A Formal Model for Detecting Parallel Key Search Attacks

Graham Steel and Judicaël Courant

June 1, 2007

Abstract

Parallel key search or meet-in-the-middle attacks have been shown to be effective against a number of security APIs, designed to regulate access to tamper resistant hardware devices. However, they are outside the scope of standard Dolev-Yao style modelling techniques. In this paper, we present an extension to the Dolev-Yao model that allows these kinds of attacks to be detected. It is work in progress: we need to introduce quantitative aspects to the analysis, and we show why existing quantitative extensions to the Dolev-Yao calculus are inadequate for this.

1 Introduction

The purpose of a security application program interface (API) is to allow untrusted code to access sensitive resources in a secure way. Tamper-proof hardware security modules (HSMs), for example, have security APIs which control access to the cryptoprocessor and memory inside the module. This allows the API to manage access to cryptographic keys. HSMs are deployed in security critical environments such as the cash machine network, where they are used to protect customers PINs and other sensitive data. They typically consist of a cryptoprocessor and a small amount of memory inside a tamper-proof enclosure. They are designed so that should an intruder open the casing or drill into it to insert probes, the memory will auto-erase in a matter of nanoseconds. In a typical ATM network application, all encryption, decryption and verification of PINs takes place inside the HSM. Security APIs typically manage keys by keeping a secret master key inside the device. This is used to encrypt all the 'working keys' used for operational functions, so that they can be securely stored outside the device.

One technique used by attackers attempting to breach security is to generate a large number of different working keys of a particular type, and to obtain some known value encrypted under all these keys. This can often be achieved by using a *key test* command provided by the API. The attacker can then encrypt this known value under random values himself until he gets a match with one of his generated keys. Note that this is far less costly than brute-force cracking one particular key. The result is a pair consisting of a key, and that key encrypted under the master key. On its own this pair is useless, but sometimes it can be used to obtain the value of other keys, such as the security-critical PIN derivation keys used to obtain customer PINs from account numbers. However, examining an API to determine whether such an attack is possible is not an easy task. At first sight it may appear that only unimportant keys can be cracked in this way. The manual discovery of an attack like the one given in [3] takes considerable work.

A promising approach to automating security API analysis involves adapting Dolev-Yao style protocol analysis techniques [7], where details of cryptographic algorithms used are abstracted away, and a logical model is constructed, with rules describing the operations of the intruder and protocol. This can be adapted quite naturally to API analysis by considering the API to be a set of 2-party protocols, each describing an exchange between the secure hardware module and the host machine [8, 9, 6]. However, previous work has not taken parallel key search attacks into account. The aim of the work in this paper is to address this problem, by proposing a formal model that allows operations corresponding to computationally feasible parallel key search attacks to be incorporated into a Dolev-Yao style model for security analysis of the API. When combined with the usual operations of the Dolev-Yao intruder, it should enable us to determine whether a parallel key search attack can be used to obtain a security-critical key. Our formalism is based on the one given in [4], where *key conjuring*, another 'non-Dolev-Yao'

operation, was incorporated into an API model. Both tricks may be necessary for some known attacks, e.g. [3].

This is a 'work in progress paper': in order for our formalism to distinguish between feasible and infeasible attacks, we will need to introduce quantitative aspects into the model, which should reflect results in computational cryptography. We provide only a sketch of our treatment here.

2 Notation

Our (mostly standard) notation for API models, including the mechanism for modelling key conjuring described in full in [4] and [5]. We summarise it here.

Cryptographic primitives are represented by functional symbols. More specifically, we consider a *signature* Σ which consists of an infinite number of constants including a special constant 0 and three non constant symbols $\{\_\}\_$ (encryption), dec (decryption) and $\oplus$ (XORing) of arity 2. We also assume an infinite set of variables $\mathcal{X}$. The set of *terms*, denoted by $\mathcal{T}(\Sigma, \mathcal{X})$, is defined inductively by

$$\begin{array}{rcll} T & ::= & & \text{terms} \\ & & x & \text{variable } x \\ & | & f(T_1, \ldots, T_n) & \text{function application} \end{array}$$

where f ranges over the functions of Σ and n matches the arity of f. For instance, the term $\{m\}_k$ is intended to represent the message m encrypted with the key k (using symmetric encryption) whereas the term $m_1 \oplus m_2$ represents the message m_1 XORed with the message m_2. The constants may represent control vectors or keys for example.

