

# Secure Middleware for Infrastructure Systems <sup>1</sup>

Technical Report

R. Blahut, T. Clancy, X. Hua, J. Kim, N. Kiyavash, M. Ma,  
K. Markandan, S. Mathur, M. Nigam, D. Pozdol, S. Song,  
S. Sriram, A. Suk, B. Wang, T. Wong, A. Vovchak

Illinois Center for Cryptography and Information Protection  
Coordinated Science Laboratory  
University of Illinois, Urbana-Champaign

January 2004

<sup>1</sup>Supported by The Boeing Company under Illinois Technology Challenge Grant 03-115. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of The Boeing Company or the State of Illinois. Portions of this document have been independently published and are ©2003 Association for Computing Machinery.



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Cryptographic Layer</b>	<b>9</b>
2.1	Elliptic Curve Cryptography . . . . .	9
2.1.1	Finite Field . . . . .	10
2.1.2	Elliptic Curves . . . . .	11
2.1.3	Cryptographic Protocols . . . . .	12
2.1.4	Implementation . . . . .	14
2.1.5	Experiments . . . . .	16
2.2	Fingerprint Vault . . . . .	16
2.2.1	Background . . . . .	17
2.2.2	Past Work . . . . .	19
2.2.3	Fingerprint Vault . . . . .	21
2.2.4	Unlocking Complexity . . . . .	28
2.2.5	Empirical Results . . . . .	32
2.2.6	Conclusion . . . . .	34
<b>3</b>	<b>SmartCard</b>	<b>35</b>
3.1	Structure . . . . .	35
3.2	Data Structures . . . . .	36
3.3	API Methods . . . . .	36
3.4	Comments on ECC . . . . .	37
<b>4</b>	<b>SESAME</b>	<b>39</b>
4.1	Protocol . . . . .	40
4.2	Relation to Kerberos . . . . .	40
4.3	GSS-API . . . . .	40
4.4	Formal Analysis . . . . .	41
4.4.1	Preliminaries . . . . .	41

4.4.2	SESAME . . . . .	42
4.4.3	Protocol Analysis . . . . .	42
4.4.4	Kerberos . . . . .	45
4.4.5	Kerberos versus SESAME . . . . .	46
4.5	Implementation . . . . .	46
<b>5</b>	<b>Distributed Computing using Jini</b>	<b>47</b>
<b>6</b>	<b>Database Security</b>	<b>49</b>
6.1	Motivation . . . . .	49
6.2	Java Wrapper . . . . .	49
6.3	Implementation . . . . .	50
6.4	Future Work . . . . .	52
<b>7</b>	<b>Role-Based Access Control</b>	<b>53</b>
7.1	Overview of RBAC . . . . .	54
7.2	Role Hierarchy . . . . .	54
7.3	Role Authorization . . . . .	55
7.4	Role Activation . . . . .	56
7.5	System Design . . . . .	56
7.6	Centralized versus Distributed RBAC . . . . .	58
7.7	Secure Communication . . . . .	59

# Chapter 1

## Introduction

The goal of this project is to design airport ticketing, check-in, and security around passengers using smartcards and biometric authentication to establish their identity. We provide a provably secure framework using advanced cryptographic algorithms and protocols. Given this framework, application developers can easily design and implement distributed applications without concern for protecting the underlying data.

This framework was implemented as cryptographic extensions to both the Java Jini distributed system architecture and Java JDBC database transaction protocols. Passengers, airport employees, and other affiliates authenticate themselves to the system using smartcards and fingerprints. Once authenticated, session keys are automatically established between the user and all other entities in the network to which the user wishes to communicate. This is accomplished using the SESAME protocol. These protocols both protect and authenticate data being transmitted on the network.

The goal was not to implement an entire airport, but rather to provide the tools for application developers to design an airport. While an airport is a motivating example for this technology, it could just as easily be used in health care, hotels, or any other similar environment. However, in order to demonstrate the technology, we provide a rudimentary airport implementation that supports the basic and most obvious passenger and employee tasks.

The technologies used in this project include:

- Smartcards: we used JavaCard smartcards manufactured by Schlumberger, which allowed flexibility in programming and design.
- Fingerprint Vault: based on the fuzzy vault [11], the fingerprint vault provides secure biometric authentication in a smartcard environment.

- Elliptic Curve Cryptography: in place of RSA, Elliptic Curve Cryptography provides the foundation of authentication using public-key digital signatures.
- SESAME: to secure the transactions between network devices, the SESAME cryptographic protocol was used.
- Java Jini: to facilitate distributed application development, the Java Jini architecture was cryptographically enhanced.
- MySQL: the majority of enterprise applications require database connectivity, thus we provide cryptographically enhanced extensions to the JDBC protocol and present its functionality with the MySQL database engine.
- Role Based Access Control: in an effort to standardize and modularize the access control mechanisms, a formal RBAC model was used.

All these technologies have been integrated to provide the security. As shown in Figure 1.1, the security framework consists of several layers. SESAME links the underlying ECC authentication to the higher-layer Jini and JDBC network protocols.

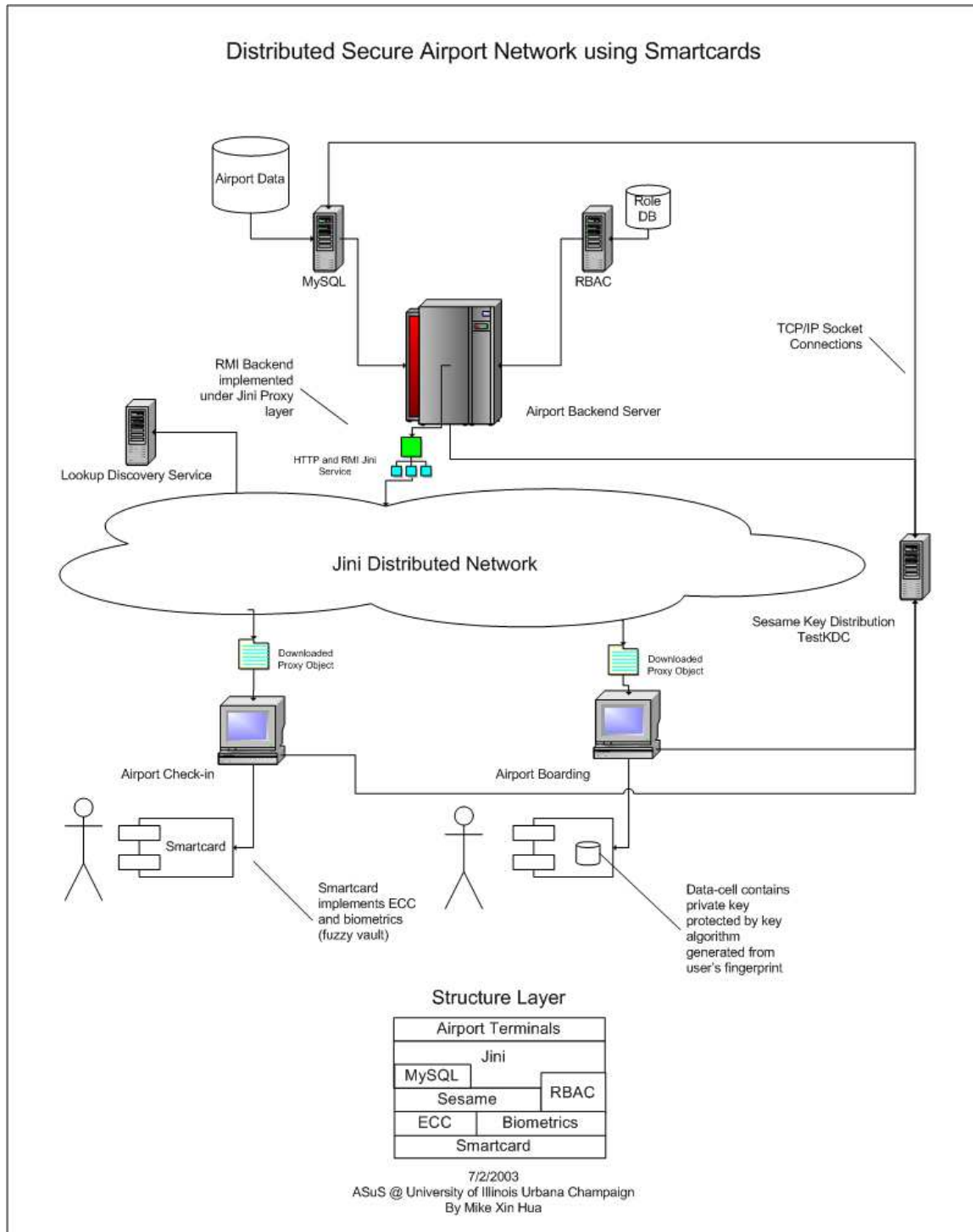


Figure 1.1: Airport security system framework



## Chapter 2

# Cryptographic Layer

There are several different cryptosystems used to protect the underlying data in the distributed architecture. The first is Elliptic Curve Cryptography, which is a public-key cryptosystem primarily used for authentication. The second is the Fingerprint Vault, used to protect your Elliptic Curve private key on the smartcard. Other symmetric-key algorithms are used, but are less significant to the overall design. The following sections describe ECC and the Fingerprint Vault in more detail.

### 2.1 Elliptic Curve Cryptography

When authenticating via public-key cryptography, to ensure security in the widely used RSA cryptosystem, key sizes must be a minimum of 1024 bits. As computing power increases, larger key sizes will be needed to guarantee reasonable levels of security. It is clear that the system becomes less efficient as the length of the key increases. Elliptic Curve Cryptography (ECC) provides an alternative to RSA. The security of ECC is based on the Elliptic Curve Discrete Log Problem (ECDLP), which is difficult to break for a large class of cryptographically-secure curves, as compared to RSA where factorization algorithms can lead to sub-exponential time attacks. This means that significantly smaller parameters can be used in ECC than in RSA. This helps in having smaller key size (150–250 bits), hence faster computations.

### 2.1.1 Finite Field

A field of a finite number of elements is denoted  $\mathbf{F}_q$  or  $\text{GF}(q)$ , where  $q$  is the number of elements. This is also known as a Galois Field. The order of a Finite field  $\text{GF}(q)$  is the number of elements in  $\text{GF}(q)$ . Further, there exists a finite field  $\text{GF}(q)$  of order  $q$  if and only if  $q$  is a prime power, that is either  $q$  is prime or  $q = p^m$ , where  $p$  is prime and  $m$  is an integer. In the latter case,  $p$  is called the characteristic of  $\text{GF}(q)$  and  $m$  is called the extension degree of  $\text{GF}(q)$ .

Now let us look at the details of two classes of finite fields, the prime field  $\text{GF}(p)$  and the binary field  $\text{GF}(2^m)$ .

The prime field  $\text{GF}(p)$  consists of the set of integers  $\{0, 1, 2, \dots, p-1\}$ , with the following arithmetic operations defined over it:

- Addition:  $a, b, r \in \text{GF}(p)$ , where  $r = (a + b) \pmod{p}$
- Multiplication:  $a, b, s \in \text{GF}(p)$ , where  $s = ab \pmod{p}$
- Inversion:  $a, b \in \text{GF}(p)$ , where  $b = a^{-1} \pmod{p}$  is computed using the Extended Euclidean Algorithm

The binary field  $\text{GF}(2^m)$  is less intuitive. Elements are represented as binary coefficients of  $m$  orthogonal bases. Commonly used is a *polynomial basis* where elements are represented as polynomials of degree at most  $m-1$  with binary coefficients. Multiplication is then computed modulo a degree  $m$  polynomial  $f(x)$ . A field element  $a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \dots + a_1x + a_0$  is associated with the binary vector  $a = (a_{m-1}a_{m-2} \dots a_2a_1a_0)$ .

Arithmetic is computed as follows:

- Addition: If  $a = (a_{m-1} \dots a_0)$  and  $b = (b_{m-1} \dots b_0)$  are elements of  $\text{GF}(2^m)$ , then  $c = a + b = ([a_{m-1} + b_{m-1} \pmod{2}] \dots [a_0 + b_0 \pmod{2}])$ .
- Multiplication: If  $a = (a_{m-1} \dots a_0)$  and  $b = (b_{m-1} \dots b_0)$  are elements of  $\text{GF}(2^m)$ , then  $c = a * b \pmod{f(x)}$ , such that  $d = a * b$  is computed using standard polynomial arithmetic and then  $c$  is the remainder of  $d/f(x)$ .
- Squaring: Since in a characteristic  $k$  finite field,  $(a + b)^k = a^k + b^k$ , squaring is easily implemented by inserting 0's between every two consecutive bits, and then computing the remainder of the result when divided by  $f(x)$ .

