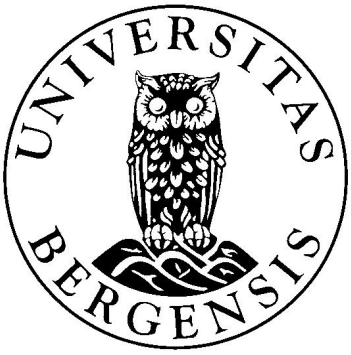
Collusion-Secure Fingerprinting

A Simulation of the Boneh and Shaw Scheme

Coding Theory and Cryptography

Thesis for the degree of
Master of Science in Informatics

Tor Røneid
14 June 2005



Department of Informatics
Universitas Bergensis

Abstract

Fingerprinting is one of many different technics for protecting digital data against illegal copying. To achieve this we have to mark every copy uniquely. We can then trace an illegal copy back to the buyer. It is very important that the mark is impossible to remove or change for non-colluding buyers.

If many buyers are collaborating they can compare their copies. They will then discover positions where the copies differ. This is obviously part of the fingerprint. And when they can identify the fingerprint, they can also alter it. But this can be prevented by constructing codes that are secure against such collaborations. We call these codes collusion-secure codes.

This paper will give an introduction to the field of fingerprinting, in particular collusion-secure fingerprinting. In addition, an overview of some important collusion-secure fingerprinting schemes will be given. One of these schemes will then be implemented and simulations on it will be executed to achieve empirical results that can be compared to the theoretical bounds.

Contents

Preface

This is my thesis for the degree of Master of Science in Informatics. I got my assignment in September 2003, so now it is time to hand it in. I picked this assignment because the concept of protecting copyrighted material is fascinating. My knowledge of the fingerprinting area was very limited, so I thought it could be interesting to familiarize myself with it.

I used the first year to get an overview of the problem area. I had a number of courses in addition to the work on this thesis, so I mainly concentrated on them. The third semester I started the programming, and in retrospect I see that this should have began earlier, because after the programming started I understood the problem area much better. Simulations and most of the thesis writing was done the last semester. The goal of this paper changed during my work. Initially the goal was to implement a couple of fingerprinting schemes, do simulations on them, and then compare the results achieved. But this was, as I discovered, too time consuming. Fortunately interesting results were achieved by the implementation of just one scheme.

So have I learned anything from the work on this paper? Absolutely! I have learned everything from the programming language C to reading and understanding theoretical papers and the problem area. Because of my very limited mathematical background, reading some of the theoretical papers was a challenge. The programming was relatively straightforward, even if a new language had to be learned. Here too, the challenge was to understand the fingerprinting scheme good enough to implement it. For the simulation part of this thesis a better background in statistics would have been an advantage, but I think I managed. Another important aspect with this work is that I acquired new experience in working independently on a large project.

Many people have been there to help and support me, so I would like to take this opportunity to sincerely thank some of them: First of all I have to thank my supervisor Hans Georg Schaathun for helping me through the process of writing this paper. He has always been there to answer questions, and his effort to improve the paper have been priceless. I must also thank my parents for backing me up through these years as a student. Finally I have to thank the students at Molde University College and at University of Bergen for making all these years as a student a memory I will cherish.

Tor Røneid
Bergen, July 27, 2005

Part I

The Problem Area

Chapter 1

Introduction

In this chapter the reader will be introduced to the concept of digital fingerprinting. We will take a look at the history, the related areas, and some of the existing fingerprinting technics.

1.1 The motivation

Illegal copying is a major problem in many areas. For digital material this is especially true, because copying such material is easy and no information is lost in the process. In addition, the growth of the Internet makes it possible to distribute the material in a much larger scale than before. And because of both technical and legal issues it is often difficult to find and prosecute the pirates.

To protect digital copies is a complicated task. In theory it is always possible to crack the copy protection. Methods like cryptography does not resolve this problem, because the information must be decrypted at one point to be able to use it. Hence the goal of digital fingerprinting is to discourage people from illegally redistributing their legally purchased copy, by increasing the risk of being caught. The recent failure of DVD and CD copy prevention systems is another argument supporting the idea of detection techniques like fingerprinting.

1.2 The origin

Fingerprinting is simply a method for a vendor to prevent illegal copying. It is an old tool which link a copy of an item to its buyer. This link allows the vendor, if an illegal copy is found, to trace the buyer responsible for its creation and redistribution. The first example we know is the use of logarithm tables. Here the vendor introduce tiny errors in insignificant digits of log(x), for a random x. We can use the logarithm table below to outline an example.

Example 1 *The goal is to mark and sell a copy of the logarithm table to a*

x	log(x)
1	$log(1) = 0.00000000000$
2	$log(2) = 0.30102999566$
3	$log(3) = 0.47712125471$

Table 1.1: The original logarithm table

buyer. The steps taken to assure that every buyer gets a modified, unique version of the table are as follows:

1. *Randomly choose an x. Here we choose* $x = 3$.
2. *We change one of the insignificant digits for* $x = 3$. *Here we change the 12th digit(1) to the value 2. This change must be unique for this buyer, because otherwise we will create duplicate tables, and we achieve nothing.*
3. *We now have a modified logarithm table, which we sell to the buyer. The link between the modified table and the buyer is stored.*

If the vendor finds an illegal copy, the stored link between the buyer and the table will be used to identify the guilty buyer.

Digital fingerprinting was introduced for the first time by Wagner [Wag83] in 1983. In digital fingerprinting the vendor embeds a secret unique mark in each copy of the digital object. This mark, the fingerprint, makes it possible to trace the guilty buyers, which we call the pirates.

