# Formal Analysis of PIN Block Attacks

Graham Steel[⋆]

*School of Informatics,*
*University of Edinburgh,*
*Scotland*

**Abstract**

PIN blocks are 64-bit strings that encode a PIN ready for encryption and secure transmission in banking networks. These networks employ tamper proof hardware security modules (HSMs) to perform sensitive cryptographic operations, such as checking the correctness of a PIN typed by a customer. The use of these HSMs is controlled by an API designed to enforce security. *PIN block attacks* are unanticipated sequences of API commands which allow an attacker to determine the value of a PIN in an encrypted PIN block. This paper describes a framework for formal analysis of such attacks. Our analysis is probabilistic, and is automated using constraint logic programming and probabilistic model checking.

*Key words:* Security API Analysis, PIN Blocks, Constraint logic programming, Probabilistic model checking

## 1 Introduction

In an automated teller machine (ATM) network, it is vital to keep the personal identification numbers (PINs) typed in by customers secure as they are passed back to the card issuer for checking. Typically, each PIN is formatted into a 64-bit block, and encrypted under a secret shared key, ready for transmission. Different card issuers prescribe different ways of formatting PINs into 64-bit blocks. Different encryption keys are used in pairwise communication between different nodes in the network. As the PIN is passed through zones in the ATM network, it may have to be re-formatted and re-encrypted several times.

[⋆] The author is supported by EPSRC grant number GR/S98139/01.
*Email address:* `graham.steel@ed.ac.uk` (Graham Steel).
*URL:* `http://homepages.inf.ed.ac.uk/gsteel/` (Graham Steel).

To avoid exposing sensitive values in the clear, these PIN block manipulations are carried out inside tamper-proof hardware security modules (HSMs), which have strictly defined APIs regulating the use of PIN block manipulation functions. Recently, a family of attacks have been discovered which exploit these operations, and the error checking performed during them, to determine the value of a PIN in an encrypted block, [2,4]. These attacks are serious, sometimes reducing the number of operations required to guess a PIN to just a dozen or so, making attacks potentially lucrative and hard to detect. They are also hard for API designers to completely eliminate, since a certain amount of the functionality that gives rise to these attacks is essential for normal operation. To compound the problem, each node in the banking network typically requires a different set of operations to be supported, and so a different configuration of the API. Even if a manufacturer's default configuration is secure, customers may configure the HSM is such a way as to unintentionally endanger security.

Following discussions with API designers at a manufacturer of HSMs, nCipher[1] plc., it became clear that a formal tool was required that would take the specification for an API, together with the particular set of PIN block formats and operations required by a customer, and determine the expected number of guesses required to determine a PIN using the best available attack. This would allow designers (and customers) to experiment with different configurations, varying degrees of legacy support and new block formats, whilst being aware of their effect on the complexity of PIN guessing.

This paper presents a prototype of such a system. It uses constraint logic programming techniques, [3], to reason about the effects of particular commands on the range of possible PIN values. This reasoning leads to the generation of a model for all possible attacks, assuming uniform distribution of PINs, which is fed to the probabilistic model checker PRISM, [8]. PRISM extracts the best attack and returns the expected number of operations required to determine the PIN. If there is no attack that will always determine the PIN, PRISM can return the probability of reducing the number of possible PIN values to a particular range. A script post-processes the output from PRISM to obtain details of the best attack for the designer to study.

In the rest of this paper, we will first explain the family of attacks we are concerned with, in §2. In §3, we show how we generate models of possible attacks for a given configuration. We explain our use of PRISM in §4, showing how we process the output to obtain details of the most effective attack. Results on a number of different API configurations are given in §5. We consider related work in §6, further work in §7, and conclude in §8.

[1] `http://www.ncipher.com/`

## 2 Background

HSMs are used in ATM networks to protect sensitive data, such as cryptographic keys and PINs, from eavesdroppers, hackers and corrupt employees. An HSM typically consists of a tamper-proof enclosure containing a processor equipped to perform cryptographic operations and a small amount of memory. This memory is commonly used to store a master key for the HSM. All the other keys required to perform PIN generation, verification, encryption and so on are stored outside the HSM, encrypted under the HSM's master key. They can only be used by feeding them back into the HSM, along with the relevant data to be manipulated. This means that all sensitive values such as PINs and keys only exist 'in the clear' inside the tamper-proof enclosure. The operations allowed by the HSM are governed by a strict API, which is designed to impose security. This is achieved, for example, by imposing types on keys, and only allowing certain types of keys to be used for certain operations.

Security APIs can be thought of as a set of two-party security protocols, each one consisting of an input from the user, and a response from the HSM. The attacker may compose these protocols in any way he chooses to effect an attack. In recent years, several such attacks have been found on these APIs, [1,4]. Some of these are attacks on the key-management scheme, and can be detected by techniques similar to those for conventional security protocol analysis, [6,13,14]. Others are flaws in PIN processing, so called 'PIN Block Attacks', [4,2]. These attacks involve the attacker reasoning about the possible values of a PIN contained in an encrypted PIN block (EPB). His knowledge of the PIN is affected by the responses from the HSM to various commands. These 'informed guessing' attacks are outside the scope of previous approaches to security protocol analysis. It is the latter type of attack that we are concerned with in this paper.

PIN block attacks assume that the attacker has access to an HSM, and a correct, but encrypted, PIN typed by a customer. He has access to the key required to verify the PIN, i.e. to obtain a yes/no answer from the HSM as to whether the PIN is correct, but only in its 'safe', encrypted form – so he must use it under the terms of the API. The problem for the attacker is to manipulate the inputs to the API commands in order to determine the value of the PIN in as few operations as possible. We are concerned in this paper with three families of PIN block attacks: digitwise attacks such as ISO-0 conversion attacks, decimalisation table attacks, and brute-force guessing attacks such as the check value attack. We will explain each of these in turn.