We equip the signature Σ with an equational theory $\mathsf{E_{API}}$ that models the algebraic properties of our operators:

$$\mathsf{E_{API}} := \left\{ \begin{array}{ll} \{\mathsf{dec}(x,y)\}_y = x & x \oplus 0 = x \\ \mathsf{dec}(\{x\}_y, y) = x & x \oplus x = 0 \\ x \oplus (y \oplus z) = (x \oplus y) \oplus z & x \oplus y = y \oplus x \end{array} \right.$$

It defines a congruence relation that is closed under substitutions of terms for variables.

We rely on a sort system for terms. Terms which respect this sort-system are said to be *well-typed*. It includes a set of base type Base and a set of ciphertext type Cipher. We have variables and constants of both types. Moreover we assume that our function symbols have the following type:

$$\begin{array}{rcccccc} \oplus & : & \mathsf{Base} & \times & \mathsf{Base} & \rightarrow & \mathsf{Base} \\ \{\_\}\_ & : & \mathsf{Base} & \times & \mathsf{Base} & \rightarrow & \mathsf{Cipher} \\ \mathsf{dec} & : & \mathsf{Cipher} & \times & \mathsf{Base} & \rightarrow & \mathsf{Base} \end{array}$$

A pure term t is a well-typed term whose only encryption symbol (when such a symbol exists) is at its root position. We say that a term t is headed with f if its root symbol is f. The set of variables occurring in t is denoted $vars(t)$. We denote by $st(t)$ the set of subterms of t. This notation is extended as expected to set of terms. A term is *ground* if it has no variable.

In the IBM 4758 CCA API, as in many others, symmetric keys are subject to parity checking. These checks are represented in our formalism by occurrences of the predicate symbols $\mathsf{chkEven}$ and chkOdd, each having a term as argument. Intuitively, $\mathsf{chkOdd}(t)$ means that t has an odd parity. Among the constants in Σ, some have a parity. By default (no explicit parity given to a constant), we will assume that such a constant has no parity. Moreover, we have some rules to infer parity from known facts, which are:

$$\begin{array}{rcl} \mathsf{chkEven}(x_1),\ \mathsf{chkEven}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkOdd}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkEven}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkOdd}(x_1 \oplus x_2) \end{array}$$

Intruder capabilities and the protocol behaviour are described using *rules* as defined below.

Definition 1 (API rule) *An API rule is a rule of the form* $\mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k), x_1, \ldots, x_n \rightarrow t$, *where*

- $x_1, \ldots, x_n$ *are variables,*

- t *is a term such that* $vars(t) \subseteq \{x_1, \ldots, x_n\}$,
- $u_1, \ldots, u_k$ *are terms of* Base *type not headed with* $\oplus$,
- $\mathsf{chk}_i \in \{\mathsf{chkOdd}, \mathsf{chkEven}\}$, $1 \leq i \leq k$.

We also assume that the rule only involves pure terms.

We now describe *key conjuring*: suppose an intruder wants to use a particular command in an attack, but does not have access to an appropriate key. For example, suppose he has no data keys (terms of the form $\{\mathsf{d1}\}_{\mathsf{km}\oplus\mathsf{data}}$), but wants to use the *Encrypt Data* command. In an implicit decryption formalism, the command is defined like this

$$\mathsf{x}, \{\mathsf{xkey}\}_{\mathsf{km}\oplus\mathsf{data}} \rightarrow \{\mathsf{x}\}_{\mathsf{xkey}}$$

If the intruder has no data keys, he can just guess values for the encrypted data key. This is called *key conjuring*. In [4], we introduce a transformation that detects all possible key conjuring rules. Key conjuring rules are of the following form:

$$\begin{array}{rcl} x_1, \ldots, x_k & \stackrel{\mathsf{new\ n}}{\rightarrow} & t, \mathsf{n} \\ \mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k) & & \mathsf{chk}'_1(v_1), [\mathsf{chk}'_2(\mathsf{n})] \end{array}$$

where our n comes from a set of *nonces*, a subset of the set of constants that does not contain the special constant 0. We assume an infinite number of nonces of both types. A nonce represents a fresh value that has been never used before. As an example, the key conjuring rule derived from the *Encrypt Data* rule above looks like

$$\mathsf{x} \stackrel{\mathsf{new\ n}}{\rightarrow} \{\mathsf{x}\}_{\mathsf{dec(n,km}\oplus\mathsf{data)}}, \mathsf{n}, \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}, \mathsf{km} \oplus \mathsf{data}))$$

The parity check on the right indicates that the intruder learns that decrypting n under key $\mathsf{km} \oplus \mathsf{data}$ gives a valid data key (i.e., a term of odd parity).

3 Parallel Key Search

A parallel key search attack proceeds in three phases: first, the intruder generates a set of encrypted keys of a particular type. Then, he obtains a set of *test patterns*, e.g. a known value encrypted under each key. Then, he generates test patterns of his own off-line until he gets a match, revealing the value of an encrypted key.