If we take a second look at the logarithm table, we can see that if two or more pirates cooperate they can find the differences in their tables. The same thing is possible for digital fingerprinting. The pirates can tamper with the part of the fingerprint which they can detect. These detectable parts can then be changed, hence the pirate collusion now owns a modified copy of the item, which we call an illegal copy with a hybrid fingerprint. To overcome the problem of pirate cooperation, we use a form of fingerprinting called collusion-secure fingerprinting. This enables the vendor to trace the pirates, even if they collude. The first paper to address the problem of collusions was written by Blakley, Meadows and Purdy[BMP85]. Here a scheme is given that specifies how large the pirate collusion must be in order to erase a fingerprint. The most well-known collusion-secure scheme is due to Boneh and Shaw[BS95],[BS98], and it is this scheme that will be examined in more detail in later chapters.

1.3 The game

The following subsections will further explain what digital fingerprinting is all about. First we will differentiate between the related areas of *steganography*, *fingerprinting*, *watermarking* and *traitor tracing*.

1.3.1 Steganography vs. Watermarking vs. Traitor Tracing vs. Fingerprinting

In *steganography* a secret message is embedded in a cover message so that it is hidden and hence cannot be found. The goal of *steganography* is therefore to hide the existence of a message from everyone but the intended receiver. For a closer look at this area, read the overview made by Anderson and Petitcolas [AP98].

While *steganography* hides a message from being read, watermarks are supposed to be detected by client machines to signal the fact that the objects are protected, and some special license is needed in order to run them. The validation of the user is the primary goal. *Watermarking* uses the concept behind *steganography* to hide the watermark, but even if the watermark is detected and the algorithmic principle of the method is known, a *watermarking* scheme should be robust against attackers. All copies of an object are identically watermarked. The paper, [HK99], by Hartung and Kutter, is recommended for a presentation of the *watermarking* field.

In *traitor tracing* a distributor broadcast encrypted data that should be available only to certain users. Each user has a unique decryption key that will decrypt the encrypted data. Some users can collude to create a new key, which also will decrypt the data. In a *traitor tracing* scheme it should be possible, with a low probability of error, to trace at least one of the creators of the new key, if the number of colluding users is less than some given threshold. For *traitor tracing* we need to access the pirate machine to trace leakage based on the secret key found on the machine. *Traitor tracing* was introduced by Chor, Fiat and Naor in [CFN94].

Fingerprinting contains elements from all the above areas. Information is hidden in some other information. It must be robust against attacks, and if a collusion of buyers create a new object (Key in *traitor tracing*) it should be possible to find some of the colluding buyers. The goal of *fingerprinting* is copyright protection, hence every copy must be individually fingerprinted. The distributor only needs to capture a copy from one of the colluding buyers to trace some of them.

1.3.2 The five parts of fingerprinting

We can divide the fingerprinting game in five main parts:

- **Codes**. The fingerprint is a codeword. One unique codeword must exist for each buyer. There exists many methods for creating such codewords.
- **Embedding**. The fingerprint has to be inserted into the object. This problem is essentially the same as the problem of watermarking.

- **Mapping**. The vendor must of course know which user to blame if an illegal copy is found. So it is necessary to map the buyer's identity to the fingerprint.
- **Pirate strategy**. The pirate collusion choose a strategy for garbling the fingerprint. Then they generate and sell copies with this false fingerprint.
- **Tracing**. If an illegal copy is found, the vendor gives the copy as input to a tracing algorithm. Hopefully the algorithm outputs the guilty users.

More details on these topics will follow in later chapters.

1.3.3 Symmetric vs. Asymmetric vs. Anonymous fingerprinting

For *symmetric* fingerprinting both the distributor and the buyer know how the fingerprinted copy looks like. For this scheme to be trustworthy the distributor must be honest, but this is not always true. A dishonest worker can distribute a copy with the same fingerprint to other buyers. If an illegal copy is found, the buyer and the distributor can blame each other. Hence the distributor can never prove to a third party that the buyer distributed the data illegally.

For *asymmetric* fingerprinting the distributor and the buyer work together to create the fingerprinted copy. The buyer sends the distributor a commitment to a chosen secret. Then they carry out a protocol, which ends with the buyer acquiring the object, fingerprinted with the chosen secret. The buyer is the only one who knows this secret. With this solution the dishonest worker can no longer distribute illegal copies, while the distributor can, when able to present a sufficiently large fraction of the secret, identify a buyer when an illegal copy is found. The distributor can obtain a proof of treachery that convinces a third party, hence *asymmetric* fingerprinting is a better choice than *symmetric*. An overview of *asymmetric* fingerprinting is given by Pfitzmann and Schunter in [PS96].

Neither of these schemes do preserve privacy, therefore *anonymous* fingerprinting has been suggested. In *anonymous* fingerprinting the distributor does not know the buyers identities. The identity can only be uncovered when an copy is used illegally and found. To accomplish this the distributor uses a registration service that stores the connection between the fingerprints and the buyers. This scheme is described by Pfitzmann and Waidner[PW97] and Pfitzmann and Sadeghi[PS99].

1.4 Outline of the thesis

Chapter 6 is the main chapter of this thesis. This chapter presents the results achieved through the simulations on the Boneh and Shaw fingerprinting scheme. The most important result outlined in this chapter is the fact that

the r parameter bound given by [BS95],[BS98] is very unprecise (for the chosen pirate strategy).

The paper is organized as follows:

Part 1: The Problem Area

- Chapter 2 describes the terminology used in the fingerprinting literature. The chapter will be divided in sections according to the different parts in a fingerprinting scheme.
- Chapter 3 will introduce the Boneh and Shaw fingerprinting scheme to the reader. The relevant parts of the scheme will be explained, and an example will be given.
- Chapter 4 mentions some other important fingerprinting schemes.