### *2.1 ISO-0 (ANSI X7.8) Attacks*

The ISO-0 attacks, [4, p. 75], are a family of digitwise attacks, i.e. attacks which determine each PIN digit in turn. The ISO-0 attacks are so-called because they exploit a property of the ISO-0 PIN block format [2]. For a 4-digit PIN, the ISO block format is defined like this:

$$\begin{aligned} B1 &= 0\ 4\ P_1\ P_2\ P_3\ P_4\ F\ \ F\ \ F\ \ F\ \ F\ \ F\ \ F\ \ F\ \ \ F\ \ \ F \\ B2 &= 0\ 0\ 0\ \ \ 0\ \ \ A_1\ A_2\ A_3\ A_4\ A_5\ A_6\ A_7\ A_8\ A_9\ A_{10}\ A_{11}\ A_{12} \\ \text{PIN Block} &= B1 \oplus B2 \end{aligned}$$

Each character in the block represents a 4-bit nybble. The 0 at the beginning of $B1$ marks the block as being in format 0. The 4 indicates the length of the PIN, in this case 4 digits. The $P_i$s are the digits of the PIN, the Fs are the fixed hexadecimal values F, and the $A_i$s are the 12 digits of the customer's personal account number (PAN). The $\oplus$ symbol signifies bitwise exclusive-or (XOR).The principle behind XORing the PIN against the PAN is to diversify blocks containing the same PIN, to defeat 'code book' attacks by eavesdroppers.

The attack arises when the PAN has to be supplied to a command in order, for example, for the encrypted PIN block to be translated into another format. For this operation, the block will be decrypted inside the HSM, and then the supplied PAN will be XORed against the clear block to reveal the top line (B1), so that the PIN digits can be formatted another way. At this point, the HSM performs an error check, to see if all the PIN digits are of decimal value. Now, suppose that instead of supplying the correct PAN, the attacker supplies a modified PAN. This modification could be to XOR in the value 8 against the first digit, producing $A_1' = A_1 \oplus 8$. Now, if the third PIN digit $P_3$ is 0, 1, 8 or 9, the error check will still pass, since these values XOR 8 all give a decimal value. However, if the PIN digit is in the range 2–7, the HSM will signal an error, since these values all XOR against 8 to give a value between A and F hexadecimal.

By repeating this process with various values, an attacker can narrow down the value of the third PIN digit. He can only narrow it down to a pair of possible values though, since each pair of digits, 0 and 1, 2 and 3, etc., will give an identical pattern of errors and passes when XORed against values chosen by the attacker.

[2] This is format 0 in ISO Standard 9564-1 (2002). It is the same as the ANSI X7.8 standard format.

As it stands, this attack is only applicable to the third and fourth PIN digits, since in the prescribed format, there is no overlap between the PAN and the first and second digits. Clulow devised an extension to the attack to overcome this. The trick is to pass off the ISO-0 PIN block as being under a different format, namely VISA-3. The VISA-3 block format looks like this:

$$P_1\ P_2\ P_3\ P_4\ F\ F\ F\ F\ F\ F\ F\ F\ F\ F\ F\ F$$

Suppose for a moment that the PAN is 00000000000, so our ISO-0 block looks like $0\ 4\ P_1\ P_2\ P_3\ P_4\ F\ F\ F\ F\ F\ F\ F\ F\ F\ F$. Now suppose we give the HSM an ISO-0 PIN block, claim that it is in VISA-3 format, and ask for it to be translated to ISO-0 format. The result is:

$$0\ 6\ 0\ 4\ P_1\ P_2\ P_3\ P_4\ F\ F\ F\ F\ F\ F\ F\ F$$

Why? Because the HSM first looks for a left-justified F-terminated string of decimal digits as a VISA-3 PIN Block. It finds $0\ 4\ P_1\ P_2\ P_3\ P_4$. This it assumes to be a 6-digit PIN[3], which it reformats in the ISO-0 style. Now the real PIN digits have been shifted so that they will all overlap with the PAN, and so can be attacked by the above method. Furthermore, when we consider the possibilities when using a non-zero PAN, the attack enables us to determine all digits uniquely. This is because there are now three possibilities for the digits overlapping the PIN: they can be decimal, and accepted as a part of the PIN; they can be F, marking the end of the PIN (and also causing no error); or they can be in the range A-E hexadecimal, in which case an error will be reported. This allows the attacker to determine the exact value of a PIN. In this paper, we show that the expected number of operations for this attack is 3.4 for each PIN digit, making 13.6 operations for a 4-digit PIN.

### *2.2 Decimalisation Table Attacks*

This family of attacks was discovered by both Bond and Zieliński, [2], and Clulow, [4, §3.5.5]. Many PIN schemes[4] assign PIN values by encrypting a customer's PAN under a secret PIN derivation key (PDK), and then decimalising the result using a decimalisation table (or 'dectab'). A decimalisation

[3] ISO Standard 9564-1 (2002) specifies that PINs may be between 4 and 12 digits long.

[4] For example, the IBM 3624 scheme, Netherlands PIN-1 scheme, and the German Bank Pool Scheme.

table maps each hexadecimal value to a decimal. The ‘standard’ 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 |

So, if the first four digits of the result of encrypting a customer’s PAN under the PDK are 4A6B hexadecimal, the assigned PIN will be 4061. Some schemes allow the customer to change her PIN at an ATM. This is achieved by fixing an offset, which when added digitwise modulo 10 to the original PIN, gives the customer’s chosen PIN. This offset is not considered to be security critical, since without the original PIN it provides no help in guessing the correct customer PIN.

Decimalisation table attacks do not determine the PIN digitwise, but rather determine first what digits are in the PIN, and then where these digits are. Suppose an attacker has an encrypted PIN block which, when supplied along with the standard decimalisation table and a known offset, correctly verifies inside the HSM (that is, the HSM reports that the PIN is correct). Now suppose the attacker alters the decimalisation table like this:

| Old value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| New value | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 1 | 2 | 3 | 4 | 5 |

He then tries the PIN verification again, with the modified dectab. If the verification still passes, then he knows there are no 0s in the PIN. If however the verification now fails, he knows there must be at least one 0 in the PIN somewhere. The problem now is to determine how many, and where. This can be accomplished by altering the offset. The attacker advances the offset by one at each position, and then at every combination of positions, until the PIN is once again reported as being correct. This reveals the location of the 0s in the PIN. The table below illustrates the process, for an example where the customer’s PIN is 3060, and the offset is 0000. We assume the attacker has already tried the modified dectab shown above, and discovered that there is at least one 0 somewhere in the PIN.

| Attacker set offset | Result from HSM | Knowledge of PIN |
|---|---|---|
| 0001 | Incorrect PIN | ???? |
| 0010 | Incorrect PIN | ???? |
| 0100 | Incorrect PIN | ???? |
| 1000 | Incorrect PIN | ???? |
| 0011 | Incorrect PIN | ???? |
| 0101 | Correct PIN | ?0?0 |

The decimalisation table attack, as described by Bond, takes an average of 16.5 guesses to determine a four-digit PIN, [2].

