# Modeling Partial Attacks with Alloy

Amerson Lin[1], Mike Bond[2], and Jolyon Clulow[2]

[1] Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
77 Massachusetts Avenue, Cambridge MA 02139
firstname.lastname@alum.mit.edu
[2] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
firstname.lastname@cl.cam.ac.uk

**Abstract.** The automated and formal analysis of cryptographic primitives, security protocols and *Application Programming Interfaces* (APIs) up to date has been focused on discovering attacks that completely break the security of a system. However, there are attacks that do not immediately break a system but weaken the security sufficiently for the adversary. We term these attacks *partial attacks* and present the first methodology for the modeling and automated analysis of this genre of attacks by describing two approaches. The first approach reasons about entropy and was used to simulate and verify an attack on the ECB|ECB|OFB triple-mode DES block-cipher. The second approach reasons about possibility sets and was used to simulate and verify an attack on the *personal identification number* (PIN) derivation algorithm used in the IBM 4758 Common Cryptographic Architecture.

## 1 Introduction

Consider, if you will, a protocol attack that exposes a secret such as a key or password. Often we expect a eureka moment, a single point in time during the attack when, with rapturous applause, the secret is completely revealed. It is clear now that the protocol is broken — the obvious security goal of key confidentiality has been violated.

However, there exists a range of more subtle attacks that result in less clear consequences. These attacks do not immediately compromise a specific security goal but instead weaken the system's security with respect to the goal. We term these attacks *partial attacks*, in contrast with the more traditional attacks we call *complete attacks*.

A partial attack can manifest itself in different ways. An attack may reveal a small amount of information such as a few bits of a key or reduce the effective key length as in the case of the meet-in-the-middle attack on 2DES [1]. The bottomline is that the effective entropy, or randomness, protecting the system is significantly reduced.

We make the distinction between complete and partial attacks in order to highlight the comparatively more challenging problem of automating the discovery of

B. Christianson et al. (Eds.): Security Protocols 2007, LNCS 5964, pp. 20–33, 2010.

partial attacks with formal analysis tools. Complete attacks can be modeled by describing the assumptions, the abilities and the goals. Partial attacks, on the other hand, are difficult to model, or even simply to identify, for three reasons. First, a reduction in entropy is non-obvious and often not captured in any security goal. Second, it is not even clear how to describe the corresponding information leakage in a formal language. Thirdly, reconstructing the secret from leaked information may be non-trivial. Nevertheless, in this paper, we show some initial progress on modeling partial attacks by demonstrating two techniques.

### 1.1 Our Contributions

Our first approach reasons about the entropy relationships of secret information. We model an attack as any sequence of steps that reaches a recognisable state known to correspond to a reduced measured of entropy. We used this strategy to verify a cryptanalytic attack, discovered by Biham [2] in 1996, on the ECB|ECB|OFB triple-mode DES block-cipher.

The second approach models the secret as an element in a large set of possible attacker guesses called the possibility set. An attack is any sequence of operations that reduces the cardinality of the possibility set faster than an exhaustive search. This was used to verify the PIN decimalisation attack found by Clulow [3] and independently by Bond and Zielinski [4] on the IBM 4758 *Cryptographic Common Architecture* (CCA).

In the development of these techniques, we used the Alloy modeling language and Alloy Analyzer automated reasoning tool to model and simulate the two aforementioned attacks [5]. The language and tool's powerful support for sets and set operations was well suited for the reasoning we required. We believe these techniques can be generalised to other formal tools.

## 2 Partial Attacks

The brute-force search for an $n$-bit key is an attack requiring $2^n$ effort on average, where $n$, the bit length of the key, is the security parameter. An equivalent measure of the cryptosystem's security is the *entropy*, $H$, of the key. Formally, let $K$ be a discrete random variable representing a cryptosystem's key, having a uniform probability distribution over $k_1, k_2, \ldots, k_{2^n}$. The information entropy, or Shannon entropy [6], of $K$, is:

$$H(K) = -\sum_{i=1}^{2^n} p(k_i) log_2 \left( \frac{1}{p(k_i)} \right) = log_2(2^n)$$

where $p(k_i) = Pr(K = k_i)$. The second equality follows from the Jensen inequality, where entropy is maximal when all keys are equiprobable. In short, a cryptosystem with a 64-bit key has an entropy of 64 bits.