Part 2: The Simulation

- Chapter 5 gives an introduction to the simulations done. The background and some facts needed to understand the simulations are given. Random number generators, and why the simulation depends on a good one, is also mentioned here.
- Chapter 6 is the main chapter which will give all simulation results achieved. These results will then be discussed, and hopefully this will give new inside to the Boneh and Shaw scheme.
- Chapter 7 discusses future work. Aspects that are important, but have received little or no attention in this paper due to time limitations, will be mentioned here.

Part 3: The Implementation

- Chapter 8 gives an introduction to the implementation of the Boneh and Shaw scheme. Here aspects like design and computer usage will be outlined.
- Chapter 9 describes all the program modules. This explanation of the modules serves as the technical documentation of the program.
- Chapter 10 gives a short user manual. This manual describes how the user can interact with the program. Common error messages are also mentioned.

Part 4: The Appendices

- Appendix A lists the program source code.

Chapter 2

Terminology

The goal of this chapter is to familiarize the reader with the terminology used in later chapters. First we will take a look at the codes used in fingerprinting, then we mention how a collusion of buyers can create a hybrid fingerprint. The concept of t-frameproof codes will then be defined, and the last section will give an introduction to tracing algorithms. Most of the notations and terminology used throughout this paper are from coding theory.

2.1 Code construction

2.1.1 The codes

The distributor maintains a set of unique fingerprints that will be used to mark copies with. In general we call this an $(n, M)_q$-code C. Here n is the length of the fingerprint, or in other words, the fingerprint contains n symbols. Such a symbol will be called a mark, and this mark must be within the alphabet q, which means that all the marks in the code must be in the interval $\{0, 1, \ldots, q-1\}$. M is the number of users supported by the code, hence the distributor can sell copies to at most M buyers.

Example 2 *Let C be a code with 4 codewords, length 5 and an alphabet of 2. C is then a $(5, 4)_2$-code.*

$$C = \begin{cases} 11000 \\ 01100 \\ 00110 \\ 00011 \end{cases}$$

As we see from the example, the code C consists of 4 codewords. The codewords of C together form a matrix which is called the code book C. From this code book we will assign row i to user i, for $1 \leq i \leq M$.

Definition 1 *$C = \{w^1, w^2, \ldots, w^M\} \subseteq q^n$ is called an (n, M) code where each codeword w^i is assigned to one user u_i, and the set of all words in C is called the code book.*

The Hamming distance between two words x and y is denoted $d(x, y)$, and the minimum distance of a code C is a denoted $d(C)$ or just d.

Example 3 *If we look at 2 codewords from the previous example, $x = 11000$ and $y = 01100$, we see that these two codewords differ in 2 positions, hence the Hamming distance is $d(x, y) = 2$. The minimum distance of the code in the previous example is $d = 2$, because 2 is the smallest Hamming distance between any two codewords in the code.*

In Section 2.3 we will examine a particular code construction called t-frameproof code. This code is not defined in this section because it is necessary to explain the *marking assumption* and the concept behind concatenated codes first.

The notation C will indicate the use of a general code. But we also have codes that are parts of other codes. Concatenation of codes will be discussed in the next subsection.

2.1.2 Concatenation of codes

Concatenation is a well-known technique from coding theory, and it has proven extremely useful for the fingerprinting area.

Definition 2 *Let C_1 be a (n_1, q) inner code, and let C_2 be an $(n_2, M)_q$ outer code. We then map the symbols of C_2 on a word from C_1. The result is the concatenated code $C'(n_1 n_2, M)_q$.*

Often the symbol Q is used to represent the size of the inner code alphabet. But through this paper we will only look at binary codes, hence we assume an inner code alphabet size of 2, with the symbols $\{0, 1\}$.

The following definition will further explain the concatenation of the codes:

Definition 3 *Let $C_1 = \{w^{(1)}, \ldots, w^{(q)}\}$ be an (n_1, q) inner code, and $v = v_1, v_2, \ldots, v_{n_2}$ an outer codeword. Then we have the concatenated word $W = w^{(v_1)}||w^{(v_2)}||\ldots||w^{(v_{n_2})}$.*

We see that the number of codewords in the inner code equals the outer code alphabet. This means that it is a one-to-one relationship between the outer code symbols and the inner codewords. The concatenation process will look at the outer code symbols, and then write the composition of the inner codewords the symbols point to. For example will the symbol 2 in the outer code point to codeword number 2 of the inner code. The resulting code will then have length $n_1 n_2$ and support for M buyers. One inner code, which is part of the concatenated code, is called a block or a component.

Example 4 *This example will illustrate the concatenation process.*
$C_2 = (4, 4)_4$ outer code

$$C_2 = \begin{cases} 1234 \\ 2341 \\ 3412 \\ 4123 \end{cases}$$

$C_1 = (4, 4)$ *inner code*

$$C_1 = \begin{cases} 1000 \\ 0100 \\ 0010 \\ 0001 \end{cases}$$

$C' = (n_1 n_2, M)_q = (16, 4)_4$ *concatenated code*

$$C' = \begin{cases} 1000||0100||0010||0001 \\ 0100||0010||0001||1000 \\ 0010||0001||1000||0100 \\ 0001||1000||0100||0010 \end{cases}$$

2.1.3 Code summary

Code	Notation	Explanation
General	n	Length of fingerprint
code	M	Number of codewords
C	q	The alphabet size
	$(n, M)_q$	The code notation
Inner	n_1	Length of fingerprint
code	q	Number of codewords
C_1	Q	The alphabet size
	$(n_1, q)_Q$	The code notation
Outer	n_2	Length of fingerprint
code	M	Number of codewords
C_2	q	The alphabet size
	$(n_2, M)_q$	The code notation
Concat.	$n_2 \cdot n_1$	Length of fingerprint
code	M	Number of codewords
C'	q	The alphabet size
	$(n_2 \cdot n_1, M)_q$	The code notation

Table 2.1: Code notation overview

2.2 Creation of the hybrid fingerprint

This section will examine the information available to a pirate collusion, and the strategies they employ when creating hybrid fingerprints. A hybrid fingerprint is a false fingerprint constructed by one or more buyers. A buyer which

construct a hybrid fingerprint will be called a pirate, and a collusion of pirates will be called a pirate collusion P. The number of pirates in P will be denoted t.