## *2.3 Brute Force Guessing Attacks*

ANSI standard X7.8 specifies that 'The system shall not be capable of being used or misused to determine a PIN by exhaustive trial and error'. However, APIs can sometimes inadvertently allow this. An example of such an attack is the check value attack discovered by Clulow, [4, §3.5.8]. It relies on an attacker being able to obtain the check value of a PIN derivation key (i.e. a 64 bit block of zeros encrypted under that key). Many APIs support a key check value command, used to ensure that a key has been imported correctly. The attacker must also be able to supply a block of 0s as the PAN to a command for verifying PINs calculated using the method shown above (§2.2).

The first step is to obtain the check value of the PDK, and decimalise the first four characters of the result using the standard decimalisation table. Store this as IPIN. Now, supply a PAN of 000000000000 to the verify PIN function, along with some encrypted PIN block (EPB) you want to crack. Start with offset 0000. Generally, the command will at first fail. Increase the offset by 1 until the command reports a successful verification. Store the final offset as OFFSET. Now we know that the PIN in the block verifies successfully when compared against IPIN + OFFSET (mod 10 each digit), and so this must be the customer's PIN.

On average, for a four-digit PIN, this attack would require one call to the check value command and 5000 calls to the PIN verify function. Although not very efficient in itself, attacks like these can be sometimes be combined with others to finish off the cracking of a PIN, so it is important to know if such attacks are possible on a given API configuration.

### *2.4 The Problem for API Designers*

API designers face the problem that they are aware of these attacks, but often have no choice but to support the functionality that gives rise to them. Legacy PIN schemes must be supported, various different PIN block formats may be needed, and different decimalisation tables may be used by different card providers. They must find ways of 'locking down' (i.e. preventing an attacker from manipulating) certain inputs to the API in order to thwart the attacks. The problem is to find ways of doing this that result in minimal inconvenience and loss of flexibility for their customers. To add to the problem, each customer typically configures the API is a different way, specifying a list of commands they require and other parameters such as decimalisation tables and settings on keys. The approach taken in this paper was motivated by discussions with a manufacturer of HSMs, nCipher. They indicated that what was required was a system that would determine the best (known) attack available as for a particular configuration of their API. It is important that the system is simple enough for the API designers to be able to change it as new types of attack are discovered. We present here a prototype for such a system, based on constraint logic programming and probabilistic model checking.

## 3 Generating Models

We model PIN cracking attacks as trees. Each node in the tree represents a state, where the attacker knows the PIN is in a certain range. For one particular attack, the edges in the tree will represent probabilistic choices, e.g. where the attacker tries a command, and the HSM may or may not report an error, depending on the actual (unknown) value of the PIN. The probabilities of each outcome depend only on properties of the state we are in, i.e. our attacks are represented by Markov chains (MCs). For a family of attacks, the tree will contain both probabilistic choices and non-deterministic choices, where the attacker chooses what he will try next. A family of attacks is thus represented by a Markov decision process (MDP). The idea is to analyse security in two stages: first, to generate an MDP representing all possible attacks, and then to use a probabilistic model checker to analyse the MDP and choose the MC representing the most effective attack. This is either the attack that determines the PIN in the lowest number of steps, or in situations where no such attack exists, the attack that reduces the PIN to the smallest number of possible values.

More formally, the nodes in our trees represent states as tuples. The first four elements of the tuple, $P_1, \ldots, P_4$, are constrained integers representing the intruder's knowledge of the PIN. In the initial state, all four are constrained to the range $0 \ldots 9$. Successive nodes in a path through the tree will have monotonically decreasing ranges of possible values. The other three values in the tuple record information for the special case of the decimalisation table attack. Section 3.2 below explains their meaning and usage.

Edges in the tree denote transitions. From each node, there are $n$ sets of edges available, for some $n \in \mathbb{N}$. If $n = 0$ then the node is an endpoint (for example, when all the PIN digits have been uniquely determined, or when no further operations are available). If $n = 1$, there is only one set of possible transitions from this node. Otherwise, each set of edges represents a non-deterministic choice. There are one or two edges in each set. A transition set with two edges represents an intruder calling an HSM command with some particular input values. One edge represents the successful execution of the command, and the other represents the HSM reporting an error. Each edge has an associated probability. The sum of these probabilities is 1. Additionally, each edge has an associated cost of 1, representing one command call. A transition set with one edge represents brute-force guessing. The probability associated with this edge is 1, and the cost depends on the range of possible PIN values represented by the node from which the edge originates (see §3.3).

As an example, Figure 1 shows a fragment from one of our trees. The line styles indicate non-deterministic choices, i.e. the pair of solid lines indicate one non-deterministic choice, the dashed lines another, etc.. The numbers on each edge indicate probabilities. For simplicity, the nodes are labelled only with the variable whose constraint changes during the transitions shown, i.e the third PIN digit, $P_3$. This tree represents three possible initial steps the intruder could take with an API that supports the ISO-0 format and a translation function.

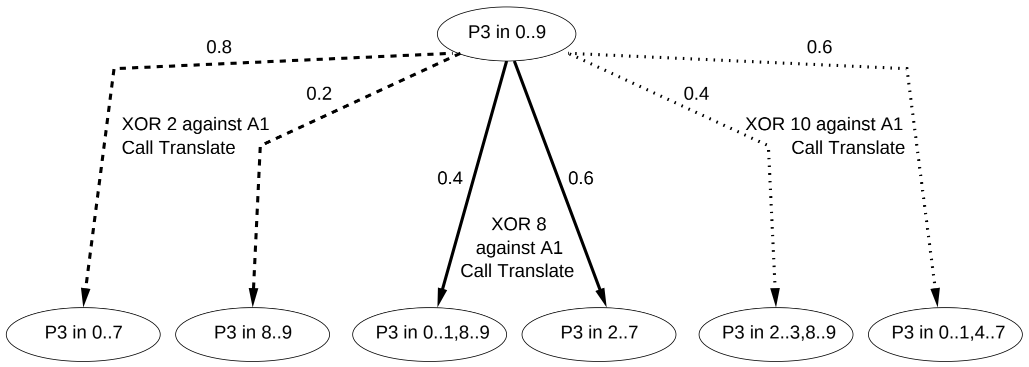

Fig. 1. Part of a model for the ISO-0 attack family

To construct an exhaustive tree of possible attacks for a given API and configuration, we use constraint logic programming in SICStus PROLOG, [3]. This is suitable because the constraint system allows us to refer easily to the range of possible values of a PIN or PIN digit, resulting in concise code. We also make use of facilities for meta-programming in PROLOG, i.e. writing clauses which generate the program, which in turn generates the model. Hence model construction proceeds in two phases: in the first phase, we analyse the API and the configuration chosen by the customer, and assert clauses for all operations available to the intruder. In the second phase, we find all possible ways of chaining these commands together to try to determine the PIN.