A partial attack may be defined as a sequence of steps resulting in a "sufficiently large" rate of reduction in the entropy protecting a stated security goal

of a cryptosystem, where "sufficiently large" is subject to further definition but should be more than the rate of information gained from a brute-force search, which is $log_2(2^n - i + 1) - log_2(2^n - i)$ for the $i$th try.

### 2.1 Entropy and Relationships

In this technique, we desire to reason directly about measurements of information entropy. Ideally, we want our model and tool to reason about absolute or relative entropy measurements of secrets after various operations. We can then analyse a sequence of operations by making sense of the net effect on entropy. However, there are technical challenges when applying this idea to formal methods and tools. Formal modeling (almost) exclusively uses the Dolev-Yao model [7] where cryptographic primitives are treated as perfect. Keys are atomic entities and are not considered as bit-streams of a certain length. How does one then create a measure of the absolute entropy at a finer level of granularity than an individual key? The next challenge is to associate a cumulative measure of entropy with a sequence of operations. Current tools do not have an obvious or easy method to facilitate this.

To work around these challenges, our solution reasons about relative entropy by making sense of input/output (I/O) relationships. I/O relationships can range from linearly-dependent output to completely random output. By describing I/O relationships that represent a reduction in entropy, an attack is then a sequence of events that delivers a particular recognisable I/O characteristic representing a recognisable but unquantifiable amount of entropy. We used this technique to analyze the ECB|ECB|OFB triple-mode block-cipher, which will be described in greater detail in section 4.

### 2.2 Possibility Sets

This technique adopts a contrasting approach by dealing with entropy directly at the finite set space level: we start with a possibility set containing all the possible guesses for the key and eliminate incorrect guesses as we progress. This involves modeling APIs using variables with values that belong to the set. Information leakage is observed if the number of possibilities reduces more quickly than with a brute-force search.

There are some obvious limitations of this approach. The first is the cardinality of the possibility set. Directly instantiating the possibility set of even a small 56-bit key is infeasible. The obvious solution is to abstract our models of systems by using reduced size parameters. The size of the possibility set is irrelevant if all we are required to show is that an attack reduces the size of the set faster than a brute-force attack does. The level of simplification required is governed by a particular tool's ability to work with sets. This includes the generation of the initial possibility set and the ability to manipulate elements in the set. Again, our choice of the Alloy modeling language and its related automated reasoning tool for their powerful support for sets proved worthy when we attempted to model and simulate the PIN decimalisation attack on IBM's 4758 architecture, which is detailed in section 5.

Before describing in detail how these techniques were used to simulate the attacks, we must first introduce our tools — the Alloy language and the Alloy Analyzer.

## 3 Alloy Overview

Alloy is a lightweight modeling language based on *first-order relational logic* [8], designed primarily for modeling software design. The Alloy Analyzer [9] is a model-finder taking descriptions written in the Alloy language. Hereafter, we loosely use Alloy to refer to both the language and the tool.

Alloy uses first-order relational logic, where reasoning is based on statements written in terms of atoms and relations between atoms. Any property or behaviour is expressed as a constraint using set, logical and relational operators. The language supports typing, sub-typing and compile-time type-checking, giving more expressive power on top of the logic.

The relational operators require a little more treatment and are described below. Let $p$ be a relation containing $k$ tuples of the form $\{p_1, \cdots, p_m\}$ and $q$ be a relation containing $l$ tuples of the form $\{q_1, \cdots, q_n\}$.