2.2.1 The marking assumption

As mentioned in Chapter 1, a distributor must embed the fingerprint into the digital copy. This embedding must ensure that:

- A non-colluding pirate cannot detect the marks.
- The pirates cannot change the state of an undetected mark without rendering the object useless. The fingerprint must survive any changes of file format and lossy compression.

The marking assumption defines what a pirate collusion is allowed to do. The possible codewords that a pirate collusion can generate are given by the following definition:

Definition 4 *Let $P \subseteq C$ be the set of fingerprints held by a coalition of pirates. The pirates can produce a copy with a false fingerprint x for any $x \in F_C(P)$, where*

$$F_C(P) = \{(c_1, \ldots, c_n) : \forall i, \exists (x_1, \ldots, x_n) \in P, x_i = c_i\}.$$

We call $F_C(P)$ the feasible set of P with respect to C.

Two or more pirates can only detect marks in which their copies differ. Hence the set of possible codewords the pirates can generate is the feasible set. To illustrate the idea of feasible sets let's consider an example:

Example 5 *Let A and B be two pirates each assigned a codeword of* 10110 *and* 10011*. The feasible set contains the positions where the codewords differ. In this example position 3 and 5 differ, hence the feasible set is $F(AB) = 1{\cdot}0{\cdot}?{\cdot}1{\cdot}?$. Then A and B use this information to construct a hybrid fingerprint, for example* 10010*.*

2.2.2 Pirate strategies

Colluding pirates can detect those marks in which their copies differ. This makes it possible to change the detected marks in order to create a hybrid fingerprint. The strategies considered by the pirates for setting the detected marks are called pirate strategies. But the pirate strategies are restricted, because they can only change what they see. The following information is assumed to be unknown by the pirates:

- The undetected marks in the object.
- Which marks in the object that correspond to which coordinates in the code (a random permutation hides this fact).

- The alternative in each mark in the object.
- Which codewords that are assigned to which users.

The pirates use a strategy that they think will fit their purpose. Examples can be to choose a strategy that will blame other users, or a strategy that minimizes the probability of getting caught. We distinguish between choosing the strategy before or after the pirate collusion locates the detectable marks. If they can choose the strategy after the detectable marks are known, they can pick an optimal strategy, hence the probability of being caught will decrease. *Adverse selection* is when an pirate collusion choose to create a hybrid fingerprint only if they observe that the error probability is in their favor.

Some simple strategies will now be mentioned [LL00b]. These strategies are general, they are not restricted to particular fingerprinting schemes.

Deterministic Strategies: For these strategies no random choices are taken by the pirates. Some deterministic strategies will now be given. These strategies require that the number of pirates must be odd and greater than 2.

- *Majority choice*: For all the detectable marks, the pirates choose the alternative that occurs the greatest number of times in the pirates' object. Hence the illegal fingerprint will, on average, have a small Hamming distance to the pirates' fingerprints.
- *Minority choice*: This is the opposite of *majority choice*. Here the illegal fingerprint on average will have a large Hamming distance to the pirates' fingerprints. This sounds better, but in fact it is just as revealing as an small Hamming distance.
- *Binary Addition*: For all the detectable marks, the pirates choose the alternative that occurs an odd number of times in the pirates' objects. It is also possible to invert the result of binary addition.

Random Strategy: Here the pirates use random choices to set the detected marks. An example of such a strategy is:

- *Random Alternative Choice*: For each of the detectable marks the pirates choose randomly among the different alternatives they can see, with equal probability.

Not much literature exists on pirate strategies, but for a more detailed discussion on the topic, take a look at the paper [LL00a].

2.3 t-Frameproof codes

We would like to construct codes where no collusion can frame a user outside the collusion. To achieve this goal we often restrict the collusion size to t pirates. The codes constructed under this restriction are known as t-frameproof codes.

Definition 5 *A code C is t-frameproof if every set $P \subset C$, of size at most t, satisfies $F(P) \cap C = P$.*

What this means is essentially that a collusion P, of size at most t, can only produce codewords that are codewords of P. This fact also makes a t-frameproof scheme combinatorially secure, because the hybrid fingerprint is a pirate fingerprint (tracing is trivial).

2.3.1 Construction of t-Frameproof codes

So, how can we construct such t-frameproof codes. Over the binary alphabet $\{0, 1\}$ we can use the following method: Let $C(M, M)$ denote all words of length M such that they each contain one 1.

Example 6 *: $C(3, 3) = \{100, 010, 001\}$, $C(4, 4) = \{1000, 0100, 0010, 0001\}$, and so on.*

Claim 1 *$C(M)$ is an M-frameproof (M, M)-code.*

Since there are M possible collusions of size $M - 1$, the code must have at least length M. Otherwise an collusion will be able to detect at least one bit position, and frame a user outside the collusion. The length of each codeword in $C(M)$ is linearly proportional to the number of users, hence the length grows as the number of users grow. This is not practical. We risk getting a fingerprint that is larger than the actual piece of data.