- Reduction: Rather than actually dividing by  $f(x)$  (a costly operation) to compute  $(\text{mod } f(x))$ , an alternative is to use  $f(x) = x^m + g(x) = 0$ , or,  $x^m = g(x)$ . Thus if  $x^{m+i} = 1$ , simply add to the result  $f(x) * x^i$ , which cancels that bit.
- Inversion: If  $a$  is a nonzero element in  $\text{GF}(2^m)$ , then the inverse of  $a$ , denoted  $a^{-1}$ , is a unique element  $c \in \text{GF}(2^m)$ , where  $a * c = c * a = 1(\text{mod } f(x))$ . Inversion uses a modified version of the Extended Euclidean Algorithm for polynomials [8] and is the most expensive operation (15 to 20 times slower than multiplication).

### 2.1.2 Elliptic Curves

Elliptic curves [19] are cubic curves of the form

$$C : y^3 = x^3 + ax + b.$$

Elliptic curves over  $\mathcal{R}^2 = \mathcal{R} \times \mathcal{R}$  is defined by the set of points  $(x, y)$  which satisfy the equation  $y^3 = x^3 + ax + b$ , along with a point  $\mathcal{O}$ , which is the point at infinity and the group identity.

An elliptic curve over a finite field  $\text{GF}(p)$  is defined by the parameters  $a, b \in \text{GF}(p)$  such that  $a, b$  satisfy  $4a^3 + 27b^2 \neq 0(\text{mod } p)$ , consisting of the set of points  $(x, y) \in \text{GF}(p) \times \text{GF}(p)$ , satisfying the equation  $y^2 = x^3 + ax + b$ . The set of points on the elliptic curve also include point  $\mathcal{O}$ , which is the point at infinity and which is the identity element under addition.

The addition operation is specified as follows.

- $P + \mathcal{O} = \mathcal{O} + P = P, \forall P$
- If  $P = (x, y)$  is on the elliptic curve, then  $(x, y) + (x, -y) = \mathcal{O}$ . The point  $(x, -y)$  is also on the curve, called the inverse of  $P$ , and denoted  $-P$ .
- If  $P = (x_1, y_1), Q = (x_2, y_2)$ , and  $P \neq Q$ , then  $R = P + Q = (x_3, y_3)$  is on the elliptic curve, where if  $\lambda = (y_2 - y_1)/(x_2 - x_1)$ ,  $x_3 = \lambda^2 - x_1 - x_2$  and  $y_3 = \lambda(x_1 - x_3) - y_1$ . Geometrically, the sum of 2 points can be visualized as the intersection point between the curve and the straight line passing through both the points.
- If  $P = Q$  then  $R = P + Q = 2 * P = (x_3, y_3)$  is on the elliptic curve, where if  $\lambda = (3x_1^2 + a)/2y_1$ , then  $x_3 = \lambda^2 - 2 * x_1$  and  $y_3 = \lambda(x_1 - x_3) - y_1$ . This operation is called doubling and can be visualized

as the intersection point between the elliptic curve and the tangent at  $P$ .

An elliptic curve over a finite field  $\text{GF}(2^m)$  is defined by the parameters  $a, b \in \text{GF}(2^m)$  satisfying the same conditions as before. The addition and doubling operations are specified as follows:

- If  $P = (x_1, y_1)$ ,  $Q = (x_2, y_2)$ , and  $P \neq Q$ , then  $R = P + Q = (x_3, y_3)$  is on the elliptic curve, where if  $\lambda = (y_2 - y_1)/(x_2 - x_1)$ ,  $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$  and  $y_3 = \lambda(x_1 + x_3) + y_1 + x_3$ .
- If  $P = Q$  then  $R = P + Q = 2 * P = (x_3, y_3)$  is on the elliptic curve, where if  $\lambda = x_1 + x_1/y_1$ , then  $x_3 = \lambda^2 + \lambda + a$  and  $y_3 = \lambda(x_1 + x_3) + y_1 + x_3$ .

The scalar multiplication, or multiplying a point to an integer, is defined as repeatedly adding a point to itself. That is,

$$k \cdot P = \sum_{i=1}^k P.$$

Many optimizations exist for computing this sum, the most common being binary expansion and non-adjacent form (NAF) [22].

In situations where inversion is very expensive relative to multiplication, it is advantageous to use projective coordinates. There are two types of projective coordinates.

Standard Projective Coordinates, represented  $(x, y, z)$ , correspond to the affine point  $(x/z, y/z)$ . The projective equation of the curve is

$$y^2z + xyz = x^3 + ax^2z + bz^3$$

Jacobian Projective Coordinates, also represented  $(x, y, z)$ , correspond to the affine point  $(x/z^2, y/z^3)$ . The projective equation of the curve is

$$y^2 + xyz = x^3 + ax^2z^2 + bz^6$$

When we use projective coordinates, the inversion is substituted by multiplication operations.

### 2.1.3 Cryptographic Protocols

Now that the fundamentals have been explained, the elliptic curve group can be used to implement various cryptographic operations, the most important of which are digital signatures and encryption.

In the following sections, let  $G$  be a generator of a particular elliptic curve group of order  $n$  over  $\text{GF}(q)$ .

**Key generation**

A public/private key pair can be generated as follows:

1. Select a random integer  $d \in \{1, \dots, n - 1\}$ .
2. Compute  $Q = dG$ .
3. Alice's public key is  $Q$  and private key is  $d$ .

It should be noted that the public key generated needs to be validated to ensure that it satisfies the arithmetic requirement of elliptic curve public key. A public key  $Q = (x_q, y_q)$  is validated using the following procedure:

1. Check that  $Q \neq \mathcal{O}$ .
2. Check that  $x_q$  and  $y_q$  are properly represented elements of  $\text{GF}(q)$ .
3. Check that  $Q$  lies on the elliptic curve.
4. Check that  $nQ = \mathcal{O}$ .

**Encryption and Decryption**

A message  $M$  can be encrypted with public key  $Q$  as follows:

1. Selects a random integer  $k \in \{1, \dots, n - 1\}$ .
2. Compute  $R = kG$ .
3. Compute  $S = M + kQ$ .

The pair  $(R, S)$  is the ciphertext for plaintext message  $m$ .

To decrypt message  $(R, S)$  using private key  $d$ :

1. Compute  $T = dR = (dk)G = kQ$ .
2. Compute  $U = S - T = M + kQ - kQ = M$ .

Thus,  $U$  is the unique decryption of message  $M$ .

EC	ECDSA	Key
ECPoint (imports BigInteger)		
EllipticCurve (imports BigInteger)		

Figure 2.1: Structure of ECC implementation

### Elliptic Curve Digital Signature Algorithm (ECDSA)

To sign a message  $M$  using private key  $d$ :

1. Selects a random integer  $k \in \{1, \dots, n - 1\}$ .
2. Compute  $R = kG = (r_x, r_y)$ .
3. Compute  $t = k^{-1}(\text{mod } n)$ .
4. Compute  $e = \text{SHA1}(M)$ , where SHA1 is the Secure Hash Algorithm.
5. Compute  $s = k^{-1}(e + dr)(\text{mod } n)$ .

The signature of message  $M$  is the pair  $(r, s)$ . Note that if either  $r = 0$  or  $s = 0$ , the process should be repeated with a new random  $k$ .

To verify the signature using public key  $Q$ :

1. Compute  $e = \text{SHA1}(M)$ .
2. Compute  $w = s^{-1}(\text{mod } n)$ .
3. Compute  $u_1 = ew(\text{mod } n)$  and  $u_2 = rw(\text{mod } n)$ .
4. Compute  $X = (x_1, y_1) = u_1G + u_2Q$ .
5. Compute  $v = x_1(\text{mod } n)$

The signature should be accepted if  $v = r$ .

#### 2.1.4 Implementation

To implement an ECC system, the followings have to be specified: domain parameters including the underlying finite field, field representation, and elliptic curve; algorithms for field arithmetic; elliptic curve arithmetic; and the cryptographic protocols. The resulting implementation includes 5 classes, organized as in Figure 2.1.

The `EllipticCurve` class defines all domain parameters. In our implementation, the prime field  $\text{GF}(p)$  has been selected as the finite field. Parameters for elliptic curves have been defined as those in NIST standard [24, 32] with a key length of 192 bits. The `ECPoint` class represents the point on the elliptic curve defined in `EllipticCurve` class. `ECPoint` defines the required operations such as addition, negation, and scalar multiplication. The `Key` class defines a method to generate a random key pair. The `EC` class defines encryption and decryption methods. The `ECDSA` class defines methods for signature and verification, and conforms to the NIST standard.

ECC has been implemented in software many times before and the most frequently it has been done in languages like C/C++. The advantage of using C/C++, as opposed to Java, is the speed. Thus, to implement a fast Java ECC is a challenging task. The key decision to make it as efficient as possible was to utilize `BigInteger` class that was provided by Java API. This class is an important component of the API, so it is highly optimized specifically for Java.

In our original implementation, we used an array of integers and used shifting operations to represent and mutate finite field elements. However, due to the efficient implementation of `BigInteger`, we decided to convert the finite field elements back and forth to `BigIntegers`, so that we could use inversion (`modInverse`) operation provided by `BigInteger`. Conversion from our original implementation to `BigIntegers` seemed justifiable since `BigInteger` stores its value in the exactly same array of integers. Since we used not only `modInverse` operation, but other operations such as addition and multiplication, which are provided by `BigInteger` class, we could get very optimized results, which will be shown later.

The following methods are externally available for application developers:

1. Key class
 

```
public static void make(String user) – Generates a key for the user
```
2. EC class
 

```
public static byte[] encrypt(byte[] m, String usr) – encrypt a message with using the user’s key
public static byte[] decrypt(byte[] str, String usr) – decrypt a message with using the user’s key
```
3. ECDSA class
 

```
public static Signature get_signature(byte[] msg, String usr)
```

Operation	Old implementation	New implementation
Key initialization	265ms	93ms
Encryption	406ms	63ms
Decryption	203ms	31ms
Signature	203ms	47ms
Verification	410ms	63ms

– sign a message with the user’s key

```
public static boolean verify_signature(byte[] msg, Signature
sg, String usr) – verify the signature with the user’s key
```

### 2.1.5 Experiments

We performed brief experiments on the time taken by each operation defined in the previous section. Experiments were performed on a 2.4GHz Pentium 4, running Windows XP. The result is shown in Table ??..

We can see that the new implementation utilizing the `BigInteger` class is 3–7 times faster than the old implementation, which used our own finite field implementation.

In the original project design, ECC was supposed to be implemented as a JavaCard applet on the smartcard [27]. Considering the limited computing power of smartcards, this decision was obvious over other public-key cryptographic algorithms. However, in the end the smartcard was not fast enough for the software-based cryptographic operations. Therefore, the possible future work is to port the current ECC algorithms into the smartcard. Since current version utilizes `BigInteger` class very much, converting `BigInteger` class will be the first task. Considering current JavaCard API only accommodates short integers and bytes, removing integers and long integers from the `BigInteger` class will be the key part of the conversion.

## 2.2 Fingerprint Vault <sup>1</sup>

In this section, the fundamental insecurities hampering a scalable, widespread deployment of biometric authentication are examined, and a cryptosystem capable of using fingerprint data as its key is presented. For our

<sup>1</sup>This section was independently published and presented at the ACM Workshop on Biometrics Methods and Applications, ©2003, Association for Computing Machinery, 1-58113-779-6/03/00011.

application, we focus on situations where a private key stored on a smart-card is used for authentication in a networked environment, and we assume an attacker can launch off-line attacks against a stolen card.

Juels and Sudan's *fuzzy vault* is used as a starting point for building and analyzing a secure authentication scheme using fingerprints and smartcards called a *fingerprint vault*.

The parameters of the vault are selected such that the attacker's vault unlocking complexity is maximized, subject to zero unlocking complexity with a matching fingerprint and a reasonable amount of error. For a feature location measurement variance of 9 pixels, the optimal vault is  $2^{69}$  times more difficult to unlock for an attacker compared to a user possessing a matching fingerprint, along with approximately a 30% chance of unlocking failure.

The overall goals of this work are as follows:

- define the fingerprint vault scheme and its associated algorithms;
- present probabilistic and systematic bounds on some of the vault parameters;
- define a class of unlocking techniques that can be used for both attacks and by legitimate users; and
- find the optimal fuzzy vault parameter choice to maximize attackers' complexity and minimize users' complexity, while ensuring reasonable reproducibility

### 2.2.1 Background

First, we present a simplified, generic protocol for use with smartcard-based biometric authentication. It is similar to those found in many popular public-key authentication schemes, such as SESAME [30, 18] and SSH [33].