We observe that every rule that introduces a fresh nonce can be used an arbitrary number of times to produce a stream of fresh, random values. We therefore model key generation rules in a similar way to key conjuring rules, i.e. they are rules of the form:

$$\begin{array}{rcl} x_1, \ldots, x_n & \stackrel{\mathsf{new\ n}}{\rightarrow} & t[x_1, \ldots, x_n, \mathsf{n}] \\ \mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_k(u_k) & & \mathsf{chk}'_1(v_1), [\mathsf{chk}'_2(\mathsf{n})] \end{array}$$

The $\mathsf{chk}_i(u_i)$s on the left indicate the set of parity predicates that must hold for the rule to apply. The generated key $t[x_1, \ldots, x_n, \mathsf{n}]$ contains some new fresh material n. The $\mathsf{chk}_i(v_i)$s indicate new parity facts which the intruder learns by applying the rule.

We have to account for the intruder generating random data for use in offline operations. This is covered by the rule

$$\begin{array}{rl} \stackrel{\mathsf{new\ n}}{\rightarrow} & \mathsf{n} \\ & [\mathsf{chkEven}(n); \mathsf{chkOdd}(n)] \end{array}$$

Note that the intruder can generate fresh terms of either odd or even parity. We now define a *fingerprint function*. A fingerprint function is a bijection over bitstrings. In the context of our definition of APIs, we can define fingerprint functions recursively as follows:

Definition 2 (Fingerprint function)

1. $f : x \rightarrow x$ *is a fingerprint function.*
2. *If* c *is a constant and* $g()$ *a fingerprint function, then* $f : x \rightarrow g(x) \oplus c$ *is a fingerprint function.*
3. *If* c *is a constant and* $g()$ *a fingerprint function, then* $f : x \rightarrow \{g(x)\}_c$ *is a fingerprint function.*

4. *If c is a constant and $g()$ a fingerprint function, then $f : x \rightarrow \{c\}_{g(x)}$ is a fingerprint function.*
5. *If c is a constant and $g()$ a fingerprint function, then $f : x \rightarrow \mathsf{dec}(c, g(x))$ is a fingerprint function.*
6. *If c is a constant and $g()$ a fingerprint function, then $f : x \rightarrow \mathsf{dec}(g(x), c)$ is a fingerprint function.*

Parallel key search is modelled by a *collision rule*: for all fingerprint functions f, g:

$$\mathsf{n}_1, \mathsf{f}(\mathsf{n}_1), \mathsf{f}(\mathsf{g}(\mathsf{n}_2)) \quad \rightarrow \quad \mathsf{g}(\mathsf{n}_2)$$

The intuitive meaning of this rule is: 'if the intruder can generate a stream of values n_1 with associated fingerprints $\mathsf{f}(\mathsf{n}_1)$, and he can also generate a stream $\mathsf{f}(\mathsf{g}(\mathsf{n}_2))$, then by parallel key search he can generate a stream $\mathsf{g}(\mathsf{n}_2)$.' Note that the rule as written does not attempt to account for the computational effort required to produce the stream of $\mathsf{g}(\mathsf{n}_2)$s. This will come later (see section 3.2). Note further that this rule is higher order: it quantifies over functions. An implementation would have to either support second order unification, or have some pre-chosen list of commonly used fingerprint functions (such as encrypting a string of 0s, the most commonly used 'key test' operation).

3.1 An Example Attack

We now give an example of a parallel key search attack in our formalism. This attack was implemented using FPGA hardware by Clayton and Bond and used to extract the keys from a working IBM 4758, [3]. Here, we give steps 1–6 of the attack, first in italics, literatim as described by Clayton[1], then as written in our model:

Step 1: Create a single DES key part of unknown value.
Step 2: Create a single DES key part of value 0.
Step 3: XOR these two key parts together to create a DES data key
Step 4: Encrypt zero [..] with the data key and display the encrypted value.
Step 5: Repeat steps 2 to 4 for values 1..16383

These five steps are accomplished using the key conjuring version of Key Part Import 3 (see [4]), using a random nonce stream n_1 of even parity values as the key part, and then using the encrypt data command to obtain the check values:

$$\xrightarrow{\text{new } \mathsf{n}_1} \begin{array}{l} \mathsf{n}_1 \\ \mathsf{chkEven}(\mathsf{n}_1) \end{array}$$