We construct shorter codes by concatenating the M-frameproof code discussed above with an Error Correcting code. The M-frameproof code is now used as the inner code C_1, and the error correcting code is the outer code, C_2. We give the definition of an Error correcting code:

Definition 6 *A set C_2 of M words of length n_2 over an alphabet of q letters is said to be an $(n_2, M, d)_q$-Error Correcting Code or in short, an $(n_2, M, d)_q$-ECC, if the Hamming distance between every pair of words in C_2 is at last d.*

The concatenated code C' is then the composition of C_1 which is an (n_1, q)-code and C_2 which is an $(n_2, M, d)_q$-ECC.

Lemma 1 *Let C_1 be a t-frameproof (n_1, q)-code and C_2 be $(n_2, M, d)_q$-ECC. Let C' be the composition of C_1 and C_2. If $d > n_2(1 - 1/t)$ then C' is t-frameproof.*

The next two lemmas will determine the values of n_2 and n_1 we must use to construct t-frameproof codes.

Lemma 2 *For any positive integer q, M let $n_2 = 8q \log(M)$. Then there exists an $(n_2, M, d)_{2q}$-ECC such that $d > n_2(1 - 1/q)$.*

Theorem 1 *For any positive integer q, t let $n_1 = 16t^2 \log(q)$. Then there exists an (n_1, q)-code which is t-frameproof.*

2.4 Tracing

A fingerprinting scheme consists of a code, a mapping between the code and the set of users, and a tracing algorithm A. After a distributor has received a hybrid copy, a method is needed in order to trace the pirates that are responsible for creating the copy. A tracing algorithm uses the fact that the pirates only can change the marks they can detect, to trace at least one of the pirates. It takes a hybrid fingerprint x as input and outputs a subset $P \subseteq C$.

The tracing algorithm output is not always trustworthy. We often talk about probabilistic and combinatorially collusion-secure schemes. In a combinatorial scheme the tracing algorithm returns a subset of the guilty collusion with probability 1. But these schemes are not common in the collusion-secure fingerprinting literature due to long codeword requirements. Instead we use probabilistic schemes. Here the tracing algorithm returns the guilty pirate with probability at least $1 - \epsilon$ for some small error rate ϵ.

The output of a tracing algorithm can be divided in three:

- Successful tracing: Tracing is successful if the output is a non-empty subset of the pirate collusion.
- Type I error: An error of Type I occurs if the output of the algorithm is empty.
- Type II error: An error of Type II occurs if the output of the algorithm is one or more innocent buyers (can also include pirates).

The Type II error is clearly more severe.

If we use a concatenated code to mark copies with, the hybrid fingerprint x will be a composition of n_2 inner codes. So when x is given as input to the tracing algorithm, it often decodes each block (inner code) using a decoding algorithm specific for the inner code. This gives a word of symbols from the outer code alphabet. This word will then be decoded with an algorithm designed for the outer code.

One example of an algorithm used for outer code decoding is Closest Neighbor Decoding, or just CND. This algorithm takes a word x as input, and returns a word $c \in C$ such that $d(c, x)$ is minimized. Hence the word in the code C, that has the most positions in common with x, will be returned. The owner of this codeword will be presented as the guilty pirate. This can always be performed in $O(M)$ operations, and for some codes it may be faster.

2.5 Model of the fingerprinting system

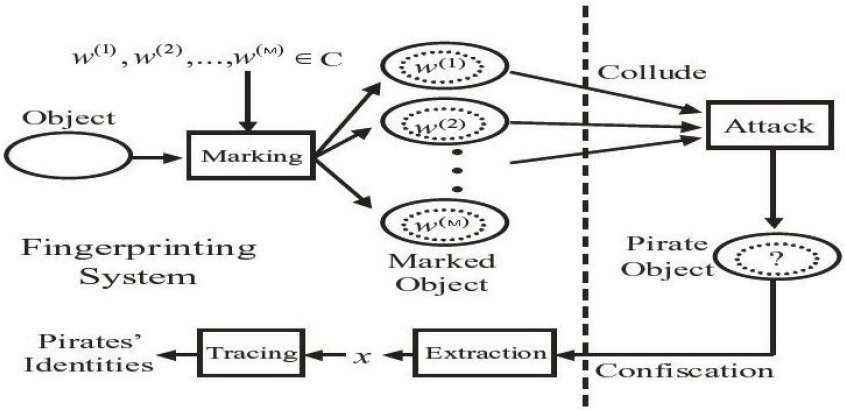


Figure 2.1: Fingerprinting system

Chapter 3

The Boneh and Shaw scheme

This chapter will explain one of the most well-known collusion-secure schemes created, the Boneh and Shaw scheme [BS95],[BS98]. Here the most important aspects will be mentioned, but proves that offer nothing to the process of understanding the scheme are left out.

In [BS98] Boneh and Shaw take a step by step approach in order to explain the scheme. Totally secure and probabilistic t-secure codes are first defined. Then they introduce the replication scheme, and finally the concatenated scheme (BS-CS) is constructed. This gives an intuitive introduction to the reader, so the approach is adopted here.

The last section of this chapter will give an example of BS-CS fingerprinting scheme. This example will deal with code construction, creation, and tracing of the hybrid fingerprint.

3.1 The facts

The BS-CS scheme consists of a binary code and a tracing algorithm that always returns one user as guilty. The code is a concatenation of an inner and an outer code. The scheme is probabilistic. Given the desired error probability (ϵ), the number of pirates the code must handle (t), and the total number of users in the system (M), it is possible to choose the code length so that the actual error probability is less than ϵ.

3.2 t-Secure Codes

A totally t-secure code is the combination of a t-frameproof code and a tracing algorithm. The tracing algorithm is used when the distributor locates a forged copy, and wants to trace the members of the guilty collusion. The construction of this algorithm is described in this section.