- `p -> q` – the *relational product* of $p$ and $q$ results in a new relation $r = \{p_1, \cdots, p_m, q_1, \cdots, q_n\}$ for every combination of a tuple from $p$ and a tuple for $q$ ($kl$ pairs).
- `p .q` – the *relational join* of $p$ and $q$ is the relation containing tuples of the form $\{p_1, \cdots, p_{m-1}, q_2, \cdots, q_n\}$ for each pair of tuples where the first is a tuple from $p$ and the second is a tuple from $q$ and the last atom of the first tuple matches the first atom of the second tuple.
- `^p` – the *transitive closure* of $p$ is the smallest relation that contains $p$ and is *transitive*. Transitive means that if $p$ contains $(a, b)$ and $(b, c)$, it also contains $(a, c)$. Note that $p$ must a binary relation and that the resulting relation is also a binary relation.
- `*p` – the *reflexive-transitive closure* of $p$ is the smallest relation that contains $p$ and is both transitive and *reflexive*, meaning that all tuples of the form $(a, a)$ are present. Again, $p$ must be a binary relation and the resulting relation is also a binary relation.
- $\sim$`p` – the *transpose* of a relation $r$ forms a new relation that has the order of atoms in its tuples reversed. Therefore, if $p$ contains $(a, b)$ then $r$ will contain $(b, a)$.
- `p <:q` – the *domain restriction* of $p$ and $q$ is the relation $r$ that contains those tuples from $q$ that start with an atom in $p$. Here, $p$ must be a set. Therefore, a tuple $\{q_1, \cdots, q_n\}$ only appears in $r$ if $q_1$ is found in $p$.
- `p >:q` – the *range restriction* of $p$ and $q$ is the relation $r$ that contains those tuples from $p$ that end with an atom in $q$. This time, $q$ must a set. Similarly, a tuple $\{p_1, \cdots, p_m\}$ only appears in $r$ if $p_m$ is found in $q$.
- `p++q` – the *relational override* of $p$ by $q$ results in relation $r$ that contains all tuples in $p$ except for those tuples which first atom appears as the first atom in some tuple in $q$. Those tuples are replaced by their corresponding ones in $q$.

The Alloy Analyzer is a model-finder that tries either to find an *instance* of the model or a *counter-example* to any specified property assertions. An instance is literally an instantiation of the atoms and relations in the model specification. It performs a bounded-analysis of the model by requiring a user-specified bound on the number of atoms instantiated in the model. With this bound, it translates the model into a *boolean satisfiability* (SAT) problem. Then, it hands the SAT problem off to a commercial SAT solver such as Berkmin [10]. The resulting solution is then interpreted under the scope of the model and presented to the user in a graphical user-interface.

## 4 Triple-Mode DES

Our first technique, which involves reasoning about entropy, was used to successfully verify an attack on the ECB|ECB|OFB triple-mode DES block-cipher, which was previously discovered by Biham [11] in 1994. In the ECB|ECB|OFB triple-mode, the plaintext undergoes two rounds of *Electronic Code Book* (ECB) encryption before an XOR with an *Output Feedback* (OFB) value. The OFB stream is generated by repeatedly encrypting an initial value. The schematic for the ECB|ECB|OFB triple-mode is shown in Figure 1.

Let us consider the cryptanalytic attack on this triple-mode of operation, as presented by Biham. Assuming that the block size is 64 bits, by providing the same 64-bit value, $v$, to all $2^{64}$ blocks, the two ECB layers would produce the same encrypted value presented to the OFB layer, thereby allowing us to

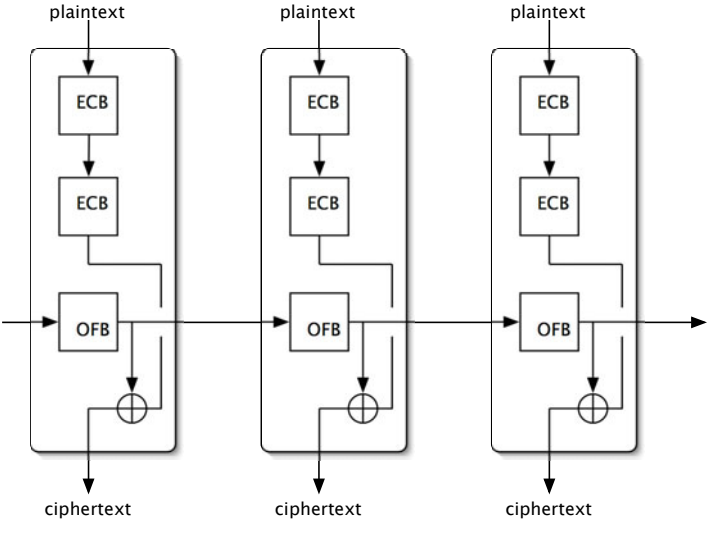


**Fig. 1.** ECB|ECB|OFB triple-mode

separate the OFB layer of encryption by obtaining $2^{64}$ ciphertext values, where each ciphertext

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

where $o_i$ is the OFB stream value such that $o_{i+1} = \{o_i\}_{K_{OFB}}$ and *K1* and *K2* are the 2 ECB encryption keys. This is equivalent to having a complete OFB stream XOR with a single value $\{\{v\}_{K1}\}_{K2}$. Isolating the OFB layer allows us to break the OFB key independently, an effort requiring $2^{64}$ amount of work. Subsequently, the two ECB layers can be broken with a meet-in-the-middle attack [1]. To break the OFB layer, we do the following:

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

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

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

where $(c_i, c_{i+1}, c_{i+2})$ is a consecutive triple of ciphertexts and $(o_i, o_{i+1}, o_{i+2})$ is a consecutive triple of OFB stream values.