In general, a server should believe a user is who they say they are if they can provide a signed message containing their identity and a nonce or challenge (i.e. random number) selected by the server. If the public key associated with the specified identity correctly verifies the signature, only the valid user could have sent it. If a certificate is not provided by the user, the server will need a database of public keys.

From the client standpoint, signing ability is required to authenticate. Figure 2.2 demonstrates how this can be done with biometric data. First, a fingerprint image is captured from a scanner. This data is sent to a terminal

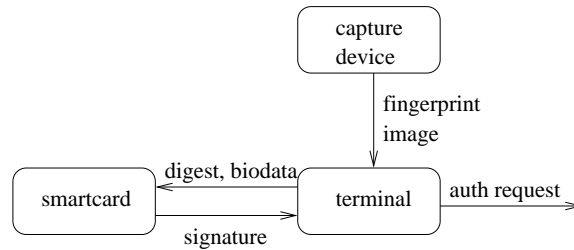


Figure 2.2: Authentication Diagram

which translates it into some smaller numeric representation, or template. Both the message digest we wish to sign and the biometric template are sent to the smartcard. Provided the biometric data is valid, the smartcard will use its internally stored private key to generate and return a signature for the message digest.

Our analysis shall focus on the methods by which smartcards use biometric data to sign a message. This assumes the worst case scenario: a smartcard has been stolen, and an attacker with complete physical access is attempting to retrieve the private key.

Given physical access, there are two main classes of physical attacks against smartcards: non-invasive or side-channel attacks, and invasive attacks.

Three of the most popular non-invasive attacks are power analysis [14], timing analysis [13], and electromagnetic (EM) analysis [1]. All of these attacks can be used to determine sensitive information on a smartcard; however, non-invasive attacks can generally be thwarted by clever algorithms, data obfuscation, and shielding techniques.

Invasive attacks [16, 17] generally involve dissolving the chip packaging and reverse engineering the processor itself. Given complete physical access, it is impossible to prevent an attacker from retrieving data stored in memory. The only way to protect against such an attack is to encrypt the contents of memory using a key not stored on the card itself. Then, an attacker may retrieve the data from the card, but it will be of no use.

Protecting the smartcard now requires efforts on two fronts. First, the smartcard processor itself must be designed such that it is immune to the various on-line side-channel attacks. Secondly, without a matching fingerprint, an attacker should not be able to obtain any sensitive information from the card. In particular, users' private keys must be stored in an encrypted

format. This second front is the focus of the work presented here.

### 2.2.2 Past Work

In the past few years, there have been several research efforts aimed at addressing the intersections between cryptography and biometrics. Here we address biometric cryptosystems in general, delve more deeply in to the fuzzy vault, and then briefly examine various polynomial reconstruction techniques.

#### Biometric Cryptosystems

In 1998, Davida, et al. [6], were among the first to suggest off-line biometric authentication. It moved biometric data from a central server into a signed form on a portable storage device, such as a smartcard. Their system was essentially a PKI-like environment that did local fingerprint matching. Its main flaw is that it required some local authentication authority to have a key capable of decrypting the template stored on the storage device. While they address the key management issues, the basic premise is still that of local fingerprint matching, and is therefore inherently insecure.

The next year there were three innovative, yet similar methods that did not perform biometric matching. The first is the fuzzy commitment scheme [12]. Here, a secret (presumably a private key used for later authentication) is encoded using a standard error correcting code such as Hamming or Reed-Solomon, and then XOR-ed it with a biometric template. To retrieve the secret, a slightly different biometric template can again be XOR-ed, and the result put through an error correcting decoder. Some small number of bit errors introduced in the key can be corrected through the decoding process. The major flaw of this system is that biometric data is often subject to reordering and erasures, which cannot be handled using this simple scheme.

In [23], a technique was proposed using the phase information of a Fourier transform of the fingerprint image. The fingerprint information and a randomly chosen key are mixed together to make it impossible to recover one without the other. In order to tolerate errors, the system used a filter that minimizes the output variance corresponding to the input images. To provide further redundancy, an encoding process stores each bit multiple times. The work does not address how much these steps reduce the entropy of the original image; thus, it is not clear that there exists a set of parameters which will allow the system to reliably recognize legitimate users while providing a reasonable amount of security.

A third paper [21] has a similar theoretical foundation to this work, but aims toward a completely different application. Here, Monroe, et al., attempt to add entropy to users' passwords on a computer system by incorporating data from the way in which they type their password. Since the biometric being used here is so radically different from fingerprints, their results are not applicable to this work.

Recently, Juels and Sudan [11] proposed the *fuzzy vault*, a new architecture with applications similar to Juels and Wattenberg's *fuzzy commitment scheme*, but is more compatible with partial and reordered data. The fuzzy vault is used here as a starting point for the biometric scheme presented in this paper.

### Fuzzy Vault

Here, we describe the original fuzzy vault, with some slight notational differences. As with any cryptosystem, there is some message  $m$  that needs to be encrypted, or in this case *locked*. Some symmetric fuzzy key can be used to accomplish this task, and then used again later to decrypt, or *unlock* the original message. Here, our message  $m$  is first encoded as the coefficients of some degree  $k$  polynomial in  $x$  over a finite field  $\mathbb{F}_q$ .

This polynomial  $f(x)$  is now the secret to protect. The locking set  $\mathcal{L}$  is a set of  $t$  values  $l_i \in \mathbb{F}_q$  making up the fuzzy encryption key, where  $t > k$ . The locked vault contains all the pairs  $(l_i, f(l_i))$  and some large number of chaff points  $(\alpha_j, \beta_j)$ , where  $f(\alpha_j) \neq \beta_j$ . After adding the chaff points, the total number of items in the vault is  $r$ .

In order to crack this system, an attacker must be able to separate the chaff points from the legitimate points in the vault. The difficulty of this operation is a function of the number of chaff points, among other things. A legitimate user should be able to unlock the vault if they can narrow the search space. In general, to successfully interpolate the polynomial they have an unlocking set  $\mathcal{U}$  of  $t$  elements such that  $\mathcal{L} \cap \mathcal{U}$  contains at least  $k+1$  elements

To summarize the vault parameters:

- $f(x)$  is a degree  $k$  polynomial in  $\mathbb{F}_q[x]$
- $t \geq k$  points in  $\mathcal{L}$  interpolate through  $f(x)$
- $r \geq t$  is total number of points in the vault

This vault shall be referred to as  $\mathcal{V}(\mathbb{F}_q, r, t, k)$ .

Spurious polynomials of degree  $k$  interpolated by  $t$  points may show up in the randomly selected chaff points. In [11], the authors present a

lemma describing the security of their scheme based on the number of these polynomials that exist in a vault with general parameters.

**Lemma 1** *For every  $\mu > 0$ , with probability  $1 - \mu$ , a vault of size  $t$  contains at least  $\frac{\mu}{3}q^{k-t}(r/t)^t$  polynomials  $f'(x)$  of degree less than  $k$  such that the vault contains exactly  $t$  points of the form  $(x, f'(x))$ .*

### Polynomial Interpolation

In order to actually reconstruct the secret locked within the fuzzy vault, the points in the unlocking set must be used to interpolate a polynomial. The unlocking set will contain both real points and chaff points.

The simplest mechanism for recovering the polynomial is a brute-force search, where various  $k + 1$  element subsets of the unlocking set are used to interpolate a degree  $k$  polynomial, using Newtonian Interpolation [9].

A second method is to use a Reed-Solomon decoder [3], as suggested by Juels and Sudan. While RS codes are traditionally used to correct errors in messages transmitted over noisy channels, they are essentially a generalization of the polynomial reconstruction problem. Using a  $(t, k)$  code,  $t$  points can be fed into the decoder, and the degree  $k$  polynomial will be returned.

There are two main RS decoding algorithms: the Berlekamp-Massey algorithm [20], and the Guruswami-Sudan algorithm [7]. Berlekamp-Massey requires  $\frac{k+t}{2}$  real points in the unlocking set while Guruswami-Sudan only requires  $\sqrt{kt}$ . Unfortunately, this extra error correcting capability requires significantly more computation. Interestingly enough, the two algorithms provide nearly identical unlocking complexities for a wide range of vault parameters, so we shall focus on the Berlekamp-Massey method.

Another field called *noisy polynomial interpolation* has had some recent advances, notably by Arora and Khot [2] and Bleichenbacher and Nguyen [5]. However, the results of this work are not applicable to the fuzzy vault. In [2], they examine the problem of finding all polynomials that interpolate through the points  $(x_i, [y_i - \delta, y_i + \delta])$ . The *noise* in our points has been removed by the existence of the fuzzy vault, so this is not useful to us. In [5], they look at the problem of interpolating the points  $(x_i, y_{i,1}), (x_i, y_{i,2}), \dots$ , but since we do not have chaff points overlapping with real points, their new algorithm is also not applicable.

### 2.2.3 Fingerprint Vault

In this section, we describe our modified *fingerprint vault*, and the algorithms used to lock and unlock data. Additionally, theoretic bounds on our ability

to successfully unlock the vault are determined.

### Generalized Fuzzy Vault

First, the fuzzy vault specifies that the size of the locking set, unlocking set, and RS codewords are all the same size. Here, we loosen this restriction.

- $t$  is the number of points in  $\mathcal{L}$
- $\tau$  is the number of points in  $\mathcal{U}$
- $n$  is the RS codeword size

In [11], the authors consider the case where  $\mathcal{L}$  and  $\mathcal{U}$  are taken from discrete sets, however for biometric purposes these values are not discrete (or are taken from a sufficiently high-resolution discrete set). Hence, for all the elements in  $\mathcal{U}$ , we need to find the closest elements in vault, and then try unlocking with those values. Consequently, a quantization problem is introduced. How closely can we pack chaff points and still maintain a reasonable probability of quantization error?

The key being used to lock our fuzzy vault is pixel coordinate locations,  $(x_i, y_i)$ , of features on a fingerprint image. Consequently, an ideal field for the vault is  $\mathbb{F}_{p^2}$ , such that  $p$  is prime.

Lemma 1 probabilistically gives us the number of spurious polynomials of a particular degree in our vault. The presence of many such polynomials is the key to proof of security in [11]. Unfortunately, for reasonable vault parameters (see Section 2.2.5), there exists a  $\delta$  with  $k \leq \delta \leq t$  such that the expected number of spurious degree  $k$  polynomials interpolated by more than  $\delta$  points is less than one. The consequence is that the brute-force search for a degree  $k$  polynomial interpolating  $\delta$  points will yield a unique result, namely, the vault secret.

The precise value of  $\delta$  such that the results of an unlocking attempt can be verified will be required later:

**Corollary 1** *A value of  $\delta$  satisfying the above requirement is*

$$\delta \geq \left\lceil \frac{\log \frac{1}{3} p^{2k}}{\log \frac{kp^2}{r}} \right\rceil \quad (2.1)$$

The requirement on the number of spurious polynomials can be rewritten as

$$\log \left( \frac{1}{3} p^{2(k-\delta)} \left( \frac{r}{\delta} \right)^\delta \right) < 0. \quad (2.2)$$



Figure 2.3: Feature Extraction Process: (a) original image, (b) after edge detection, (c) including feature points

After expanding and rearranging, we have

$$\delta > \frac{\log \frac{1}{3} p^{2k}}{\log \frac{\delta p^2}{r}} \quad (2.3)$$

Unable to isolate the  $\delta$ , on the right hand side substitute  $k$  for  $\delta$ . In general,  $\delta$  is larger than  $k$ , and the logarithm of the two is relatively close. For reasonable vault parameters, this only increases  $\delta$  by approximately 0.3, and either does not affect the result or adds a small probabilistic safety net. ■

However, an attacker will still have to search through the space of possible polynomials. We will show in Section 2.2.4 that this complexity can still be formidable. First, however, we will define our algorithms.

### Feature Extraction

Feature extraction has been the focus of much research in past years, and this paper does not address it in detail. For the tests performed in this paper, the VeriFinger toolkit from Neurotechnologija Ltd. [31] was used to extract fingerprint features. The feature extraction process is visually represented in Figure 2.3.

In 2.3(a), one can see an image scan of a fingerprint. This is received directly from the fingerprint capture device. Various edge detection algorithms are then used to convert that information into 2.3(b). This figure features a cleaned-up version of the original scan. From there, features can be readily identified, as in 2.3(c). Each identified are in 2.3(c) represents a fingerprint *minutiae*, which is a location where a fingerprint ridge either splits or ends.

Here, we shall consider the feature extraction and alignment as a black box, yielding normalized  $(x, y)$  pixel coordinates of fingerprint minutiae. The outputs from the black box for several scans of the same finger are generally close to one another, unless the fingerprint image was severely clipped. The intra-scan variance imposes limitations on our system.