Definition 7 *A code C is totally t-secure if there exists a tracing algorithm A*

satisfying the following condition: If a collusion P of at most t users generates a word x then $A(x) \in P$.

If a distributor locates an illegal copy x generated by a collusion, x will be given as input to the tracing algorithm. A totally t-secure scheme is of course combinatorial, so the output will be at least one member of the collusion. It is not very likely that the tracing algorithm will return all the users in the guilty collusion, because some of the members might have been passive in the construction process. They have not contributed to the resulting fingerprint. The following lemma gives an important condition for a code to be totally t-secure.

Lemma 3 *We have a set of disjoint collusions $P_1, \ldots, P_i$, of at most t users each, that do not have any member in common. Denote the feasible set of each of these collusions by $F(P_1), \ldots, F(P_i)$. If C is totally t-secure then $P_1 \cap \ldots \cap P_i = \emptyset \Rightarrow F(P_1) \cap \ldots \cap F(P_i) = \emptyset$.*

This means that when all the collusions don't have any members in common, and the feasible sets of all collusions don't share common codewords, then C is totally t-secure. This prevent the situation where collusions with no common members can construct the same codewords. Then it would be impossible to know if the guilty users were caught.

Unfortunately pirates can use a strategy called the *majority word strategy*. This allows a collusion of 2 or more users to create a codeword outside their feasible set.

Theorem 2 *For $t \geq 2$, $M \geq 3$ and $q = 2$ there exists no totally t-secure (n, M)-codes.*

We prove this theorem for a collusion of 2 pirates. We have 3 users u_1, u_2, u_3 each with one distinct codeword $w^{(1)}, w^{(2)}, w^{(3)}$. We define the majority word $m = MAJ(w^{(1)}, w^{(2)}, w^{(3)})$ like this:

$$m_i = \begin{cases} w_i^{(1)}, & \text{if } w_i^{(1)} = w_i^{(2)} \text{ or } w_i^{(1)} = w_i^{(3)} \\ w_i^{(2)}, & \text{if } w_i^{(2)} = w_i^{(3)} \end{cases}$$

So the majority word m will be feasible for all three collusions, but the intersection of the collusions will be empty. The following example will illustrate why this is the fact.

Example 7 *The $(3,3)$-code $C = \{100, 010, 001\}$ has the majority word $m = 000$, because this word is feasible by all possible collusions $\{u_1, u_2\}$, $\{u_1, u_3\}$, $\{u_2, u_3\}$. If the collusion contains the words $\{100, 010\}$, then we see that the marks in positions 1 and 2 can be detected and changed to 0. Since all the collusions can construct the majority word, the intersection of the feasible sets for all the collusions contain at least one word that is the majority word. And as we see above, the intersection of the collusions is empty. C is therefore not a totally 2-secure code by Lemma 3.*

So how can we solve this *majority word strategy* problem. The key is to use randomness. The distributor uses a random permutation string R on the codewords before they are embedded into the objects. The point is that the random choices will be hidden from the users. This enables the distributor to catch at least one member of the pirate collusion with high probability.

Let P be a collusion of t users that constructs a hybrid fingerprint. Schemes that let us catch at least one of the users in P with probability $1 - \epsilon$ is called t-secure codes with ϵ-error. We see that the use of combinatorial schemes now are abandoned.

Definition 8 *A fingerprinting scheme is t-secure with ϵ-error if there exists a tracing algorithm A satisfying the following condition: If a collusion P of at most t users generates a word x then*

$$Pr[A(x) \in P] > 1 - \epsilon$$

where the probability is taken over the random string R and the random choices made by the collusion.

3.3 The Boneh and Shaw Replication Scheme

This section will discuss the q-secure codes that are the building block (the inner code) for the logarithmic t-secure codes that will be discussed in the next section. This q-secure code is called the Boneh and Shaw Replication Scheme (BS-RS). Why it is called a replication scheme will soon be explained. This q-secure code will help us to construct shorter codes of length $O(log(q))$. To recapitulate, q is the number of codewords in the inner code (and the outer code alphabet, but that is not relevant in this section).

First an q-secure (n_1, q)-code with ϵ error for any $\epsilon > 0$ is constructed. This will make it possible for the distributor to trace an illegal copy back to a member of a collusion with high probability, no matter how large the collusion is. Let c_m represent the column of height q with first m bits set to 1 and the rest to 0. The code $\Gamma(q, r)$ consists of all columns $c_1, \ldots, c_{q-1}$ duplicated r times. The r is called the replication factor, hence we call this scheme the replication scheme. We can look at Γ as a matrix with ones at the main diagonal and above, and zeroes below. Let $C(3, 4)$ be the code:

$$C = \begin{cases} 111 \\ 011 \\ 001 \\ 000 \end{cases}$$

This code consists of 3 columns of height 4. The first column consist of one 1, and we therefore call it c_1. Column two consists of two 1's so we call it c_2, and

so on. Since $r = 3$ we repeat each column c_i 3 times. We get the code:

$$\Gamma = \begin{cases} 111111111 \\ 000111111 \\ 000000111 \\ 000000000 \end{cases}$$

Let $w^{(1)}, \ldots, w^{(q)}$ be the codewords of $\Gamma(q, r)$. The distributor will choose a random permutation π over w^i. Before embedding the marks into the data, a random permutation on the codewords will be executed. The location of the bit embedded by mark i is then hidden from the users, hence the *majority word strategy* is no longer a problem (see example 8). We can for example choose the random permutation $\pi = \{2, 1, 5, 7, 3, 9, 4, 8, 6\}$, and then map the code according to this:

$$\Gamma_p = \begin{cases} 111111111 \\ 001101111 \\ 000101010 \\ 000000000 \end{cases}$$

The following theorem determines the size of r for a code Γ such that given q users the code Γ is q-secure with ϵ error. This means that, with high probability, a collusion of q pirates can not frame a user outside their collusion.