### 4.1 Alloy Model of ECB|ECB|OFB Attack

To describe the ECB|ECB|OFB triple-mode, we need to model encryption, the XOR operation, the various blocks and the input/output relationships between the blocks. To model encryption, we define a set of `Value` and a set of `Key` atoms. We also define relations `xor` and `enc` as fields, represented as tuples of the form `(Value,Value,Value)` and `(Key,Value,Value)` respectively. Finally, we define two singleton sets `K1, OFB` of type `Key`, representing the combined ECB—ECB key and the OFB key respectively. All these are expressed with the following:

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

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

one sig OFB, K1 extends Key {}
```

We use the `enc` relation to represent the encryption function: a value, encrypted under a key, results in some value. The `one -> one` multiplicity constraint states that the binary relation `(Value,Value)` for each key is a one-to-one function. In this relation, the tuple $(k_1, v_1, v_2)$ reads: $v_1$ encrypted under $k_1$ gives $v_2$ and this generalizes to the representation *(key,plaintext,ciphertext)*[1]. Although the

[1] The encryption function can be expressed in many other equivalent forms, such as `(Value,Key,Value)`.

encryption property has been sufficiently defined, we sometimes need to add certain constraints to prevent ALLOY from producing trivial instances or counterexamples. In our model, we enforce that a particular value does not encrypt to itself and that two different keys produce different encryption relations.

```
fact EncFacts {
   all k:Key, b:Value | k.enc[b] != b
   all disj k,k':Key | some (k.enc - k'.enc)
}
```

To model the XOR operation, we only need to define the three properties of XOR:

1. The *commutativity* property states that $v_1 \oplus v_2 = v_2 \oplus v_1$ for any $v_1, v_2$.

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

   By saying that `xor.b` is equal to its transpose, we succinctly capture the required property.

2. The *associativity* property states that $(v_1 \oplus v_2) \oplus v_3 = v_1 \oplus (v_2 \oplus v_3)$ for any $v_1, v_2, v_3$.

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

3. The *identity* is the special value such that $identity \oplus v = v$ for any $v$.

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

   The `iden` constant represents the universal *identity relation*, which is a binary relation containing tuples of the form $(a, a)$. Therefore, the identity statement says that the identity XOR relation contains only identity tuples.

Now, let us proceed to model the DES blocks and their input/output behaviour. We define a set of blocks `Block`, where each block contains a plaintext `p`, an OFB value `ofb` and a ciphertext `c`. Next, we impose an ordering on the blocks with the general `util/ordering` module packaged in ALLOY. The module also provides operations (`first, last, next[x], prev[x]`) to help us navigate through the ordering: `first` and `last` return the first and last elements of the ordering while `next[x]` and `prev[x]` return the element after or before the element `x`.

```
open util/ordering[Block] as ORDER

sig Block {
  p: one Value, // plaintext
  ofb: one Value, // OFB value
  c: one Value // ciphertext
}
```

Next, we proceed to describe the data flow within these blocks. A block's output ciphertext is the result of the encryption of its plaintext under the combined ECB—ECB key `K1`, followed by an XOR with the block's OFB value. For all blocks $b$ except the last, the OFB value of the next block $b'$ is the encryption $b's$ OFB value with the OFB key.

```
fact {
  all b: Block | b.c = (K1.enc[b.p]).xor[b.ofb]

  all b: Block-last | let b' = next[b] |
  b'.ofb = OFB.enc[b.ofb] }
```

Lastly, we describe the condition required to identify the successful isolation of the OFB layer — the essence of Biham's attack — by specifying the property on the resulting ciphertexts as a predicate. We then ask the Alloy Analyzer to find an instance, within a scope of 8 atoms (a power of two), where the predicate is true.

```
pred FindAttack() {
  all b':Block-first-last | let b=prev[b'], b''=next[b'] |
  some d:OFB.enc.Value | let d'=OFB.enc[d], d''=OFB.enc[d'] |
  b.c.xor[b'.c] = d.xor[d'] && b'.c.xor[b''.c] = d'.xor[d'']
}

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

The full model can be found in Appendix A. Alloy was able to simulate the attack in 10 seconds[2], attesting to Alloy's power in dealing with relational properties. However, it is important to note that the same ciphertext property in our model could have been achieved with many different sets of input plaintexts. Since Alloy is non-deterministic, it could have found any one of them.