### Feature Noise

Each person's fingerprint consists of a fixed set of minutiae locations  $m_i = (x_i, y_i) \in \mathcal{M}$ . However, due to systematic errors in image capture, processing, and alignment, noise is added to each point, such that our final points are

$$m'_i = (x_i + n_{xi}, y_i + n_{yi})$$

Additionally, the processing noise may discard features or add additional features that are not present on the actual fingerprint.

The next step in the analysis is to derive a model for the noise introduced by the capture device and extraction algorithms. From this information, we can derive matching error probabilities. To simplify analysis, an additive Gaussian noise model will be assumed.

To estimate the distribution on the minutiae locations, statistical data is needed. To accomplish this, features were extracted from  $N$  sample fingerprint images of the same person, and then aligned. The data used here was associated using the bounded nearest neighbor averaging technique described in the next section. The result is a set of expected values  $(\bar{x}_i, \bar{y}_i)$  for each detected minutiae, the number  $n \leq N$  of samples having a minutiae in that neighborhood, and the variance and covariance of those  $n$  minutiae,  $(\sigma_{x,i}^2, \sigma_{y,i}^2, \rho_i)$ . Figure 2.4 shows these regions where features reliably appear.

These values will be used later to compute a bound on the possible density of chaff points for a given quantization error probability.

### Locking Set

The locking set  $\mathcal{L}$  is computed in a method similar to the method for finding minutiae variances. A user's finger is scanned and processed  $N$  times, resulting in  $N$  sets of minutiae,  $b_1, \dots, b_N$ . These are correlated using the following algorithm with distance threshold  $T$  and multiplicity threshold  $S$ :

1. let  $A$  be the set of average points with multiplicity
2. for each minutiae set  $b_i$
3.   for each minutiae  $m_j \in b_i$

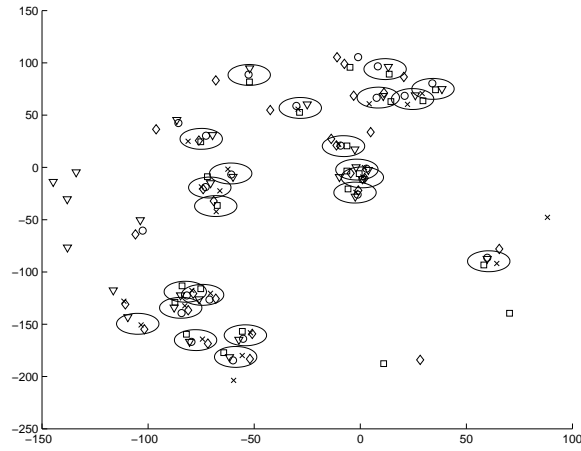


Figure 2.4: Plot of minutiae from 5 scans of the same person, with reliable regions marked

4. find element in  $n_k \in A$  such that  $|n_k - m_j| < T$
5. select closest  $n_k$  that has not already been used
6. if no matches, add  $m_j$  to  $A$  with multiplicity one
7. else add  $m_j$  to average and increase multiplicity
8.  $\mathcal{A} = \{a \in A : \text{multiplicity}(a) > S\}$

Features appearing in  $S$  or fewer scans are discarded as noise. These points generally occur in the edge regions of the image, where feature extraction is less reliable. This locking set can then be used to create the fingerprint vault.

### Chaff Points

The number and location of chaff points is limited by the variance in the fingerprint capturing and feature extraction algorithm. Chaff points cannot be placed too close to real points, or they will cause quantization problems. Given some acceptable distance  $d$  is found, chaff points can be placed anywhere as long as they are at least distance  $d$  from any real points. Additionally, there is no reason to place chaff points next to each other at any distance less than  $d$ , because an attacker can immediately ignore them as unlikely candidates, as they are so close together.

**Lemma 2** *Given elements of  $\mathbb{F}_{p^2}$  have pairwise Euclidean distance no less than  $d$ , the total number of elements  $r$  with packing density  $\rho$  is less than  $\frac{4\rho p^2}{d^2\pi}$ .*

The problem reduces to packing circles within a rectangle. The square of possible locations has area  $p^2$ , and the circles have area  $(\frac{d}{2})^2 \pi$ . As a result, the number of circles is bounded above by

$$r \leq \rho \left( \frac{p^2}{(d/2)^2 \pi} \right) = \frac{4\rho p^2}{\pi d^2} \quad \blacksquare \quad (2.4)$$

The optimal packing technique for circles is using a hexagonal lattice, and has a packing density of  $\rho = \frac{\pi}{2\sqrt{3}} \approx 0.91$  [29]. Unfortunately, this density could never be achieved, as we require the locations of chaff points to look random. If they all existed on a lattice, any discontinuities in the lattice pattern would be the real points.

Points can be randomly packed by repeatedly selecting random field elements and putting a chaff point there if it is at least distance  $d$  from all other points. This yields a packing density of  $\rho \approx 0.45$ , and is guaranteed to be random. Another technique, *random close packing* [10], could yield densities closer to  $\rho \approx 0.75$ , but these techniques have not been sufficiently studied in the two-dimensional case, and their randomness has never been quantified.

For this results to be useful, a minimum distance  $d$  between points needs to be computed in terms of the vault parameters. Here, we make a simplifying assumptions which will slightly loosen our bound, but yield simpler results: the noise distribution is spherically Gaussian, or the two axes are independent and identically distributed.

**Lemma 3** *The probability of successfully decoding a single point at distance at least  $d$  from all others using the maximum likelihood rule is*

$$P_s \leq 1 - \exp\left(\frac{-d^2}{8\sigma^2}\right) \quad (2.5)$$

The success probability is bounded below by integrating the Gaussian of distance  $d/2$  from its mean, a frequently made simplification in communication theory [4]. Here, we use polar integration on the multivariate Gaussian distribution to compute the probability. Consequently, the computation is

$$P_s \leq \int_0^{2\pi} \int_0^{d/2} \frac{1}{2\pi\sigma^2} e^{-r^2/2\sigma^2} r \, dr \, d\theta \quad (2.6)$$

which simplifies to the above expression.  $\blacksquare$

**Theorem 1** *The probability of error for decoding points at least  $\delta$  out of  $t$  points in the fuzzy vault  $\mathcal{V}(\mathbb{F}_{2^p}, r, t, k)$  is bounded below by*

$$P_e \geq \sum_{i=\delta}^t \binom{t}{i} \exp\left(-\frac{\rho p^2}{2r\pi\sigma^2}\right)^i \left(1 - \exp\left(-\frac{\rho p^2}{2r\pi\sigma^2}\right)\right)^{t-i} \quad (2.7)$$

for a given point variance  $\sigma^2$ .

The result above is essentially a combination of the first two lemmas. The vault is designed to allow some decoding errors, since there are  $t \geq \delta$  valid points available, any combination of at least  $\delta$  successful decodings is necessary. Also note that we are ignoring the probability of an error yielding another real point, rather than a chaff points. The real points used in the vault can be chosen to minimize the probability of this event.

The distance  $d$  is defined by the first lemma as

$$d^2 \leq \frac{4\rho p^2}{\pi t}. \quad (2.8)$$

Substituting for  $d$  in the  $P_s$ , we can compute  $P_e$  as

$$P_e \geq \sum_{i=\delta}^t \binom{t}{i} (P_s)^i (1 - P_s)^{t-i} \quad (2.9)$$

which expands to the given expression. ■

Figure 2.5 shows the error probability as a function of the vault size for the optimal and randomized packing methods. Also included is error probabilities from actual fingerprint data using the random packing method. For each person, five fingerprint scans were available. The first four were used to create a vault, and the fifth was used to try and unlock it. The plotted probabilities represent the fraction of data sets that had enough true points in the unlocking set to successfully unlock the fingerprint vault. The experimental results are quite close to the theoretical, which is impressive given our simplified noise model.

Note that this is the probability of being able to successfully decode, and does not deal with the complexity of actually performing that decoding.

### Unlocking Set

The unlocking set  $\mathcal{U}$  starts off initially as some set  $U$  of minutiae locations from a single scan of the user's finger. To select the elements of the unlocking

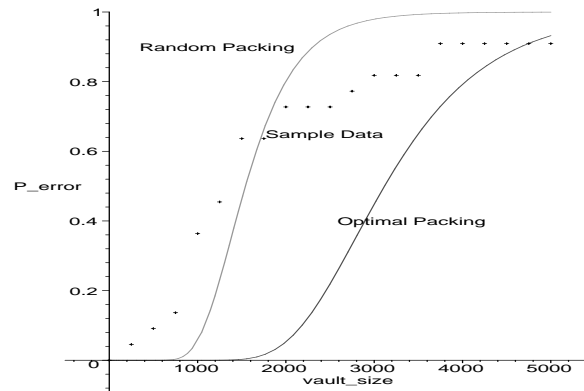


Figure 2.5: Probability of error as a function of  $r$ , using  $\sigma^2 = 9$  and  $p = 251$ , with vault parameters  $t = 40$  and  $\delta = 12$ , plotted with error probabilities using random packing in sample data sets.

set, for each point in  $U$  the user finds the closest point in the vault. If the fingerprint capture process did not introduce noise features, we should have  $\mathcal{U} \subseteq \mathcal{L}$ . However, there is some probability that a minutiae will by chance be closer to a chaff point than a true point. Also, there is a chance that  $U$  will contain spurious minutiae which was not included in  $\mathcal{L}$ .

This is where Reed-Solomon codes comes into play. For any  $n$  points, we can determine the polynomial, or Reed-Solomon codeword in this case, if at least  $\frac{\delta+n}{2}$  of those points are correct. If  $n$  is reasonably close to  $r$ , then very few attempts will be required to compute the polynomial.

## 2.2.4 Unlocking Complexity

Vault unlocking can be viewed in two contexts. The first is the complexity of a valid user unlocking a vault with a matching fingerprint image. One goal is to minimize this complexity. The second context is the complexity of an attacker without fingerprint information trying to crack the vault. We wish to maximize this complexity while the attacker wishes to minimize it.

There are two obvious techniques for unlocking the vault. The first is the brute-force method, or  $\mathbf{bf}(r, t, k)$ , where  $r$  is the total number of points,  $t$  is the number of real points, and  $k$  is the degree of the polynomial. For an attacker,  $r$  and  $t$  are the same as the ones in the vault parameter, however for a valid user,  $r$  is the size of their unlocking set and  $t$  is the number of non-chaff points in that set.

**Theorem 2** *The complexity of the  $\mathbf{bf}(r, t, k)$  problem using a suitable  $\delta$  to ensure a unique result is  $\mathcal{C}_{bf} = \binom{r}{\delta} \binom{t}{\delta}^{-1}$ .*

The proof is a fairly straight-forward combinatorics argument. In the brute-force method, we must find  $\delta$  points that interpolate a degree  $k$  polynomial. There are  $\binom{r}{\delta}$  sets of any  $\delta$  points. Of those sets,  $\binom{t}{\delta}$  will yield successful results, as all  $\delta$  points will exist on the degree  $k$  polynomial. The quotient of the two is the expected number of trials required to open the vault. ■

**Example 1** *Consider a vault over  $\mathbb{F}_{2512}$  with  $r = 1000$  total points and  $t = 40$  real points over a degree  $k = 8$  polynomial. Under Corollary 1,  $\delta = 12$ . Using Theorem 2, the complexity of an attacker breaking the vault is approximately  $2^{58}$  polynomial interpolations.*

**Example 2** *Consider the same vault, only a valid user intersects their unlocking set with the vault to obtain  $r = 30$  points,  $t = 22$  of which are real points. With such a small vault, obviously  $\delta = k + 1 = 9$ . Using Theorem 2, the complexity of a valid user unlocking the vault is approximately  $2^7$  polynomial interpolations.*

The second example illustrates that a brute-force decoding algorithm is less than ideal a valid user. Another method of unlocking the vault is through the use of a Reed-Solomon decoder. In the  $\mathbf{rs}(r, t, n, \delta)$  problem,  $r$ ,  $t$ , and  $\delta$  have the same meanings as before, and  $n$  is the size of the Reed-Solomon codewords involved.

**Theorem 3** *The complexity of the  $\mathbf{rs}(r, t, n, \delta)$  problem over  $\mathbb{F}_{p^2}$  is*

$$\mathcal{C}_{rs} = \binom{r}{n} \left( \sum_{i=\max(\frac{n+\delta}{2}, n-r+t)}^{\min(n,t)} \binom{r-t}{n-i} \binom{t}{i} \right)^{-1} \quad (2.10)$$

*such that  $n$  satisfies  $\delta \leq n \leq \min(r, 2t - \delta)$  and  $n|(p^2 - 1)$ .*

The argument here is similar to the proof for brute-force. We select and try codewords of size  $n$ . There are  $\binom{r}{n}$  such codeword selections. The number of such codewords that succeed is a more difficult question to answer.