Figure 2 shows how the different parts of the system, which we call AnaBlock, relate to each other. The purpose of the config file is to specify an exact configuration of a product that a customer is considering installing. The first part of this file is a list of commands in the HSM's API that the customer intends to enable. The second part specifies other install-time options specific to the API in question. For example, for the nCipher payShield API, if IBM PIN verification keys are used, the customer can specify a number $n$ for each key that specifies the number of right-justified PAN digits which must appear in the validation data. The minimum $n$ that is used in an installation can have an impact on the best available attack, so is included in the config file.

The API file is designed to be modified by an API designer or security engineer. It consists of a set of rules which specify what operations are available to an intruder when particular commands have been enabled. Currently, the main weakness of our system is that some detailed work is required to produce an AnaBlock API file from the written specification of the API. This work must be done by hand for each API, though our experience in modelling the IBM 4758 CCA API and nCipher payShield API suggest that much work can be re-used. Though this method seems to work well for assessing the vulnerability of a system to families of known attacks, it does not provide any guarantees of security against unknown attacks, although the process of writing these API definition files has already allowed us to discover a previously unknown variation of the ISO-0 attack (see §4). The main aim of our immediate future work will be to address these problems, by allowing the API designer to directly specify the operation of API commands, and then analysing this specification automatically to generate the API definition file (see §7).

The rules in the API file are used by AnaBlock to meta-program the clauses of a recursively defined predicate called `determine`. Each clause corresponds to a particular HSM command, with particular values for the user-set inputs like the PAN and offset. By executing a suitable `findall` query on this predicate, we generate the tree of all possible attacks. The `determine` clauses are meta-programmed in such a way as to restrict the paths that can be taken in the tree to those which make some sense. This is accomplished by labelling each operation as being digitwise, an operation on the whole PIN, or a guessing step. Then, since brute force guessing is expensive, we only allow such operations to be used when nothing else applies to the state we are in. Whole-PIN

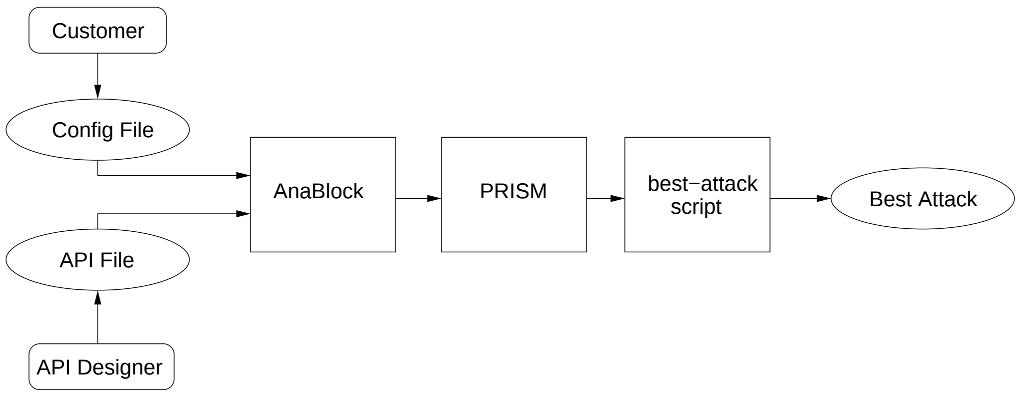

Fig. 2. AnaBlock system diagram

operations can only be used on a given state if no digitwise operations are available, and guessing steps only when nothing else is available. This pattern of execution of the `determine` predicate is illustrated in Figure 3. As a further optimisation, to break symmetry when considering digitwise operations, we try first all digitwise operations on the first digit of the PIN, and then the second, then the third and then the fourth.

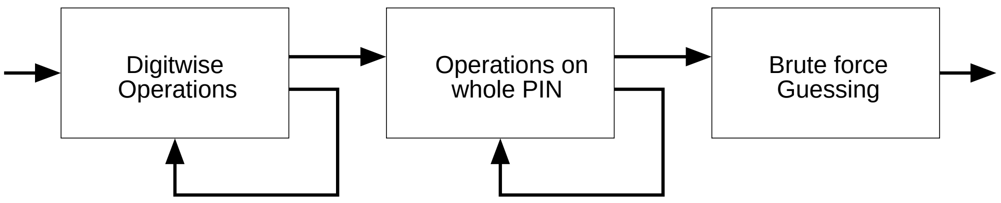

Fig. 3. Operation of `determine` predicate

## *3.1 Modelling Digitwise Operations*

A `determine` clause for a digitwise operation creates two new states: one corresponding to the intruder's revised knowledge of the PIN if an error is signalled by the HSM, and one corresponding to that for a successful execution of the command. The `determine` predicate is then recursively called on these two states. The following algorithm describes in detail the operation of such a clause:

(1) If we have already tried this command on the same PIN range, stop.
(2) Calculate the overlap between the range of PIN values accepted by this command, and the current range of possible PIN Values. Call this `Accept_Range`
(3) Calculate the overlap between the range of PIN values rejected by this command, and the current range of possible PIN Values. Call this `Error_Range`
(4) If `Accept_Range` $= \emptyset$ or `Error_Range` $= \emptyset$, stop.

(5) Count total number of possible PINs that satisfy `Accept_Range`.
Call this $p$
(6) Count total number of possible PINs that satisfy `Error_Range`.
Call this $q$
(7) Output in PRISM format a transition, with the probability of moving into a state representing `Accept_Range` as $\frac{p}{p+q}$, and the probability of moving into a state representing `Error_Range` as $\frac{q}{p+q}$
(8) Recursively call `determine` on `Accept_Range`
(9) Recursively call `determine` on `Error_Range`

To count the number of possible PINs that satisfy some constrained range, we use the built-in meta-predicate `labeling` to enumerate all possible assignments of constrained variables. Given that we are assuming uniform distribution of PINs inside encrypted PIN blocks, this leads directly to the correct probability of getting an error from the HSM. This automated approach for calculating transition probabilities is an important part of our technique, since it provides modularity, in the sense that commands can be combined in any order and the probabilities will be correct. This allows us to analyse varying configurations and APIs without having to change AnaBlock itself.