Key Part Import 3:

$$\begin{array}{rcl} \mathsf{n}_1,\ \mathsf{data} & \xrightarrow{\text{new } \mathsf{n}_2} & \mathsf{n}_2, \{\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_1\}_{\mathsf{km} \oplus \mathsf{data}} \\ \mathsf{chkEven}(\mathsf{data}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype})) \\ \mathsf{chkEven}(\mathsf{n}_1) & & \end{array}$$

Encrypt Data:

$$\begin{array}{rcl} 0, \{\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_1\}_{\mathsf{km} \oplus \mathsf{data}} & \rightarrow & \{0\}_{\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_1} \\ \mathsf{chkOdd}(\mathsf{dec}(\{\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_1\}_{\mathsf{km} \oplus \mathsf{data}}, \mathsf{km} \oplus \mathsf{data})) & & \end{array}$$

This set of tasks less than 10 minutes to run, and at the end of them we have 16384 different encrypted versions of zero. We can then use our FPGA design to crack these values in parallel. Hence we have determined the "unknown" value from Step 1.

This is modelled by the following rules:

$$\begin{array}{rcl} & \xrightarrow{\text{new } \mathsf{n}_3} & \begin{array}{l} \mathsf{n}_3 \\ \mathsf{chkOdd}(\mathsf{n}_3) \end{array} \\ \mathsf{n}_3, \mathsf{n}_1 & \rightarrow & \mathsf{n}_3 \oplus \mathsf{n}_1 \\ 0, \mathsf{n}_3 & \rightarrow & \{0\}_{\mathsf{n}_3} \end{array}$$

Collision rule:

$$\mathsf{n}_3, \{0\}_{\mathsf{n}_3 \oplus \mathsf{n}_1}, \{0\}_{\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_1} \quad \rightarrow \quad \mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data})$$

[1] See `http://www.cl.cam.ac.uk/~rnc1/descrack`

Step 6: Combine the unknown value from Step 1 with a known value so as to create a data key. We now know the value of a data key within the IBM 4758.

$$\begin{array}{rcl} & \overset{\mathsf{new}\ \mathsf{n}_4}{\longrightarrow} & \mathsf{n}_4 \\ & & \mathsf{chkEven}(\mathsf{n}_4) \\ \textit{Key Part Import 3}: & & \\ \mathsf{chkEven}(\mathsf{n}_4),\ \mathsf{n}_2,\ \mathsf{n}_4,\ \mathsf{data} & \rightarrow & \{\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_4\}_{\mathsf{km} \oplus \mathsf{xtype}} \\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype})) & & \\ \mathsf{chkEven}(\mathsf{data}) & & \\ \mathsf{n}_4, \mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{xtype}) & \rightarrow & \mathsf{dec}(\mathsf{n}_2, \mathsf{km} \oplus \mathsf{kp} \oplus \mathsf{data}) \oplus \mathsf{n}_4 \end{array}$$

3.2 Quantitative Analysis

If considered as a regular Dolev-Yao rule, the collision rule is far too powerful. For our model to be realistic, we must introduce a quantitative element to the model, taking account of the computational cost of particular operations. Previous efforts to introduce quantitative modelling to the Dolev-Yao model have assigned concrete costs, [2], or probabilities, [1], to each rule. This is insufficient for modelling attacks involving parallel key search, since the cost of using the rule will vary depending on the cost of obtaining the values n_1, $f(n_1)$, and $f(n_2)$. Instead, we require a calculus that assigns a cost to each term in the intruder's knowledge. Each API or intruder rule, with k terms on the left hand side, has associated with it a cost function $C : \mathbb{N}^k \rightarrow \mathbb{N}$, representing the cost of obtaining the term on the right hand side of a rule as a function of the costs of the terms on the left. The cost functions should be paramaterised in terms of offline work and access to the API using a similar treatment to the 'concrete security' analysis of symmetric key encryption schemes, where a symmetric encryption oracle is considered. Developing our quantitative analysis is a priority in our ongoing work on this topic.

References

[1] P. Adão, P. Mateus, T. Reis, and L. Viganò. Towards a quantitative analysis of security protocols. *Electr. Notes Theor. Comput. Sci.*, 164(3):3–25, 2006.

[2] I. Cervesato. Fine-grained msr specifications for quantitative security analysis, 2004.

[3] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, pages 579–592, 2002.

[4] V. Cortier, S. Delaune, and G. Steel. A formal theory of key conjuring. To appear at the 20th IEEE Computer Security Foundations Symposium (CSF20).

[5] V. Cortier, S. Delaune, and G. Steel. A formal theory of key conjuring. Technical Report 6234, Unitde recherche INRIA Lorraine, February 2007.

[6] 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.

[7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[8] 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.

[9] 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.