In general, our Reed-Solomon code requires  $\frac{n+\delta}{2}$  elements to successfully produce the degree  $k$  polynomial interpolated by at least  $\delta$  points. As a

result,  $\frac{n+\delta}{2} \geq t$  (or  $r$  if  $r$  is relatively small). Since  $n \geq \delta$ , the overall condition on  $n$  is derived:  $\delta \leq n \leq \min(r, 2t - \delta)$ .

How many sets of  $n$  points will succeed? Well, there must be at least  $\nu = \frac{n+\delta}{2}$  real points and no more than  $n - \nu$  chaff points. Given we have  $i$  real points where  $\nu \leq i \leq n$ , there are  $\binom{r-t}{n-i}$  ways to choose the chaff points and  $\binom{t}{i}$  ways to select the real points. This results in the summation

$$\sum_{i=\max(\nu, n-r+t)}^{\min(n,t)} \binom{r-t}{n-i} \binom{t}{i}. \quad (2.11)$$

The additional constraints on  $i$  guarantee that we never select more chaff points or real points than we actually have. ■

**Corollary 2** *The complexity of  $\mathbf{bf}(r, t, \delta) = \mathbf{rs}(r, t, \delta, \delta)$ , or taking  $n = \delta$  reduces Reed-Solomon unlocking into a brute-force unlocking.*

Since  $\delta \leq t \leq r$ , the summation ranges can be determined:  $\min(\delta, r) = \delta$ ,  $\max(\delta, \delta - r + t) = \delta$ . Since the summation only operates with  $i = \delta$ , the combinations become  $\binom{r-t}{0} \binom{t}{\delta} = \binom{t}{\delta}$ . Thus, the overall expression simplifies to  $\binom{r}{\delta} \binom{t}{\delta}^{-1}$ . ■

**Corollary 3** *For  $r \gg t$ ,  $\mathbf{bf}(r, t, \delta) \leq \mathbf{rs}(r, t, n, \delta)$ , for all  $n \geq \delta$ , or unless an attacker can eliminate a significant number of chaff points of a locked vault, he or she can do no better than a brute-force attack.*

We wish to select an  $n$ , such that  $k \leq n \leq 2t - k$  that minimizes  $\mathcal{C}_{\text{rs}}$ . To do this, we shall expand the equation for  $\mathcal{C}_{\text{r}}$ , and examine the terms that most significantly contribute to the overall complexity.

First, for  $r \gg t$ , the terms can be slightly simplified.

$$\mathcal{C}_{\text{rs}} = \binom{r}{n} \left( \sum_{i=(n+\delta)/2}^{\min(n,t)} \binom{r-t}{n-i} \binom{t}{i} \right)^{-1} \quad (2.12)$$

Now, expanding the summation:

$$\binom{r}{n} \left( \binom{r-t}{(n-\delta)/2} \binom{t}{(n+\delta)/2} + \dots \right)^{-1} \quad (2.13)$$

However, notice that we really are only interested in the first term of the summation. For  $r \gg t$ , the first combination is significantly larger than

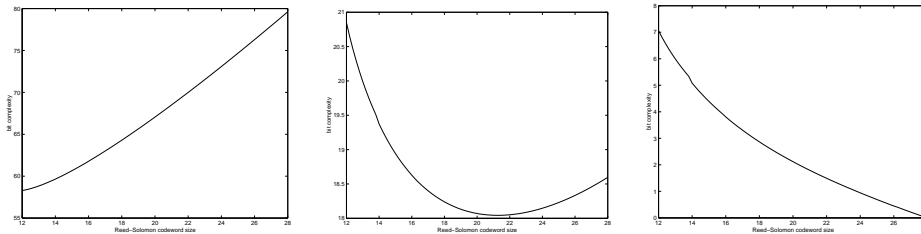


Figure 2.6: Log of complexity for Reed-Solomon decoding as a function of codeword size; (a) complexity of full attack,  $\text{rs}(1000, 40, n, 12)$ ; (b) complexity of partial information attack,  $\text{rs}(120, 40, n, 12)$ ; (c) complexity of legitimate unlocking,  $\text{rs}(30, 22, n, 12)$ .

the second. Additionally, the second term in the summation will be approximately  $r$  times smaller. As a result, if we write out the dominant terms:

$$\binom{r}{n} \binom{r-t}{(n-\delta)/2}^{-1} \quad (2.14)$$

To minimize our complexity, we wish to minimize the numerator and maximize the denominator. However, these are conflicting goals. Fortunately, the first term contributes approximately  $r^t$  times more to the result than the second term. Consequently, to minimize  $\binom{r}{n}$ , select the smallest  $n$  possible. ■

Let's examine the previous two examples in the context of Reed-Solomon decoding. Figure 2.6 illustrates the complexity of a full-scale attack, an attack where partial fingerprint information is known, and a legitimate unlocking of the vault. We can see that depending on the relationship of  $r$  and  $t$ , the optimal method for unlocking the vault can change. By using a Reed-Solomon decoder, a valid user can now unlock the vault in 1 or 2 tries, while an attacker can still do no better than a brute-force attack.

If an attacker is able to eliminate many of the chaff points, presumably through side knowledge of some of the fingerprint characteristics, finding the optimal attack now becomes more interesting. The minimum complexity is no longer one of the bounding cases.

In this paper, we assume that we can choose chaff points in such a way as to confuse the attacker and force him to consider all points. For the most part we can disregard attacks where an attacker can eliminate certain chaff points based on the probable minutiae configurations. Research [25, 26, 28] indicates that the probability of two people having the same fingerprint is approximately  $1 \times 10^{-80}$ . Assuming fingerprints are equiprobable, this cor-

responds to  $-\log_2(1 \times 10^{-80}) \approx 265$  bits of entropy. Thus, for a complexity-theoretic attack, the minutiae entropy is sufficiently large, thus trivializing attacks which take probable minutiae configurations into account. This also indicates we could never achieve more than 265-bit security, regardless of vault parameters.

### 2.2.5 Empirical Results

Throughout the paper, the term “reasonable vault parameters” has been used repeatedly. Here, we use actual fingerprint data to determine what “reasonable” is. Each of the vault parameters,  $p$ ,  $r$ ,  $t$ , and  $k$  has limitations placed on it by the behavior of actual fingerprint data.

The locking and unlocking algorithms were implemented in MATLAB, and sample fingerprint data was used to test the error probabilities. Four scans of the same individual were used to create a vault, and a fifth used to try and unlock it. The number of true points in the unlocking sets for these real vaults was used to validate the statistical models.

The field,  $\mathbb{F}_q$ , defines the underlying mechanics of our entire system. Throughout, we have been using  $\mathbb{F}_{p^2}$ , for prime  $p$ . In general, we wish to represent a feature pixel location. For the examples presented so far,  $p = 251$  was used. This way, minutiae locations can be stored in 16-bit numbers, and  $251^2 - 1 = 63000 = 2^3 \cdot 3^2 \cdot 5^3 \cdot 7$ , a very smooth number, yielding many choices for the Reed-Solomon codeword size.

Increasing the fingerprint image resolution and consequently the field size has little effect on the resulting security. As the resolution increases, so does the minutiae variance,  $\sigma$ . These two parameters cancel one another out, making the underlying field selection based more on convenience than security.

The number of real points,  $t$ , is the size of the locking set. The algorithm described earlier takes several scans of the same person and locates minutiae that appear in two or more of the scans. This algorithm was implemented in MATLAB and used to create various locking sets. For 5 scans of each person and  $S = 1$ , we obtain locking sets which ranged from 25 to 60 points, with mean 38 and standard deviation 11.

The degree of the polynomial the vault protects is bounded below by the amount of data we wish to encode in it. Each coefficient is an element of  $\mathbb{F}_{251^2}$ , and can therefore hold 15.9 bits of information. As a result, a 128-bit key can be encoded using 9 coefficients, or in a degree 8 polynomial. Consequently, we shall consider  $k \geq 8$ .

The total number of points  $r$  depends on the number of chaff points

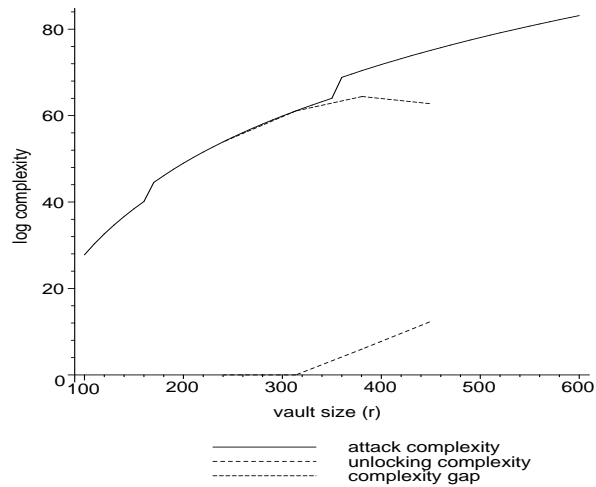


Figure 2.7: Vault performance as a function of vault size, with  $k = 14$ ,  $\tau = 20$ , and  $t = 38$  over  $\mathbb{F}_{251^2}$

added to the vault, and is a function of the desired error probability. Figure 2.5 gave the probabilities for a particular set of input values. Here, we shall examine this trade-off in more detail.

First, examine how the vault performs as a function of its size. Figure 2.7 shows both the complexity of a normal user, and the complexity of an attacker for a vault with 38 real points over a degree 14 polynomial. We can see that as the total number of points increases, so do both complexities. In order to keep user complexity to a minimum, we shall select the largest value of  $r$  such that the user has zero complexity.

The other key parameter that can be varied to alter our vault performance is  $k$ , the degree of our polynomial. Figure 2.8(a) shows the attack complexities as a function of  $k$ . This complexity was computed by first finding the maximum number of points  $r$  such that the user has zero complexity, and from there computing  $\delta$ , the minimum number of points interpolating our polynomial in order to guarantee success. Using  $r$  and  $\delta$ , the difficulty of a brute-force attack can be computed.

Figure 2.8(b) is essentially a reality check on our selection of  $\tau$ , the size of our unlocking set. For a given  $\tau$  and  $k$ , real fingerprint data was again used to compute the probability of successfully unlocking the vault. It can be seen that given  $\tau = 20$ , approximately 20 to 30 percent error occurs. From a user's perspective, this means that every couple times they access their

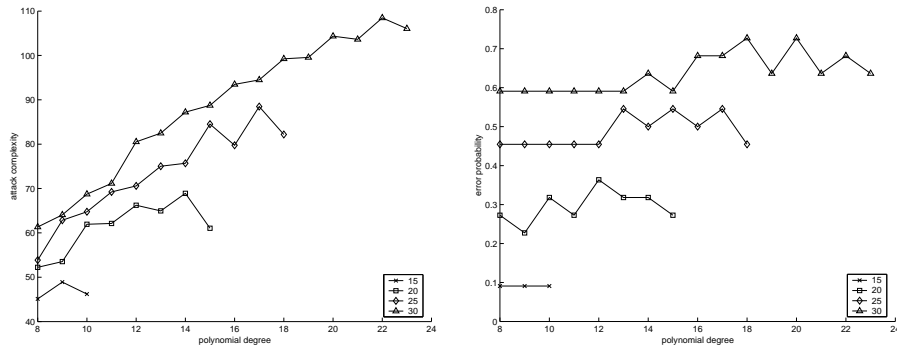


Figure 2.8: Vault performance as a function of  $k$  and  $\tau$ : (a) attack complexities as a function of  $k$  for various  $\tau$ ; (b) decoding failure as a function of  $k$  for various  $\tau$

smartcard, a second fingerprint scan will be required in order to successfully unlock the vault. This seems reasonable given that we expect that the false positive rate to be infinitesimally small. The corresponding curve in 2.8(a) indicates that the maximum complexity is  $2^{69}$  for  $k = 14$ .

Consequently, over  $\mathbb{F}_{251^2}$  we have determined the optimal vault to be:

- polynomial:  $k = 14$ ,  $\delta = 17$
- chaff points:  $r = 313$ ,  $d = 10.7$
- attack complexity:  $2^{69}$

## 2.2.6 Conclusion

We have considered the practical implications of using fingerprint information to secure a smartcard. Because fingerprints are often inconsistent, we must resort of a fuzzy scheme for storing the secret key. We show that with real-life parameters, it is impossible to ensure the security envisioned by Juels and Sudan. However, we define a modified scheme called the *fingerprint vault*, provide associated algorithms and a mechanism for finding optimal vault parameters. Parameters are provided which makes retrieving the secret  $2^{69}$  times more difficult for the attacker than a legitimate user.

