag, 2006.

[Kwi03] M. Kwiatkowska. Model checking for probability and time: from theory to practice. In *Proc. 18th Annual IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 351–360. IEEE Computer Society Press, 2003.

[NS06] G. Norman and V. Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[RB03] R.Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 579–592, London, UK, 2003. Springer-Verlag.

[SHJ+02] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *2002 IEEE Symposium on Security and Privacy*, page 273284, Berkeley, California, May 2002.

[Shm04] V. Shmatikov. Probabilistic model checking of an anonymity system. *Journal of Computer Security*, 12(3/4):355–377, 2004.

[Ste05] G. Steel. Deduction with XOR constraints in security API modelling. In R. Nieuwenhuis, editor, *Proceedings of the 20th Conference on Automated Deduction (CADE 20)*, number 3632 in LNAI, pages 322–336, Tallinn, Estonia, Jul. 2005. Springer-Verlag Heidelberg.

[Ste06] G. Steel. Formal analysis of PIN block attacks. *Theoretical Computer Science*, 367(1-2):257–270, 2006.

[Vig94] Laurent Vigneron. Associative-commutative deduction with constraints. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 530–544, London, UK, 1994. Springer-Verlag.

[YAB+05] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R.L. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report TR-644, University of Cambridge Computer Laboratory, 2005.