## 5 PIN Decimalisation

Our second technique, which involves possibility sets, was used to model a *personal identification number* (PIN) guessing attack discovered by Clulow [3] and independently by Bond and Zielinski in 2003 [4]. They found that the IBM 4758 security module used *decimalisation tables* to generate PINs and that made it susceptible to what they termed the *PIN-decimalisation* attack, an attack allowing the adversary to guess the customer's PIN in ∼15 tries, thereby subverting the primary security mechanism against debit card fraud. The attack targets the IBM PIN derivation algorithm, which works as follows:

1. calculate the original PIN by encrypting the *primary account number* (PAN) with a secret "PIN generation" DES key

[2] Alloy Analyzer version 4.0 RC 11 on a 2GHz PC, 1GB RAM.

2. convert the resulting ciphertext into hexadecimal
3. convert the hexadecimal into a PIN using a "decimalisation table". The actual PIN is usually the first 4 or 6 numerals of the result.

The standard decimalisation table used is shown below:

```
0123456789ABCDEF
0123456789012345
```

During decimalisaton, the hexadecimal values on the top line are converted to the corresponding numerical digits on the bottom line. The IBM command for verifying a PIN created in this way is `Encrypted_PIN_Verify`, which allows the programmer to specify the decimalisation table, the PAN and an encrypted PIN. The encrypted PIN is first decrypted and then compared to the PIN generated from the PAN using the generation algorithm described earlier. The command returns true if the PINs match, false otherwise. We have no direct access to the input PIN encryption key, but we assume we can inject trial PINs, for instance by entering them at an automated teller machine.

The attack can be broken down into two stages: the first determines which digits are present in the PIN and the second determines the true PIN by trying all possible digit combinations. Had the decimalisation been hard-coded, the attack could have been avoided. We describe the first stage of the attack on a simplified system — one with a single-digit PIN. To determine the digit present in the PIN, we pass in the desired (victim's) PAN, an encrypted trial PIN of 0 and a *binary* decimalisation table, where all entries are 0 except the entry for hex-digit $i$, which is 1. The trial PIN of 0 will fail to verify only if the pre-decimalised PIN (or encrypted PAN) contains hex-digit $i$.

### 5.1 Alloy Model of PIN Decimalisation Attack

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

```
open util/ordering[State] as ORDER

sig NUM{}

sig HEX{}
```

```
one sig TrueHEX extends HEX {}    // correct pre-decimalised PIN

sig State {
  trialPIN: one NUM,
  dectab: HEX -> one NUM ,
  guess: set HEX
}
```

Next, we model the event of the attacker choosing a particular number as a trial PIN. We use a predicate to constrain the possibility set `guess` in two states $s, s'$ based on the choice of trial PINs in state $s$. If the trial PIN does verify, then the pre-decimalised PIN must contain the hexadecimal values corresponding to the trial PIN in the given decimalisation table, and vice versa. In either case, the attacker is gaining knowledge about the value of the pre-decimalised PIN. We use the intersect operation to express this.

```
pred choose[s,s': State] {
( (s.trialPIN = s.dectab[TrueHEX]) &&
  (s'.guess = s.guess & s.dectab.(s.trialPIN)) )
||
( (s.trialPIN != s.dectab[TrueHEX]) &&
  (s'.guess = s.guess & s.dectab.(NUM-(s.trialPIN))) )
}
```

Finally, we set up the attack by defining transition operations across the states. Here, we define a predicate `init` on a state $s$ to initialise the attacker's possibility set. The predicate `Attack` simulates the attack as the adversary makes the command calls with different decimalisation tables and trial PIN choices. The predicate defines a relation between each state and the next, with the last ending in success. It also says that the attacker can do better than a brute-force attack in at least one command call, a proxy for information leakage. The last two constraints are necessary in order to produce non-trivial instances. Without them, our attacker would simply succeed on the very first try, by luck.

```
pred init[s:State] {
  s.guess = HEX
}

pred Attack () {
  init[first]
  all s:State-last() | let s'=next(s) |  choose(s,s')
  some s:State-last | let s'=next[s] |
       (#(s.guess) - #(s'.guess)) > 1
  #(last().poss) = 1
}

run Attack for 16 but exactly 10 NUM
```

The full model can be found in Appendix B. ALLOY was able to find a non-trivial attack sequence in a few seconds[3], but it did not choose the decimalisation table as we did for maximum efficiency [4]. It is challenging to request that ALLOY find the fastest attack with average luck since ALLOY was designed to find *any* instance and not a *particular* one.