There are a couple ways by which security may be improved. An obvious way is to use multiple fingerprints to store a longer private key which could be hashed down to the appropriate length. Another way is to improve the detection and extraction algorithms so as to lower  $\sigma$ , allowing us to pack in more chaff points.

## Chapter 3

# SmartCard

Today smartcards are used mainly for identification and storing user information. Sometimes, they are also used to store private keys and to execute cryptographic operations. In this project, smartcards store personal information, such as identification, ticket numbers, and passenger status.

Smartcard modules include two parts: a JavaCard applet and a wrapper. The JavaCard applet is a program installed on the smartcard and provides data structures and methods. Since only a single program can interface with the card at a time, we wrote a wrapper so that other modules can access the JavaCard applet.

In our cursory implementation, the smartcard module is used in each stage of ticketing, check-in, and checkpoint verification. When a passenger is issued a ticket, it assigns the appropriate ticket numbers to the smartcard. When the passenger checks in, it checks that the ID and ticket numbers are consistent with those in database and update the status. When the passenger passes the security checkpoint, it checks he or she has the right ticket number and status.

### 3.1 Structure

- Application
- Wrapper (imports CyberFlex Access API 4.3 - smartcard interface)
- JavaCard Applet (imports JavaCard API 2.2)

## 3.2 Data Structures

In our implementation, the applet on the smartcard was organized as follows:

- `TxBuffer []`: buffer containing the id, maximum length is 50.
- `TxTicket []`: buffer containing the ticket numbers, delimited by comma, maximum length is 50.
- `TxFlag []`: flag representing the status of passenger.

## 3.3 API Methods

The JavaCard applet exports the following methods:

- `public static void install(byte buffer[], short offset, byte length)` – called when the applet is installed; creates an instance of the applet
- `public boolean select()` – called when the applet is selected
- `public void process(APDU apdu)` – method of the applet; dispatches messages to the class methods depending on the instruction type

The wrapper exports the following methods:

- `public static boolean SCConnect(String reader)` – connects the smart card reader
- `public static boolean SCSetFlag(byte flag)` – sets the smart-card status flag
- `public static byte SCGetFlag()` – gets the current value of the flag
- `public static boolean SCSetString(String SendString)` – sets the ID string
- `public static boolean SCSetTicketNo(String SendString)` – sets the ticket number
- `public static int SCGetStringLength()` – gets the length of the current ID; required because applications needs to provide the length of the string when it wants to retrieve the string

- `public static int SCGetTicketLength()` – gets the length of the current ticket
- `public static String SCGetString(int length)` – gets the current ID
- `public static String SCGetTicketNo(int length)` – gets the current ticket number
- `public static void SCDisconnect()` – disconnects from the smartcard reader

### 3.4 Comments on ECC

Since the software implementation of ECC on the smartcard was impractical, smartcard were simply used for storage in our cursory airport implementation. Clearly, the future direction will be to implement ECC algorithms in hardware on the smartcard, so that it can store the private key and play a major role in authentication by being integrated with fingerprint verification process. As more processor power becomes available on smartcards, the software implementation of ECC algorithms could also be practical.



## Chapter 4

# SESAME

The SESAME (Secure European System for Applications in a Multivendor Environment) protocol provides underlying security mechanism that protect the data being used over our distributed system. SESAME is used to provide cryptographic protection of the underlying data. Every entity in the network, including services and users, are authenticated via public key authentication protocols. These cryptographic data from these authentications are then used to create symmetric session keys to protect every transaction.

While we implemented the complete SESAME protocol, we focused primarily on authentication. SESAME supports a wide variety of access control mechanisms, however we chose to implement authorization separately using a centralized RBAC system (see chapter 7).

SESAME provides the following:

- network single signon;
- distributed access control using digitally signed tokens;
- full cryptographic protection of exchanges between users and remote applications;
- support for multiple domain operation with different security policies;
- robust scalability for even the largest of networks through its use of public-key technology;
- interoperability through international standards; and
- support for the widely used Generic Security Service API (GSS-API).

In this chapter, we describe the protocol, prove its security features, and describe our implementation.

## 4.1 Protocol

To access the distributed system, a user first authenticates to an authentication server (AS) to get a cryptographically protected token used to prove his or her identity. The user then presents the token to a privilege attribute server (PAS) to obtain a guaranteed set of access rights contained in a privilege attribute certificate (or PAC). The PAC is a specific form of Access Control Certificate that conforms to ECMA and ISO/ITU-T standards.

The PAC is presented by the user to a target application whenever access to a protected resource is needed. The target application makes an access control decision according to the user's security attributes from the PAC, and other access control information (for example an Access Control List) attached to the controlled resource.

To provide cryptographic protection of interchanged data, SESAME needs to establish temporary secret cryptographic keys shared pairwise between the participants. Kerberos key distribution protocols can be used for this, but they can also be either supplemented, or where appropriate completely replaced by public key technology.

## 4.2 Relation to Kerberos

Similar work, aimed specifically at UNIX systems, has been done at MIT which has developed a basic distributed single signon technology called Kerberos. Kerberos has been proposed as an Internet standard [15].

In the light of this work, the SESAME project decided that in its early implementation some of the SESAME components would be accessible through the Kerberos V5 protocol (as specified in RFC1510), and would use Kerberos data structures, as well as new SESAME ones. This has shown unequivocally that a product quality approach reusing selected parts of the Kerberos specification is workable and that a world standard is possible incorporating features of both technologies. SESAME adds to Kerberos heterogeneity, sophisticated access control features, scalability of public key systems, better manageability, audit, and delegation.

## 4.3 GSS-API

Another important development in the field of open distributed system security has been the Generic Security Services Application Program Interface (GSS-API). This interface hides from its callers the details of the specific un-

derlying security mechanism, leading to better application portability, and moving generally in the direction of a better interworking capability.

The GSS-API also completely separates the choice of security mechanism from choice of communications protocol. A GSS-API implementation is viable across virtually any communications method. GSS-API is an Internet and X/Open standard. SESAME is accessed through the GSS-API, extended to support features needed to provide distributed Access Control.

## 4.4 Formal Analysis

In an effort to show our overall system is secure, we would like to prove that the underlying cryptographic layer is secure. To accomplish this, we perform an analysis of SESAME using BAN logic.

### 4.4.1 Preliminaries

Here are the basic notations of the formalism in BAN-logic. Typically,  $A$ ,  $B$ , and  $S$  denote specific principals;  $K_{ab}$ ,  $K_{as}$ ,  $K_{bs}$  denote specific shared keys;  $K_a$ ,  $K_b$ , and  $K_s$  denote specific public keys, and  $K_a^{-1}$ ,  $K_b^{-1}$ , and  $K_s^{-1}$  denote the corresponding secret keys;  $N_a$ ,  $N_b$ , and  $N_c$  denote specific statements. We have  $P$ ,  $Q$ , and  $R$  range over principals;  $X$  and  $Y$  over statements,  $K$  ranges over encryption keys.

$P \models X$  :  $P$  believes  $X$ , or  $P$  would be entitled to believe  $X$ . The principal  $P$  may act as though  $X$  is true.

$P \triangleleft X$  :  $P$  sees  $X$ . Someone has sent a message containing  $X$  to  $P$ , who can read and repeat  $X$ .

$P \vdash X$  :  $P$  once said  $X$ . The principal  $P$  at some time sent a message including the statement  $X$ .  $P$  believed  $X$  when he sent the message.

$P \Rightarrow X$  :  $P$  has *jurisdiction* over  $X$ . The principal  $P$  is an authority on  $X$  and should be trusted on this matter.

$\sharp(X)$  : The formula  $X$  is *fresh*.  $X$  has not been sent in a message at any time before the current run of the protocol.

$P \stackrel{K}{\leftrightarrow} Q$  :  $P$  and  $Q$  may use the *shared key*  $K$  to communicate.

$\stackrel{K}{\mapsto} P$  :  $P$  has  $K$  as a *public key*. The secret key  $K^{-1}$  will never be discovered by any principal except  $P$ , or principals trusted by  $P$ .

Table 4.1: Simplified description of the SESAME protocol

Message 1	$A \rightarrow AUTH$	$A, \langle T_a \rangle_{K_a^{-1}}$
Message 2	$AUTH \rightarrow A$	$\{PAS, \{TGT\}_{K_{auth,pas}}, K_{a,pas}, \langle T_{auth} \rangle_{K_{auth}^{-1}}\}_{K_a}$
Message 3	$A \rightarrow PAS$	$\{A, T'_a\}_{K_{a,pas}}, \{TGT\}_{K_{auth,pas}}$
Message 4	$PAS \rightarrow A$	$\{\langle T_{pas}, PAC \rangle_{K_{pas}^{-1}}\}_{K_{a,pas}}$
Message 5	$A \rightarrow B$	$\{A, \langle T_{pas}, PAC \rangle_{K_{pas}^{-1}}\}_{K_{a,b}}, \{A, T''_a, K_{a,b}\}_{K_b}$
Message 6	$B \rightarrow A$	$\langle \{T''_a + 1\}_{g(K_{a,b})} \rangle_{f(K_{a,b})}$

$P \stackrel{X}{\Leftarrow} Q$  : The formula  $X$  is a secret known only to  $P$  and  $Q$ .

$\{X\}_K$  : This represents the formula  $X$  encrypted under the key  $K$ .

$\langle X \rangle_Y$  : The represents  $X$  combined with the formula  $Y$ .

#### 4.4.2 SESAME

Table 4.1 describes a simplified version of the protocol, and table 4.2 gives an in-depth explanation of the notation used.

#### 4.4.3 Protocol Analysis

First we state the assumed initial beliefs of the players:

Table 4.2: Description of notation

<b>Message (1)</b>	<b>Client requests ticket-granting ticket</b>
$A$ :	User tells $AUTH$ name $A$ from current client
$T_a$ :	User tells $AUTH$ the time and random number together as $T_a$
$K_a^{-1}$ :	$A$ 's private key
<b>Message (2)</b>	<b>PAS returns ticket-granting ticket</b>
$TGT$ :	Ticket to access PAS server, contains $K_{a,pas}$
$T_{Auth}$ :	sends back the current time along with $TGT$ lifetime
$K_a$ :	$A$ 's public key
$K_{a,pas}$ :	shared symmetric key between $A$ and $PAS$ generated by $AUTH$
$K_{auth,pas}$ :	Symmetric key shared by $AUTH$ and $PAS$
$K_{auth}^{-1}$ :	$AUTH$ 's private key. Confirms that the nonce was sent from $AUTH$
<b>Message (3)</b>	<b>Client request PAC</b>
$T'_a$	Data and time
$TGT$	Assures $PAS$ that that user has been authenticated by $AUTH$
$K_{a,pas}$	Symmetric key shared by $A$ and $PAS$ issued by $AUTH$
<b>Message (4)</b>	<b>PAS returns PAC</b>
$A$	Confirms that this $PAC$ is for $A$
$T_{pas}$	Time stamp and lifetime of the $PAC$
$PAC$	Privilege Attributes Certificates
$K_{pas}^{-1}$	$PAS$ private key. Verifies that $PAC$ was issued by $PAS$
$K_{a,pas}$	Symmetric key shared by $A$ and $PAS$
<b>Message (5)</b>	<b>Client requests service</b>
$PAC$	Privilege Attributes Certificates
$K_{a,b}$	Symmetric key shared by $A$ and $B$ generated by $A$
$K_b$	$B$ 's public key
$T_{pas}$	Nonce (data)
$T''_a$	Time stamp sent by $A$
<b>Message (6)</b>	<b>Authentication of server to client</b>
$T''_a + 1$	Assures $A$ that this is not a replay of an old message
$f(K_{a,b})$	Integrity key
$g(K_{a,b})$	Confidentiality key

$$\begin{array}{lll}
A \models^{K_a} A & B \models^{K_b} B & B \models (A \Rightarrow K_{a,b}) \\
A \models^{K_b} B & B \models^{K_{pas}} PAS & B \models (PAS \Rightarrow PAC) \\
A \models^{K_{auth}} AUTH & & A \models \#(T_{pas}) \\
A \models^{K_{pas}} PAS & A \models (AUTH \Rightarrow TGT) & A \models \#(TGT) \\
AUTH \models^{K_a} A & A \models (AUTH \Rightarrow K_{a,pas}) & A \models \#(T_{auth}) \\
PAS \models^{K_{pas}} PAS & A \models (A \Rightarrow K_{a,b}) & PAS \models \#(TGT) \\
AUTH \models AUTH \xleftrightarrow{K_{auth,pas}} PAS & PAS \models (AUTH \Rightarrow TGT) & B \models \#(T_{pas}) \\
PAS \models AUTH \xleftrightarrow{K_{auth,pas}} PAS & PAS \models (AUTH \Rightarrow K_{a,pas}) & 
\end{array}$$

Now we can conclude the following for each message:

**message 1:** *A* identifies him/herself to *AUTH* by signing a nonce with her private key. hence

$$AUTH \triangleleft A, \langle T_a \rangle_{K_a^{-1}}$$

*AUTH* can verify the identity of *A* by simply decrypting  $\langle T_a \rangle_{K_a^{-1}}$  with *A*'s public key.