## *3.2 Decimalisation Table Attacks*

As mentioned above, the first 4 arguments of the `determine` predicate are the PIN digits, each expressed as a range of possible values. Clauses for modelling whole-PIN operations, i.e. decimalisation attacks, also refer to the last 3 arguments. The 5th argument is called `Last_Dectab`, and contains the value of the last decimalisation table tried against the PIN. An integer value $i$ here means that the last dectab tried was the standard table modified by adding 1 to all values $i$, following the pattern in §2.2. The 6th argument is `Last_Dectab_Hit`, which records the value of the dectab last time we hit a digit, i.e. last time the PIN failed to verify after we modified the table. The 7th is `Last_Offset`, which records the last offset value which was tried. Our algorithm for executing the dectab attack runs like this:

```
(1) Start with Last_Dectab=-1, Last_Dectab_Hit=11, Last_Offset=0.
(2) If all PIN digits determined, stop.
(3) If Last_Dectab_Hit > Last_Dectab:
        Increase Last_Dectab by 1
        If the dectab hits, set Last_Dectab_Hit to Last_Dectab and go to 4,
    else adjust digit constraints and go to 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

This algorithm is executed in AnaBlock by two `determine` clauses. The first succeeds only when the value of `Last_Dectab_Hit` is greater than `Last_Dectab`, in which case it increases the dectab and creates two new states, one for a a dectab hit and one for a miss, and recursively calls `determine` on these states. The second clause succeeds only when `Last_Dectab_Hit` is equal to `Last_Dectab`, in which case the offset is increased by one and two new states are again created, one for an offset hit and one for a miss. The values of `Last_Offset` and `Last_Dectab_Hit` are adjusted as required for these states, and again a recursive call is made.

To apply suitable constraints to the PIN digits for these newly created states, we make use of the SICStus propositional constraints mechanism. For example, if we get a dectab hit with the decimalisation table increased at position 2, we know at least one of the PIN digits is 2. We add the following constraint to the PIN digits in the new state created:

$$P_1 = 2 \vee P_2 = 2 \vee P_3 = 2 \vee P_4 = 2$$

To adjust the constraints for a dectab miss, it is a simple matter of removing the dectab value from the range of possible values for each digit. An offset miss is more complicated, and requires propositional constraints. Again, this is best illustrated by an example. Suppose we have had the dectab hit at position 2, as above. Suppose the first suitable offset to try is 0001. If this fails to verify, then we know that either the last PIN digit is not 2, or if it is 2, then there is at least one other 2 in the PIN. This is most concisely expressed by adding the following constraint to the state created:

$$\neg(P_1 \neq 2 \wedge P_2 \neq 2 \wedge P_3 \neq 2 \wedge P_4 = 2)$$

Note that our scheme for the decimalisation table attack is a little different to the original method proposed by Bond. His attacker first tried all dectab values, collated the hits, and then stepped through the offsets as required, resulting in an average of 16.5 operations to determine the PIN. Our scheme is fractionally more efficient, requiring 16.15 operations (see §5), but more pertinently it is simpler to describe as a transition system for a model checker.

## *3.3 Brute Force Attacks*

A brute force attack is available when, for example, as explained in §2.3, a check value command and an IBM PIN verification command are enabled. In this case, a `determine` clause is added to the model which simulates this brute-force attack as a single ‘black box’ transition, with probability 1 of success, and with an associated cost of $(n/2) + 1$ for a state representing $n$ possible PIN values (the 1 is the initial call to the check value function). Different brute force guessing attacks may have different costs - each one is added along with the relevant formula for calculating the cost. Note that the actual values of the PIN variables are not set by a brute force guessing step, but we do not recursively call `determine` on the resulting state, so the node reached becomes an endpoint. In the output for PRISM, this transition sets the 4 boolean variables `P1_guessed`, `P2_guessed` etc. to true. This allows us to specify this guessed state together with those where the PIN digits have been set to unique values when we do our model checking.

### *3.4 Output for PRISM*

The models we output for the probabilistic model checker PRISM are written in PRISM’s own input language, [8]. Each model consists of a single PRISM ‘module’, or process. PRISM does not reason directly about constraints, so we must create explicit Boolean variables for representing PIN digit ranges (`P1_could_be0, P1_could_be1, ...`). We also add a variable which holds the number of possible PINs represented by a particular state. This is calculated by AnaBlock, again using the `labeling/2` meta-predicate. This variable, `PIN_Possibilities`, is used for specifying properties when model checking. Finally, as mentioned above, our PRISM models include the 4 extra Boolean variables `P1_guessed`, `P2_guessed`, etc.. In the initial state, these are set to false. They are set to true as each PIN digit is determined, or when brute force guessing is used to guess all digits.

## 4 Analysing Models

Having generated a model reflecting all the options available to the attacker, we use PRISM to analyse the model to extract the best attack. We used a beta-release of PRISM version 3.0, which is now publicly available. The costs/reward mechanism is still under development, and this is the first version which allows the user to export the costs for each transition, which is necessary to reconstruct attacks discovered by our method. PRISM has several engines for model checking, but currently only the multi-terminal binary decision diagram (MTBDD) engine supports costs and rewards calculations. PRISM supports PCTL, [7], a probabilistic extension of the CTL logic, as its language for describing properties of Markov decision processes.

## *4.1 Checking Correctness*

Before examining attacks, we would like to be sure that the models we have created are in some sense correct. The use of a model checker gives us a natural means of doing this. We can specify some correctness properties to PRISM and have them checked automatically for us. For example, for attacks like the dectab attack which we know are capable of determining the correct PIN, we would like to be sure that the probability of eventually arriving in a state where all digits are known is 1. This is achieved by checking the following property, as specified in PRISM's syntax for PCTL:

$$\begin{aligned} \mathit{Pmax} =? \quad [\mathit{true}\ U\ (\ &\text{P1\_guessed} \wedge \text{P2\_guessed} \wedge \\ &\text{P3\_guessed} \wedge \text{P4\_guessed})\,] \quad (\dagger) \end{aligned}$$

The $U$ is the (strong) until operator in PCTL. By stating the property with $\mathit{Pmax} =?$, we are asking PRISM not to check the property, but to return the probability that the property holds, i.e. that eventually all digits are guessed. We can further check that the chances of the final PIN being any particular value are exactly 0.0001, or 1 in 10 000. By doing this, we check that our transition probabilities have been calculated correctly to reflect a uniform distribution of PIN values. We can auto-generate a large file containing this property for all possible PINs, and have that checked, or be satisfied with checking each digit individually (which requires only 40 properties). Our models described above do indeed pass these tests.

## *4.2 Optimising Attacks*

Having assured ourselves that our models are correct under our assumption of uniformly distributed PINs, we can proceed to analyse the models for the best attack. For models where the best attack always determines the PIN, the property we are interested in is the expected number of operations required. To determine this in PRISM, we model check the property:

$$\begin{aligned} \mathit{Rmin} =? \quad [F(\ &\text{P1\_guessed} \wedge \text{P2\_guessed} \wedge \\ &\text{P3\_guessed} \wedge \text{P4\_guessed})\{\text{"init"}\}\{\text{min}\}\,] \end{aligned}$$

As used in this context, the $F$ operator specifies that the property must hold at some time in the future with probability 1. By setting $\mathit{Rmin} =?$, we are asking PRISM to return the minimum expected cost required to arrive in a state where all PIN digits are known, starting from the initial state. For some attacks, or combinations of attacks, it may not always be possible to determine the PIN exactly. In these cases, we may be interested in knowing the attack which has the highest probability of obtaining the PIN. This is determined by model checking the property (†) above. We may also be interested in the attack with the highest probability of reducing the range of PINs to some particular size. The latter can be obtained, for some limit on the size of PIN values $k$, by model checking

$$Pmax =? \quad [true\ U\ \text{PIN\_Possibilities} \leq k]$$

Having obtained a measure of the performance of the best attack, we will often want to know exactly what the attack is, i.e. how to perform it step by step. AnaBlock has a small script called 'best-attack' which takes the output from PRISM and produces the attack. The inputs to the script are the list of states, the transition matrix labelled with transition costs, and the cost/reward matrix. All of these can be obtained as output from PRISM, at the same time as model checking is performed, using command line switches. The best attack is then reconstructed by the following algorithm:

```
(1) Let $r$ be the expected cost of the best attack. Let $s$ be the initial state.
(2) If $s$ is an endpoint, stop.
(3) Let $count = 0$
(4) From state $s$, let $T$ be the set of transitions corresponding to non-deterministic
    choice number $count$.
(5) Calculate $r' = \Sigma_{t \in T} P(t).C(t)$, where $P(t)$ is the probability associated
    with transition $t$, and $C(t)$ is the cost associated with $t$.
(6) If $r \neq r'$, increase $count$ by 1, and go to 4.
(7) If $r = r'$, store $count$ as the non-deterministic choice to take at state $s$.
    For every state $s'$ reachable from $s$ by the chosen transitions, recursively
    call the procedure from step 2, with $s = s'$, and $r$ the reward in the
    costs/reward matrix for state $s'$.
```

The output from the script is a file in the format of the graph drawing program 'dot'. An example is given in Figure 4. This is the optimised version of the ISO-0 attack, shown for one PIN digit.

## 5 Results

To evaluate our system we conducted a series of experiments. These experiments were carried out with a generic API, following that set out in [4, §3.3]. Our configuration file has 13 settings (6 commands, 4 block formats, and 3 extra switches to lock-down the PAN, dectab and offset), allowing $2^{13}$ different

| No. | Attack | $P(determined)$ | $E(steps)$ |
|---|---|---|---|
| (1) | ISO-0 (full) | 1 | 13.6 |
| (2) | Dectab | 1 | 16.145 |
| (3) | Dectab & ISO-0 (restricted) | 1 | 15.275 |
| (4) | ISO-0, Check Value & IBM 3624 PINs | 1 | 57.8 |

| No. | 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 1
Experiments with AnaBlock and PRISM. See text (§5 for details)

configurations. For our experiments, we chose 7 typical configurations, deliberately selected to allow different kinds of attack. The results are summarised in Table 1.

(1) This is the most insecure configuration, with all commands and block formats enabled. AnaBlock identifies the most effective attack as the full ISO-0 format attack, described in §2. Note that the performance of this attack, with an expected 3.4 operations per digit, is as good as a binary search on individual digits, making it the ideal attack of this type.
(2) With the PAN locked down, the intruder's best option is the full decimalisation table attack, as explained in §2.2.
(3) The configuration for this experiment allows the PAN to vary, but does not support the VISA-3 block format. The best attack discovered is a combination of the decimalisation table attack with the restricted form of the ISO attack. The restricted form consists of just the first phase, as described in §2. AnaBlock's tree uses these digitwise operations first, then continues with decimalisation table operations. Interestingly, the dectab attack is not significantly improved.
(4) This is a novel variation of the ISO-0 attack, that we discovered when analysing an API with no translation function, but allowing IBM 3624 PIN verification. The original version of the 3624 algorithm allows for the PIN to be calculated based on validation data supplied by the user, that need not be the same as the PAN (some schemes, for example, convert the PAN into validation data by taking the ASCII encoding of each decimal

digit rather than encoding the numbers directly). This gives the intruder the potential to independently vary the PAN and the validation data as inputs to the verification command. Given this ability, the intruder can perform operations like first XORing 1 against the first digit of the PAN, increasing the offset by 1 in its third digit, and repeating the call to the PIN verify command. If the verify call succeeds, then he knows that XORing 1 against the third digit of the PIN decreases it by 1, i.e. the third digit of the PIN has bit 1 set. If the call fails, he knows that bit 1 must be unset. He can repeat this for all bits of the PIN, increasing the offset by 2, by 4, and then by 8, and then repeat the whole procedure for the fourth digit, thereby determining the last two digits of the PIN.

In this experiment, the configuration file also enables the intruder to guess PINs by brute force, by enabling the check value command. The performance of the resulting best attack reflects the fact that the first part of the attack uniquely determines the last two digits in an expected 6.8 operations, and then an expected 51 operations are needed to determine the first two digits (one call to the check value command, then an average of 50 guesses). PRISM determined that the best scheme to carry out the first part of the attack is to look first at the 8-bit, since if this is set, the 4-bit and the 2-bit must be unset, and need not be tested.

(5) This attack is the best available when no decimalisation table operations are available, and only the ISO-0 block format is supported. The $k$ values in this part of the table represent the size of the PIN range. The figures in the table then indicate the probability this attack has of reducing the PIN range to less than or equal to $k$. For the best restricted ISO-0 attack, the PIN range is reduced to 400 with probability 1, but cannot be reduced further.