Theorem 3 *For $q \geq 3$ and $\epsilon > 0$ let $r = 2q^2 log(\frac{2q}{\epsilon})$. Then the fingerprinting scheme $\Gamma(q, r)$ is q-secure with ϵ-error.*

The length of this code is $r(q-1) = O(q^3 log(\frac{q}{\epsilon})$.

Before we describe an algorithm for tracing pirates it is necessary to look at an important notation:

1. We define B_m to be all bit positions in which the users see columns of type c_m. The number of elements in B_m is r. In above example $B_1 = \{1, 2, 3\}, B_2 = \{4, 5, 6\}$ and $B_3 = \{7, 8, 9\}$.

2. For $2 \leq s \leq M - 1$ define $R_s = B_{s-1} \cup B_s$.
   In our example $R_2 = \{1, 2, 3, 4, 5, 6\}$ and $R_3 = \{4, 5, 6, 7, 8, 9\}$

3. For a binary string x, let weight(x) = number of 1's in x.

The following example will give some intuition to the terminology just explained:

Example 8 *The hidden permutation π prevents the collusion from knowing which marks represent which bits in the code. For simplicity we here look at the unscrambled code, but we still assume that the only information available*

to the pirate collusion is the detected marks. We have the code:

$$\Gamma = \begin{cases} w_1 = 111111111 \\ w_2 = 000111111 \\ w_3 = 000000111 \\ w_4 = 000000000 \end{cases}$$

If we look at the codewords in Γ*, we see that without* w_2*, all bit positions in* R_2 *(the first* 6 *columns) are equal. The users with codewords* w_1, w_3 *and* w_4 *can not determine whether the bits in* R_2 *comes from* B_1 *or* B_2*. So when they produce a codeword* x*, the number of* 1*'s will be distributed evenly within locations in* R_2*, with high probability. If the* 1*'s in* R_2 *is not evenly distributed, then with high probability user* 2 *is a member of the pirate collusion.*

Given a word x generated by some collusion P, Algorithm 1 outputs a member of P with probability $1 - \epsilon$.

Algorithm 1 *Given unscrambled* $x \in \{0,1\}^{n_1}$ *with unreadable marks set to* 0*, find a subset of the collusion that produced* x*.*

1. *If* $weight(x|_{B_1}) > 0$*, output user* 1 *as guilty*
2. *If* $wight(x|_{B_{q-1}}) < r$*, output user* q *as guilty*
3. *For all* $s = 2$ *to* $q - 1$ *do: Let* $k = weight(x|_{R_s})$*. If*

$$weight(x|_{B_{s-1}}) < \frac{k}{2} - \sqrt{\frac{k}{2} log \frac{2q}{\epsilon}}$$

output user s *as guilty.*

The correctness of this algorithm is given by the next two lemmas:

Lemma 4 *Consider the code* $\Gamma(q, r)$ *and* $r = 2q^2 log(\frac{2q}{\epsilon})$*. Let* L *be the set of users that* Algorithm 1 *outputs as guilty on input* x*. With probability at least* $1 - \epsilon$ *the set* L *is a subset of the collusion* P *that produced* x*.*

Lemma 5 *Consider the code* $\Gamma(q, r)$ *where* $r = 2q^2 log(\frac{2q}{\epsilon})$*. If* L *is the set of users* Algorithm 1 *pronounces as guilty on input* x*, then* L *is not empty.*

3.4 The Boneh and Shaw Concatenated Scheme

We will take a look at the requirements to achieve a t-secure logarithmic length code. To achieve such codes Boneh and Shaw uses the concatenation of an inner code and an outer code:

- As the inner code the Boneh and Shaw Replication Scheme (BS-RS) is used. This is the code described in the previous section as Γ. It is a binary $(r(q-1), q)$-code which is q-secure with ϵ error. The r is the amount of duplication and q is the number of codewords.

- As the outer code Boneh and Shaw use the Random Code (RC) scheme due to Chee [Che96]. We use the notation C_2 for this code, which is an (n_2, M)-code over the alphabet q. Each symbol in each codeword is chosen uniformly at random from the alphabet. The length is given by n_2, while M is the number of users supported by the code.
- We then concatenate the two codes to create a concatenated code. We call this resulting scheme the Boneh and Shaw Concatenated Scheme (BS-CS), and we write $C'(n_1 n_2, M)_q$. The concatenation is achieved as described in Chapter 2. We map each of the outer codeword symbols to a inner codeword. The concatenated code contains M codewords and has length $n_1 \cdot n_2$, where $n_1 = r(q-1)$.

The following theorem defines the code parameter sizes:

Theorem 4 *Given integers M, t and $\epsilon > 0$ we set*

$$q = 2t, n_2 = 2t \log(\frac{2M}{\epsilon}), r = 2q^2 \log(\frac{4qn_2}{\epsilon})$$

then $C'(n_1 n_2, M)_q$ is a t-secure code with ϵ-error. The code contains M words, and has length $n_1 n_2$.