**message 2:** *A* receives a message from *AUTH* encrypted with him/her public key, and recovers *TGT* as well as  $K_{a,pas}$ . Hence now:

$$A \models A \xleftrightarrow{K_{a,pas}} PAS$$

since *A* believes *AUTH* has authority over  $K_{a,pas}$ .

**message 3:** *A* requests *PAC* from *PAS* by sending *TGT* which contains  $K_{a,pas}$ . This implies

$$PAS \models A \xleftrightarrow{K_{a,pas}} PAS$$

**message 4:** *PAS* sends the *PAC* to *A* along with the current time and lifetime. *A* knows that it was indeed from *PAS* since it has been signed with *PAS*'s private key.

Table 4.3: Description of the Kerberos protocol

Message 1	$A \rightarrow AUTH$	$A, PAS, T_a$
Message 2	$AUTH \rightarrow A$	$\{PAS, TGT, T_{auth}, K_{a,pas}\}_{K_{a,auth}}$
Message 3	$A \rightarrow PAS$	$A, B, TGT$
Message 4	$PAS \rightarrow A$	$\{T_{pas}, K_{a,b}, B, \{T_{pas}, K_{a,b}, A\}_{K_{b,pas}}\}_{K_{a,pas}}$
Message 5	$A \rightarrow B$	$\{T_{pas}, K_{a,b}, A\}_{K_{b,pas}}, \{A, T_a\}_{K_{a,b}}$
Message 6	$B \rightarrow A$	$\{T_a + 1\}_{K_{a,b}}$

**message 5:** Now  $A$  establishes a connection with  $B$ , and generates a symmetric key for them to share. Hence

$$A \models A \stackrel{K_{a,b}}{\leftrightarrow} B$$

$B$  receives the message from  $A$  decrypts the second part using his/her private key. Verifying that  $A$  has privileges,

$$B \models A \stackrel{K_{a,b}}{\leftrightarrow} B$$

By decrypting the first part, now we have

$$B \models A \models A \stackrel{K_{a,b}}{\leftrightarrow} B$$

The fourth message simply assures  $A$  that  $B$  believes in the key, and received  $A$ 's last message. The final result is:

$$\begin{array}{ll} A \models A \stackrel{K_{a,b}}{\leftrightarrow} B & B \models A \stackrel{K_{a,b}}{\leftrightarrow} B \\ A \models B \models A \stackrel{K_{a,b}}{\leftrightarrow} B & B \models A \models A \stackrel{K_{a,b}}{\leftrightarrow} B \end{array}$$

#### 4.4.4 Kerberos

Here we will briefly recall the Kerberos authentication protocol. Table 4.3 describes a simplified version of the protocol.

#### 4.4.5 Kerberos versus SESAME

Kerberos uses symmetric key cryptography and requires shared secrets between realms. This limits the extent to which Kerberos can be used realistically, since it raises scalability issues. It is vulnerable to password-guessing attacks, and depends on the security of the client workstation. Kerberos does not support access control. A client can be authenticated to the server but that server needs to be preconfigured with the privilege attributes of the client. In some sense, Kerberos provides an identity based control.

SESAME is an authentication and access control protocol, that also supports communication confidentiality and integrity. It provides public key based authentication along with the Kerberos style authentication, that uses symmetric key cryptography. Sesame supports the Kerberos protocol and adds some security extensions like public key based authentication and an ECMA-style Privilege Attribute Service.

### 4.5 Implementation

Rather than starting from scratch, we began with a partial implementation called UIUC-TINYSESAME from another research group here at the University of Illinois. They partially completed the SESAME authentication protocol by extending a Java Kerberos implementation. After fixing several security flaws with the authentication phase due to protocol implementation errors, we completed the protocol. The only part not fully implemented was the PAC protection method since it was not necessary for our access control model.

The most important API methods in SESAME are:

- Authenticate method in SecurityServices.java – Authentication given userID, the returned value is the credential. If the credential is null, authentication fails.
- Init method in GSSContext.java – Initiate a security context.
- Accept method GSSContext.java – Accept a security context. After accepting the security context, a security context is established.
- Wrap method in GSSContext.java – Encrypt the data using shared secrete key.
- Unwrap method in GSSContext.java – Decrypt the data using shared secret key.

## Chapter 5

# Distributed Computing using Jini

We use Jini as the distributed computing network service. The main reason to choose Jini is that our project is written by Java and Jini is particularly designed for Java. The benefits of Jini are:

- distributed network system;
- shared resources; and
- plug-and-play discovery of new resources and objects on network.

The architecture of Jini includes:

- remote interfaces for clients to integrate;
- unnecessary remote upgrades;
- autodiscovery of backend services; and
- communication between clients and the backend services

Our main goal was to secure the communication between clients and services using the SESAME protocol. SESAME allows Jini to also be secured and it provides each terminal authentication based on public key infrastructure. Since the Jini protocol has no notion of a session, establishing a SESAME secure context and creating session keys was difficult. Thus, our first task was to add a cryptographically protection concept of sessions to Jini. This was done by:

- client and server both implement a SecureContextData (SCD) object
- server uses hash tables to keep multiple SCD
- $TC = \{\text{byte}[\text{sessionID}+\text{seqNum}]\}$  encrypted by the sessionKey
- increments sequence number to avoid replay attacks
- client calls a function `backend.function([TC, argument1, argument2, ...]sk`

All arguments along with sessionID and sequence number are concatenated into a byte array and encrypted using the session key. The function is called `backend.function(sessionID, combinedArgs)`.

## Chapter 6

# Database Security

The database is an important part of any distributed system. In our implementation, any information about the current, past, and future status of the user is stored here. As a result it was important that we developed a secure database operations system. Thus we wrote a wrapper that sits around around the database to interact with SESAME using the GSS-API and ultimately establish a secure context with anyone who wants to query the database.

### 6.1 Motivation

We started off by making changes to the MySQL code. Since MySQL is written in C and SESAME was written in Java, this integration was not a simple task. As a result MySQL could not directly make calls to the GSS-API. We made changes to MySQL using the Java Native Interface (JNI). JNI allows C programs to call Java methods. However we ran into problems with that. The Java threads and POSIX threads were not working properly together. Thus we decided to use a wrapper.

### 6.2 Java Wrapper

The wrapper is completely written in Java and uses MySQL-JConnector driver. JConnector is a Java driver written to be used in Java programs. The wrapper is written in such a way that any database can be used. The wrapper runs on the same machine as the database. The database requires root access to make queries and the wrapper has the root access. No other user is allowed to make queries to the database. Thus in order to hack into

the database, a hacker would have to be a root user of the database. The only way to query the database is to first establish a secure context with the wrapper and then make queries. Thus the security of this implementation is as secure as the SESAME protocol itself. Any user can connect to the database as long as they are able to establish a secure context.

Once the secure context is established, the wrapper receives a query and unwraps it. Then the query is sent to the database. If the query expects a result back then the result is wrapped (i.e. encrypted) and sent to the user else if the query was an update, an acknowledgment is sent back.

### 6.3 Implementation

In order to transfer SQL results on the network we have used the Rowset object provided with java.sql package. Rowsets make it easy to send tabular data over a network. They can also be used to provide scrollable result sets or updatable result sets when the underlying JDBC driver does not support them. These are the main uses for a rowset. Assuming the SQL result is stored in Resultset object `rs`, following should be done in order to send the data over the network.

```
Resultset rs;
//Now perform the SQL query and get the result into rs.
```

```
CachedRowSet crs = new CachedRowSet();
crs.populate(rs);
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(crs);
byte [] objInBytes = baos.toByteArray();
```

Then send `objInBytes` over the network.

```
byte[] outTok = ctxt.wrap(objInBytes, 0, objInBytes.length, null); //The wrapping
```

```
if (outTok!=null) sendToken(outTok, outTok.length);
```

```
public int sendToken (byte tok[], int len)
{
```

```

byte buf[] = new byte [4];

buf[0] = (byte)((len >> 24) & 0xff);
buf[1] = (byte)((len >> 16) & 0xff);
buf[2] = (byte)((len >> 8) & 0xff);
buf[3] = (byte) (len & 0xff);

try {
    clientOutput.write (buf, 0, 4);
    clientOutput.write (tok, 0, len);
    clientOutput.flush ();
} catch (Exception e) {
    System.out.println("while sending token: " + e.toString());
}
return 0;
}

```

On the receiving side do the following.

```

byte[] inTok = readToken();
public byte [] readToken ()
{

    int i = 0, len = 0;
    int ch;

    // Read the length of the token first
    while (i++ < 4) {
        try {
            //System.out.println("inside receive result 9" );
            ch = cln.serverInput.read ();
            len = (len << 8) + ch;
        } catch (Exception e) {
            System.out.println("error 1 "+e.toString());
        }
    }
    System.out.println("inside receive result 9.0.1" );
    rt.gc();
    byte buf [] = new byte [len];
}

```

```

System.out.println("inside receive result 9.0.2" );
try {
    //System.out.println("inside receive result 9.1" );
    i = cln.serverInput.read (buf, 0, len);
} catch (Exception e) {
    System.out.println("error 2 "+e.toString());
}

if (i > 0) {
    byte tok [] = new byte [i];

    System.arraycopy(buf,0,tok,0,i );

    return tok;
}
else return null;
}

byte[] buff = ctxt.unwrap(inTok, 0, inTok.length, null);
ByteArrayInputStream bis = new ByteArrayInputStream(buff);
ObjectInputStream ois = new ObjectInputStream(bis);
Object obj = ois.readObject();
CachedRowSet crs = new CachedRowSet();
crs = (CachedRowSet) obj;

```

Now use `crs` as any other `CachedRowSet` object to get the SQL query data.

## 6.4 Future Work

The database right now is MySQL however for a full scale system to work properly, we would need a commercial database like Oracle or Microsoft SQL Server. Although the wrapper does not require much change when new database is used, still it would be better if it is tested against these other databases first.

## Chapter 7

# Role-Based Access Control

In this research project, we needed an RBAC server to perform access control and authorization. The authorization in our cryptographically secure architecture for airport security was based on role-based access control. As far as we know, RBAC is not widely used in the industry (such as in the Department of Defense). One of the aims of this research project is to demonstrate the natural extension and applicability of RBAC to industries, such as the airport and airline.

Initially, we were supposed to use Boeing's RBAC implementation; however, due to unforeseen bureaucratic delays and issues, we were unable to get a copy of their software. We later investigated RBAC software on NIST's website. Although its support for RBAC was comprehensive (although not as complete as Boeing's), the software was implemented in perl (since it was intended to be used on a Web server). Since we needed a version in Java (most of our software was written in Java), we decided to write our own implementation of RBAC.

Our implementation provides much support for RBAC, although, it is by no means complete. Administrators can add and delete operations; add, delete, or modify roles; and add, delete, or modify users. Users can be assigned any subset of roles and can activate any subset of the roles they have been authorized to assume. While our implementation has support for the basic RBAC features, possible future work includes adding support for role hierarchy, and static and dynamic separation of duty policies.

## 7.1 Overview of RBAC

Unlike in other types of access control mechanisms, in RBAC users do not have discretionary access to enterprise objects. Instead, access permissions are administratively assigned to roles, and users are administrated assigned to roles. Therefore, users are given access permissions based on their assigned roles. This simplifies management of authorization and allows for greater flexibility and simplicity in specifying and enforcing consistent access control policies.

Access is a specific ability or privilege to do something with a resource. For example, for a given object, users may be able to view, modify, or delete it. These are all different types of accesses. An example relating to computers is a file where the different accesses include reading, modifying, appending, and deleting. Access control is the means by which the access is specified; in other words, the ability to access a certain resource is specifically enabled or restricted through some mechanism.

In role-based access control, access decisions are made by determining the role of the user. Users can assume any subset of its assigned roles. These roles are usually administratively assigned and are determined by a careful and thorough analysis of how an organization operates and the privileges different roles require to perform their duties and tasks.

Within the RBAC framework, a user is a person, a role is a collection of job functions, and an operation represents a particular mode of access to a set of one or more protected RBAC objects. The following figure shows the relationships between users, roles, and operations. The use of double arrows indicates a many-to-many relationship. For example, a role can have one or more operations, and likewise, an operation can be associated with any number of roles.