(6) This attack is the decimalisation table without the offset part. The configuration file has the PAN and offset locked down. There is only a 1 in 1000 chance of determining the PIN exactly (when all the digits are the same), but we can see that the attack will certainly reduce the range to 36 possibilities, and has a good chance of reducing it still further.

(7) The configuration file for this experiment allows the intruder to translate blocks and vary the PAN, but IBM 3624 PIN verification is not supported, and neither is the VISA 3 format. The resulting best attack is a combination of those available in (5) and (6). Having both available significantly improves the best available attack. This is illustrated clearly by the graph in Figure 5. Here, the x-axis shows the values of $k$ supplied to the model checker, i.e. the number of possible PINs, and the y-axis shows the probability of reducing the number of possible PINs to less that or equal to that $k$. Notice that even if the attacker's only way of finally guessing PINs is to do it at an ATM, where he only has 3 chances before the account in blocked, this attack gives him a good chance of success.

The run times for the experiments in Table 1 are shown in Table 2. These times were recorded on a 3.6GHz Pentium IV machine running Linux 2.6.12 and Sun JRE 1.5.0. In general, the more secure the configuration, the fewer options the intruder has, so more secure configurations produce shorter run times. The model requiring the most memory was that for experiment 3, which peaked at around 1.2Gb. Resource requirements vary depending on the Java virtual machine (JVM) used, and the settings supplied to the JVM. A time-space tradeoff can be made if one or the other of these resources is limited (or cheap).

| No. | AnaBlock time | PRISM time | Total time |
|---|---|---|---|
| (1) | 50 mins | 38 mins | 88 mins |
| (2) | 71 sec | 7 mins | 8 mins |
| (3) | 56 sec | 11 mins | 12 mins |
| (4) | 3 sec | 43 sec | 46 sec |
| (5) | 3 sec | 14 sec | 17 sec |
| (6) | 10 sec | 2 mins | 2mins |
| (7) | 25 sec | 16 mins | 16 mins |

Table 2
Runtimes for the experiments

Our results give some new insights into the attacks, and their combination. The result from experiment (7) is particularly interesting. Here, we can see what would happen if an API designer decided to lock down the offset, and chose to support only PIN block formats which don't allow the full ISO-0 attack. The system as a whole is still highly vulnerable.

The AnaBlock source code and the files used for the experiments in Table 1 are available from `http://homepages.inf.ed.ac.uk/gsteel/AnaBlock`.

## 6 Related Work

Most known API attacks are due to Bond, [1], and Clulow, [4]. Previous work on automated analysis of APIs has looked at key-management schemes rather than PIN-processing, e.g. [9,6,13,14]. However, there has been some work on probabilistic analysis of other security protocols. For example, Shmatikov analysed the 'Crowds' protocol for ensuring anonymity using PRISM, [12]. This protocol works by routing messages through a network in a random fashion, thereby preventing even a group of colluding attackers from establishing the origin of messages. Shmatikov was able to find a previously unknown flaw: as the group size increases, the attacker's confidence in the identity of the sender also increases.

There is also related work in the field of guessing attacks, e.g. [10,5]. The major difference between our work and previous work in this area is that we consider 'online' guessing rather than 'offline', where terms are merely considered to be either guessable or not guessable. Our analysis considers the complexity of guessing particular terms, and the effect of the HSM's responses on that complexity.

Our approach has some similarities to the automated generation and analysis of attack graphs, [11]. In this work, the level of abstraction is much higher, with nodes representing tasks that the intruder must perform, like effecting a buffer overflow attack. Tasks are assigned a probability of success, and graphs are checked to establish the overall chance of a successful intrusion. There may be scope for some crossover here: our ideas for the use of costs to measure the difficulty of an attack could be used to enhance attack graphs, or our attack trees could be included as subgraphs in larger attack graphs used to represent the process by which an intruder might gains access to a running HSM, or obtain a fake card.

## 7 Further Work

The main focus of our future work will be to develop automated support for the generation of the API specification file. This will involve choosing a specification language for the API designers to describe the operation of their functions, and then analysing these specifications to produce a set of all possible operations the intruder may call. This is quite an ambitious goal, but it presents an interesting challenge, and may allow us to discover previously unknown attacks. By hand-coding these specification files we have already discovered a novel variant of an attack (in experiment 4, above §4), which leads us to suspect that a comprehensive, automated approach to analysing these operations would lead to the discovery of many more variations.