## 6 Conclusion

In this paper we raise the above concepts as starting points for formal modeling and demonstrate how we employed these strategies to model a number of attacks that have not been amenable to traditional theorem proving or model checking tools. In particular, we demonstrated the modeling of a cryptanalytic attack on triple-mode DES and the leakage of PINs from a banking security module. We do acknowledge that our models were constructed with an attack in mind. Nevertheless, our techniques of reasoning about the entropy of input/output relationships and of using possibility sets certainly allow for unusual attacker goals to be described. Although we do not present any new attack discoveries, we believe that our work serves as a proof of concept of how research in this field can proceed.

We end by suggesting an area for exploration — an approach we call *ability-based modeling*. In this approach, the attacker is given an unusual ability and the target cryptosystem's security is modeled with regard to this ability. Using triple-mode DES again as an example, analysing the cipher's security against an adversary with the ability to break 64-bit keys could yield interesting results. In some cases, this actually reduces a partial attack to a complete attack.

## References

1. Diffie, W., Hellman, M.: Exhaustive cryptanalysis of the NBS data encryption standard. Computer 10(6), 74–84 (1977)
2. Biham, E.: Cryptanalysis of triple-modes of operation. Technical Report, Computer Science Department, Technion (CS0885) (1996)
3. Clulow, J.: The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal (2003)
4. Bond, M., Zielinski, P.: Decimalisation table attacks for PIN cracking. Technical Report, University of Cambridge (560) (2003)
5. Lin, A.: Automated analysis of security APIs. Master's thesis, Massachusetts Institute of Technology (2005)
6. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal 27, 379–423, 623-656 (1948)
7. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29 (1983)
8. Jackson, D.: Alloy 3.0 Reference Manual. Software Design Group (2004)
9. Software Design Group, MIT (2004), `http://alloy.mit.edu`
10. Goldberg, E., Novilov, Y.: Berkmin: A fast and robust SAT-solver (2002)
11. Biham, E.: Cryptanalysis of multiple modes of operation. Technical Report, Computer Science Department, Technion (CS0833) (1994)

[3] ALLOY ANALYZER version 4.0 RC 11 on a 2GHz PC, 1GB RAM.

## A Triple-Mode DES Alloy Model

```
/* Model for cryptanalytic attack on the ECB|ECB|OFB
   triple-mode DES block cipher.  In this model, the
   two ECB layers are combined into a single layer.

   note: Tested on Alloy Analyzer 4.0 RC 11.
   Since Alloy is non-deterministic, running this model
   on different computers may produce different results.

*/

module ECBOFB
open util/ordering[Block] as ORDER

// set of values (e.g. 64-bit values)
sig Value {
  xor : Value one -> one Value
}

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

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

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

// set of keys
sig Key{
  // encryption relation
  enc: Value one -> one Value
}

fact EncFacts {
  // no value encrypts to itself
  all k:Key, b:Value |  k.enc[b] != b

  // every encryption is different
  all disj k,k':Key | some (k.enc - k'.enc)
}

one sig OFB, K1 extends Key {}
```

```
// set of Blocks,
// p=plaintext, ofb=OFB sequence, c=ciphertext
sig Block {
  p: one Value,
  ofb: one Value,
  c: one Value
}

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

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

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

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

## B PIN Decimalisation Alloy Model

```
// Model for PIN decimalisation attack on IBM 4758.

module PIN
open util/ordering[State] as ORDER

sig NUM{}         // set of numbers

sig HEX{}         // set of hexadecimal digits

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

sig State {
```

```
  trialPIN: one NUM,          // attacker picks a trialPIN
  dectab: HEX -> one NUM ,    // and a decimalisation table
  guess: set HEX              // attacker's possibility set
}

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

pred init [s:State] {
  // attacker's initial possibility set is all hexadecimal values
  s.guess = HEX
}

pred Attack () {
  // initialise on the first state atom
  init[first]

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

  // better than brute-force by choosing an appropriate
  // decimalisation table
  some s:State-last | let s'=next[s] |
              (#(s.guess) - #(s'.guess)) > 1

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

run Attack for 16 but exactly 10 NUM
```