## 7.2 Role Hierarchy

Because roles can have overlapping responsibilities and privileges (users who are in different roles may need to perform similar operations), repeatedly specifying these general or common operations for each role that gets created is cumbersome, inefficient, and error-prone. Instead, RBAC includes the idea of role hierarchy to simplify this administrative task. A role hierarchy defines roles that have unique attributes and that may "contain" other roles, that is, one role may implicitly include the operations, constraints, and objects that are associated with another role. The advantage of role

hierarchy is that it allows for a natural way of organizing roles to reflect authority, responsibility, and competency. One example of role hierarchy in the healthcare field is the role 'specialist' containing the roles of 'doctor' and 'intern'. This means that the role of a specialist implicitly includes the operations, constraints, and objects of the roles of doctor and intern without having to explicitly specify these attributes.

### 7.3 Role Authorization

Role authorization is the association of a user to a role. However, role authorization is subject to three conditions:

1. The user should be given no more privilege than is necessary to perform the job.
2. The role in which the user is gaining membership is not mutually exclusive with another role for which the user already possesses membership.
3. The numerical limitation that exists for role membership cannot be exceeded.

The first property relates to the principle of least privilege, which requires that a user is given only the set of privileges that are necessary to complete the user's job functions, and nothing more. The successful application of this principle requires identifying the user's job functions, determining the minimum set of privileges that are necessary to perform that function, and lastly, restricting the user to the domain with those privileges and nothing more. In other type of access control mechanism other than RBAC, this would be often difficult or costly to achieve.

The second property preserves a policy of static separation of duty or conflict of interest. Static separation of duty means that when a user is a member of one role, the user cannot be a member of another role. This avoids conflict of interest. For example, a user who is authorized to assume the role of a teller should not be allowed to also assume the role of an auditor, and vice versa. Therefore, the roles of teller and auditor are mutually exclusive.

The third property preserves the cardinality property, which restricts the number of users who are authorized to assume a given role. In other words, some roles can only be occupied by a maximum number of users at a given time. Therefore, a user can be a member of a role only if the total number of members of that role does not exceed the maximum allowed.

## 7.4 Role Activation

A user assumes a permitted role by establishing a session during which the user is associated with a subset of roles for which the user has membership. In order to perform an operation, the user must have authorization to assume the role that is associated with the operation. However, this is not the only condition before an operation can be executed. A role can be activated only if:

1. The user is authorized for the role being proposed for activation.
2. The activation of the proposed role is not mutually exclusive with any other active role(s) of the user.
3. The proposed operation is authorized for the role that is being proposed for activation.
4. The operation being proposed is consistent within a mandatory sequence of operations.

Before any operation can be executed, the role for which the user is authorized needs to be activated. However, this role may not be permitted with other activated roles. This is known as dynamic separation of duty. Static separation of duty addresses conflicts of interest issues during the time a user is authorized for a role. However, there are times when it is permissible for a user to be a member of two roles which do not pose a conflict of interest when acted independently, but only raises problems when acted in simultaneously. Dynamic separation of duty allows for greater flexibility in assigning roles, but minimizes conflicts of interest problems by placing constraints on the simultaneous activation of roles.

## 7.5 System Design

- **Operation** – This class represents a single operation or access privilege for RBAC.
  - name – the name of the operation
- **Role** – This class represents a single role in RBAC.
  - name – the name of the role
  - time – the expiration time for this role

- operations – a list of `Operation` objects that have been authorized for this role
- **User** – This class represents a single user in RBAC.
  - name – the name of the user
  - rolesList – a list of role information

The list of roles has been authorized for the user. For each role, there is a Boolean flag to indicate whether the role has been activated, and the remaining time for the role.

**RoleManagementDialog** – This class is the main RBAC administration dialog for role management. The administrator uses this interface to modify any particular role, for example, authorizing or de-authorizing operations for this role.

**UserManagementDialog** – This class is the main RBAC administration dialog for user management. The administrator uses this interface to modify any particular user, for example, authorizing or de-authorizing roles for this user. Also, for each role that is authorized to the user, an expiration time can be specified. If the user does not manually renew the role before the expiration time, then the RBAC server will automatically deassign the role from the user.

**RbacDialog** – This class is the main RBAC administration dialog for the entire RBAC administration. The following figure shows the dialog. Administrators use this interface to access all of the other administrative operations, such as adding/deleting operations (access privileges), adding/deleting roles, modifying roles via the `RoleManagementDialog` object, adding/deleting users, and modifying users via the `UserManagementDialog` object.

**Rbac** – This class is the main RBAC object that provides the major interface into the RBAC operations, such as “activateRole” and “addUser”. The user only accesses this object through the `RbacDialog` and other RBAC GUI dialogs.

**boolean verifyRBACaccess(String user, String operation)** – This is the only function that should be used by task providers or services. This function verifies the access privilege for a specific user. The task provider calls this function with the name of the user in the first parameter, and the name of the operation in the second parameter. The RBAC server then verifies that whether any of the activate roles of the user has permission to perform this operation. If so, the RBAC server returns a boolean value of true; otherwise, it returns a boolean value of false.

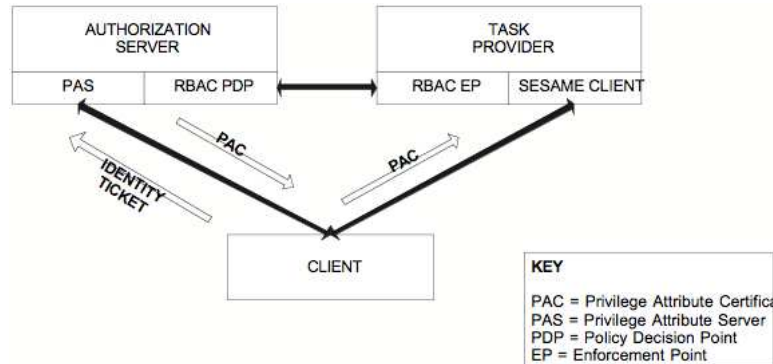


Figure 7.1: Distributed RBAC

## 7.6 Centralized versus Distributed RBAC

In our original design, we expected to have a distributed RBAC system. In such a system, we would then be required to take full advantage of SESAME, specifically, its support for RBAC. SESAME provides support through the Privilege Attribute Certificate (PAC), which identifies the activated roles for a specific user. The PAC is the token by which the user identifies its activated roles to any service provider or task provider which performs the task or operation on behalf of the user.

The figure 7.1 shows the communication between the various entities in setting up and using a PAC. After successful authentication, the client has an identity ticket, which proves the client is who he claims to be. Now, let's say a client wants to use a particular service. First the client sends the authorization server (RBAC server) its identity ticket and the role it wants to assume. The authorization server validates the identity ticket, and determines whether or not the client is authorized to assume the requested role, and whether the assuming of this role creates any potential conflicts with the roles the client has currently assumed. If the client is authorized to assume this role, and there are no conflicts, the RBAC server creates a PAC which specifies the client's activated roles and the expiration times. The PAC is then sent to the client. Now, let's say the client wants a task provider to perform a task. The client provides the task provider with the PAC. The task provider validates the PAC, and performs the task only if the operation is authorized for at least one of the client's assumed roles and if the roles have not yet expired.

However, in our system, we used a centralized RBAC than a distributed

RBAC because there was only one entity, the RBAC server that had the sole responsibility of making the policy decision. Every task provider, upon a request by a client, contacts the RBAC server with the client's name, and the operation the client wants to perform. The RBAC server would then make the policy decision and tell the task provider whether or not the client has the privilege to perform the requested operation. This centralized RBAC server rendered the PAC useless and made our design simpler. However, in the future, as the network may grow larger with independent policy decision administrators, a distributed RBAC architecture will be required.

## 7.7 Secure Communication

We developed a protocol for secure communication between RBAC server and client. The detail of this protocol is as follows:

- Security Context with RBAC server creation - The client sends out a request to RBAC server initiating to establish a "security context". This request includes a signature signed by the service, a symmetric key encrypted by the RBAC Server's public key, a nonce, and a timestamp. RBAC server will send back a reply consisting the decision signed by his private key. If the decision is "YES", the security context is established. After creating the security context, the client and RBAC server shared a secret key, which will be used for later communication.
- Role selection - The client sends out a request for applicable roles. After he gets the reply from RBAC server, he will pick one role as the active role. And he will inform RBAC server his choice so that RBAC server can update its database accordingly. The communications are all encrypted by the secret key.
- Access control - The client ask RBAC server the access control decision. The RBAC server will make the decision based on the user's active role and sends the decision back to the client. The communications are all encrypted by the secret key.



# Bibliography

- [1] D. Agrawal, B. Archambeault, J. Rao, and P. Rohtagi. The em-side channel(s). Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002.
- [2] Sanjeev Arora and Subhash Khot. Fitting algebraic curves to noisy data. ACM Symposium on Theory of Computing, STOC 2002.
- [3] Richard Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.
- [4] Richard Blahut. *Modem Theory: An Introduction to Telecommunications*. Cambridge University Press, preprint.
- [5] Daniel Bleichenbacher and Phong Q. Nguyen. Noisy polynomial interpolation and noisy chinese remaindering. Advances in Cryptology, EUROCRYPT 2000.
- [6] G. Davida, Y. Frankel, and B. Matt. On enabling secure applications through off-line biometric identification. IEEE Symposium on Privacy and Security, 1998.
- [7] V. Guruswami and M. Sudan. Improved decoding of reed-solomon and algebraic-geometric codes. Symposium on Foundations of Computer Science, FOCS 1998.
- [8] D. Hankerson, J. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000.
- [9] F. B. Hildebrand. *Introduction to Numerical Analysis*. McGraw-Hill, 1956.
- [10] H. Jaeger and S. Nagel. Physics of granular states. *Science*, 255(1524), 1992.

- [11] Ari Juels and Madhu Sudan. A fuzzy vault scheme. ACM Conference on Computer and Communications Security, CCS 2002.
- [12] Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. ACM Conference on Computer and Communications Security, CCS 1999.
- [13] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. Advances in Cryptology, CRYPTO 1996.
- [14] Paul Kocher, Joshu Jaffe, and Benjamin Jun. Differential power analysis. Advances in Cryptology, CRYPTO 1999.
- [15] John Kohl and B. Clifford Neuman. The kerberos network authentication service (version 5). Internet Request for Comments RFC-1510, September 1993.
- [16] M. Kuhn and R. Anderson. Tamper resistance: A cautionary note. Workshop on Electronic Commerce, USENIX 1996.
- [17] O. Kummerling and M. Kuhn. Design principles for tamper-resistant smartcard processors. Workshop on Smartcard Technology, USENIX 1999.
- [18] M. Looi, P. Ashley, L. Tang Seet, R. Au, and M. Vandenwauver. Enhancing sesamev4 with smart cards. International Conference on Smartcard Research and Applications, CARDIS 1998.
- [19] J. Lopez and R. Dahab. Overview of elliptic curve cryptography. Technical Report IC-00-10, State University of Campinas, May 2000.
- [20] J. L. Massey. Shift register synthesis and bch decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969.
- [21] F. Monrose, M. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. ACM Conference on Computer and Communications Security, CCS 1999.
- [22] F. Morain and J. Olivos. Speeding up the computation on an elliptic curve using addition-subtraction chains. *Information Theory Applications*, 24:531–534, 1990.
- [23] Randall K. Nichols, editor. *ICSA Guide to Cryptography*, chapter Biometric Encryption. McGraw-Hill, 1999.

- [24] National Institute of Science and Technology. Recommended elliptic curves for federal government use. July 1999.
- [25] J. Osterberg, T. Parthasarathy, T. Raghavan, and S. Sclove. Development of a mathematical formula for the calculation of fingerprint probabilities based on individual characteristics. *Journal of the American Statistical Association*, 72:772–778, 1977.
- [26] S. Pankanti, S. Prabhakar, and A. Jain. On the individuality of fingerprints. *IEEE Transactions on PAMI*, 24:1010–1025, 2002.
- [27] Henna Pietilainen. Elliptic curve cryptography on smartcards. Master’s thesis, Helsinki University of Technology, October 2000.
- [28] S. Sclove. The occurrence of fingerprint characteristics as a two-dimensional process. *Journal of the American Statistical Association*, 74:588–595, 1979.
- [29] H. Steinhaus. *Mathematical Snapshots*. Dover, 3 edition, 1992.
- [30] M. Vandenwauver, R. Govaerts, and J. Vandewalle. Overview of authentication protocols: Kerberos and sesame. pages 108–113. IEEE Carnahan Conference on Security Technology 1997.
- [31] Verifinger. Neurotechnologija ltd. <http://www.neurotechnologija.com>.
- [32] American National Standard X9.62-1998. Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm, working draft. September 1998.
- [33] Tatu Ylonen. Ssh secure login connections over the internet. pages 37–42. Security Symposium, USENIX 1996.