Another area for development is the breaking of symmetry in our PRISM models. We already try digitwise operations in order, as described in §3. For the decimalisation table attack, there is still some scope for breaking symmetries. After a correct offset is discovered, the digits that are set to the dectab value could be shifted to the left and all remaining unknown digits shifted to the right. This would have no effect on the complexity analysis, but would cut much duplication from the model. Care would have to be taken when combining this with other attacks, however.

Our model as it stands abstracts away some detail from the attacks. For example, the HSM will not allow an attacker to use an account number digit which is hexadecimal, which is required in the ISO-0 attack. In reality, depending on the account number in use, he may have to make 2 XOR operations, to construct the required modification value. The final outcome will be the same, but more operations may be required. To analyse this, we would also have to build a model of uniformly distributed account number digits, which seems over-complex. Future work will try to address this issue.

## 8 Conclusions

We have presented a framework for the analysis on PIN block attacks, based on constructing models with a mixture of non-deterministic and probabilistic choices, and using a probabilistic model checker to find the most effective attack. This framework has been used to analyse 3 families of attacks in several variations and combinations. This ability to combine and vary attacks, without having to make changes to the model generation software, is a key advantage of our approach. Our use of constraint logic programming made prototyping the system a relatively simple job. In particular, it allowed us to easily generate the correct transition probabilities for the model.

Our framework seems to provide a good solution to the problem of finding and analysing the best attack once the potential capabilities of the intruder have been established. The focus for our future work is to automate the generation of these capabilities for a particular API specification.

## Acknowledgements

The author is grateful for instructive conversations with: Nicko van Someren, Jon Geater, and Ron Carter at nCipher, Mike Bond and Jolyon Clulow at the University of Cambridge, and Marta Kwiatkowska and Dave Parker at the University of Birmingham. The anonymous reviewers also made several helpful suggestions for improving the paper.

## References

[1] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, pages 67–75, October 2001.

[2] M. Bond and P. Zieliński. Decimalisation table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, January 2003.

[3] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, pages 191–206, Southampton, UK, September 1997.

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

[5] P. Drielsma, S. Mödersheim, and L. Viganò. A formalization of off-line guessing for security protocol analysis. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *LNAI*, pages 363–379. ETH Zürich, Computer Science, Springer, March 2005.

[6] 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.

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

[8] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 322–323. IEEE Computer Society Press, 2004.

[9] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[10] G. Lowe. Analysing protocol subject to guessing attacks. *Journal of Computer Security*, 12(1):83–98, 2004.

[11] 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*, pages 273–284, Berkeley, California, May 12-15 2002.

[12] V. Shmatikov. Probabilistic analysis of anonymity. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 119–128, Washington, DC, USA, 2002. IEEE Computer Society.

[13] 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 Lecture Notes in Artificial Intelligence, pages 322–336, Tallinn, Estonia, July 2005. Springer-Verlag Heidelberg.

[14] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.

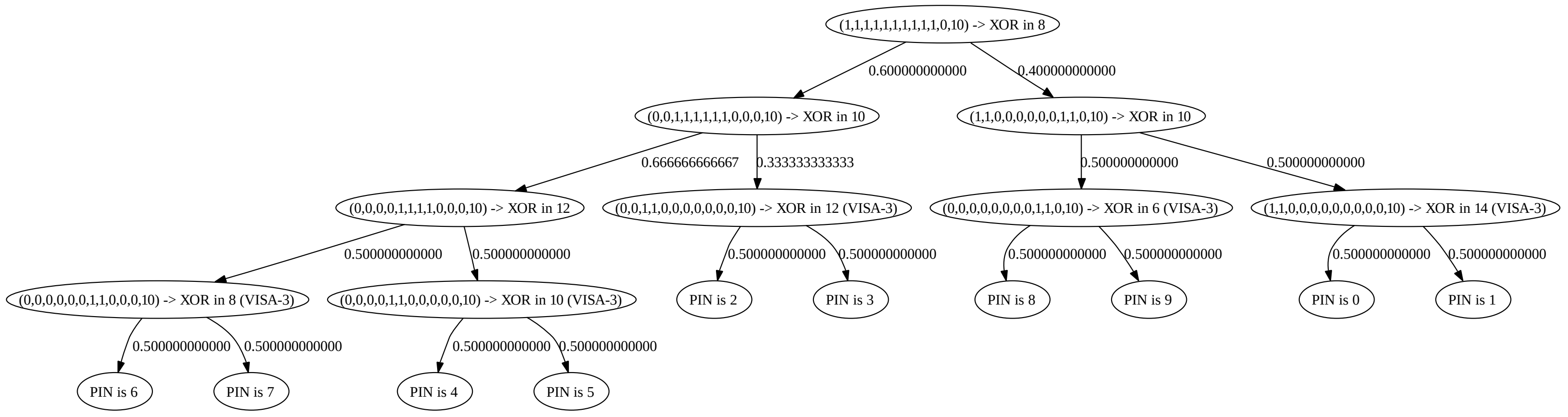
(1,1,1,1,1,1,1,1,1,0,10) -> XOR in 8
0.600000000000
0.400000000000
(0,0,1,1,1,1,1,1,0,0,0,10) -> XOR in 10
(1,1,0,0,0,0,0,0,1,1,0,10) -> XOR in 10
0.666666666667
0.333333333333
0.500000000000
0.500000000000
(0,0,0,0,1,1,1,1,0,0,0,10) -> XOR in 12
(0,0,1,1,0,0,0,0,0,0,0,10) -> XOR in 12 (VISA-3)
(0,0,0,0,0,0,0,0,1,1,0,10) -> XOR in 6 (VISA-3)
(1,1,0,0,0,0,0,0,0,0,0,10) -> XOR in 14 (VISA-3)
0.500000000000
0.500000000000
0.500000000000
0.500000000000
0.500000000000
0.500000000000
0.500000000000
0.500000000000
(0,0,0,0,0,0,1,1,0,0,0,10) -> XOR in 8 (VISA-3)
(0,0,0,0,1,1,0,0,0,0,0,10) -> XOR in 10 (VISA-3)
PIN is 2
PIN is 3
PIN is 8
PIN is 9
PIN is 0
PIN is 1
0.500000000000
0.500000000000
0.500000000000
0.500000000000
PIN is 6
PIN is 7
PIN is 4
PIN is 5

Fig. 4. Optimised ISO-0 Attack (1 digit)

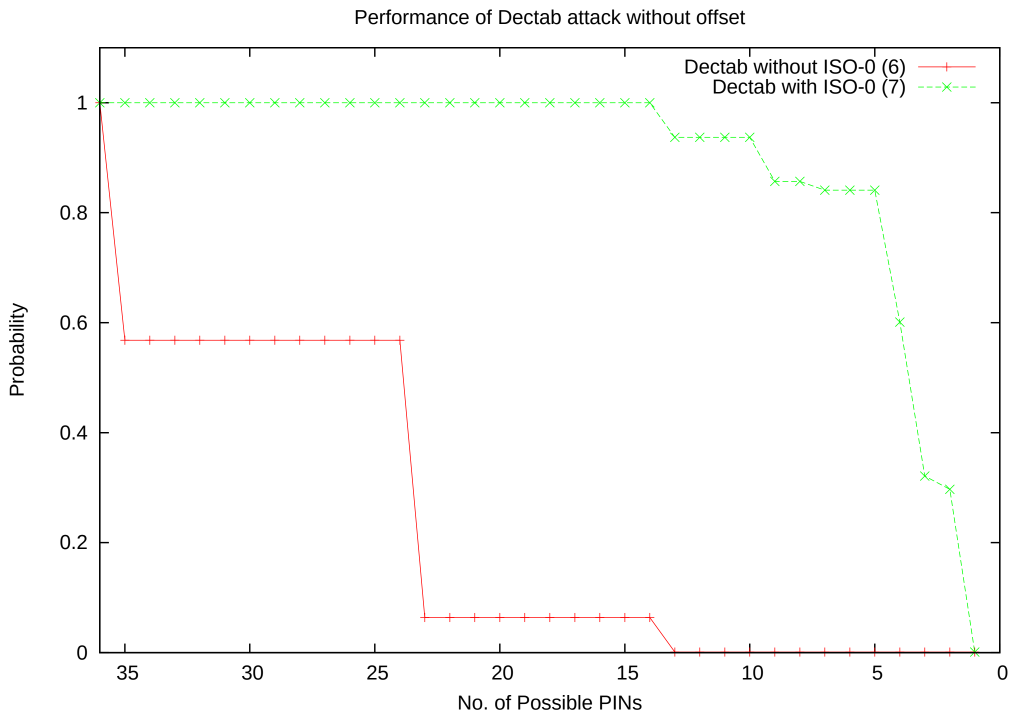

Fig. 5. Attacks (6) and (7)