The following algorithm will describe how we can trace a hybrid fingerprint x with the use of Algorithm 1 and closest neighbour decoding

Algorithm 2 *Given an unscrambled $x \in \{0,1\}^{(n_1 n_2)}$ with unreadable marks set to 0, find a subset of the collusion that produced x.*

1. *Apply Algorithm 1 to each of the n_2 components of x. For each component $i = 1, \ldots, n_2$, arbitrarily choose one of the outputs of Algorithm 1. Set y_i to be this chosen output. Note that y_i is a number between 1 and q. Next, form the word $y = y_1, \ldots, y_{n_2}$.*
2. *Find the word $w \in C_2$ which matches y in the most number of positions (ties are broken arbitrarily).*
3. *Let u be the user whose codeword is derived from $w \in C_2$. Output user u as guilty.*

Lemma 6 *Let x be a word which was produced by a collusion P of at most t users. Then with parameters as in theorem 4, Algorithm 2 will output a member of P with probability at least $1 - \epsilon$.*

Recent work has shown that the BS-CS scheme is better than assumed by it's two authors. Chapter 4.1 gives a short summary of the [Sch03a] analysis, and improvements on the outer code and decoding algorithm. The simulation results given in Chapter 6 will show that the BS-CS scheme is much better than initially believed, for an particular pirate strategy.

3.5 An example

This section will provide an example of the BS-CS scheme. Because the smallest codes are too large to include here, this example will not be logarithmic.

First we construct a concatenated code from an inner and an outer code. Then we create a pirate hybrid fingerprint, and at last we find the guilty pirate by tracing the hybrid fingerprint. We use the notation C_1 for the inner code.

3.5.1 Construction of the code

We want to provide for a maximum of 4 users, so we randomly choose the outer code (RC): $C_2 = (n_2, M)_q = (4, 4)_4$:

$$C_2 = \begin{cases} 1432 \\ 3421 \\ 2313 \\ 1223 \end{cases}$$

Since the alphabet of the outer code is 4, the number of codewords in the inner code also must be 4. Hence the inner code (BS-RS) is: $C_1 = (q, r) = (4, 3)$:

$$C_1 = \begin{cases} 111111111 \\ 000111111 \\ 000000111 \\ 000000000 \end{cases}$$

We then concatenate $C_1(4, 3)$ with $C_2(4, 4)_4$ as described in Chapter 2. The resulting code will be the concatenated code (BS-CS): $C'(n_1 n_2, M)_q$.

$$C' = \begin{cases} W_1 = & 111111111\|000000000\|000000111\|000111111 \\ W_2 = & 000000111\|000000000\|000111111\|111111111 \\ W_3 = & 000111111\|000000111\|111111111\|000000111 \\ W_4 = & 111111111\|000111111\|000111111\|000000111 \end{cases}$$

C' have $M = 4$ codewords with length $n_2 r(q-1) = 4 \times 3(4-1) = 36$. Therefore the code parameters of C' is $(36, 4)_4$. We can see that every codeword in C' consists of 4 components, i.e. 4 copies of $C_1(q, r)$.

In addition to keeping the codewords of C_2 hidden from the users, we also keep the permutation π on the codewords of C' secret. We choose a permutation word π randomly and then we execute the permutation on the codewords in C' before they are embedded into the copies.

$$Perm. \begin{cases} \pi = & 327916485\|358791426\|892741563\|867942135 \\ W_1 = & 111111111\|000000000\|110100000\|111110001 \\ W_2 = & 001100010\|000000000\|110110110\|111111111 \\ W_3 = & 001101111\|001110000\|111111111\|101100000 \\ W_4 = & 111111111\|011110101\|110110110\|101100000 \end{cases}$$

3.5.2 Construction of the hybrid fingerprint

We have a collusion P of $t = 2$ users. These two users will we call user A and user B. A has the fingerprint W_1, and B has W_3. User A and B then compare their copies to produce a new illegal copy. Their feasible set is given by $F(AB)$ and the hybrid fingerprint is written HFP. HFP_p is the permutated hybrid fingerprint. Detectable marks will be written as ?.

$$Hybrid \begin{cases} W_1 = & 111111111 \| 000000000 \| 110100000 \| 111110001 \\ W_3 = & 001101111 \| 001110000 \| 111111111 \| 101100000 \\ F(AB) = & ??11?1111 \| 00???0000 \| 11?1????? \| 1?11?000? \\ HFP_p = & 101111111 \| 000100000 \| 111101010 \| 101110000 \end{cases}$$

When the collusion constructs this codeword it must either use 1 or 0 for the marks detected (or it can set the mark to be unreadable). Here it uses 1's for approximately half the marks, and 0's for the rest (this depends on the pirate strategy).

3.5.3 Tracing the pirates

When the distributor receives a copy of the object marked with this hybrid fingerprint, the goal is to find at least one of the members of the guilty pirate collusion P. The first step is to use the permutation word π on the codeword in order to get the non-permutated hybrid fingerprint:

$$perm^{-1} = \begin{cases} HFP_p = & 101111111 \| 000100000 \| 111101010 \| 101110000 \\ \pi = & 327916485 \| 358791426 \| 892741563 \| 867942135 \\ HFP = & 101111111 \| 000000100 \| 110001111 \| 000100111 \end{cases}$$

The second step is to give the fingerprint as input to Algorithm 2. This algorithm breaks the codeword HFP into 4 components, and gives each component as input to Algorithm 1. The ϵ value used in Algorithm 1 is calculated like this:

$$\epsilon_{in} = 2q \cdot 2^{-(r/(2q^2))}$$

The first word given as input to Algorithm 1 is 101111111. First the word is tested against the first condition: $weight(x|_{B_1}) > 0$. B_1 is the 3 first positions in the word x. Is the number of 1's in B_1 higher than 0? The answer is yes (2), hence this condition is met, and Algorithm 1 gives the output 1. The output returned for all the 4 components of the hybrid fingerprint is given below:

1. Input to Algorithm 1: 101111111. Output: $y_1 = 1$ (condition 1 is met).
2. Input to Algorithm 1: 000000100. Output: $y_2 = 4$ (condition 2 is met).
3. Input to Algorithm 1: 110001111. Output: $y_3 = 3$ (condition 3 is met).
4. Input to Algorithm 1: 000100111. Output: $y_4 = 2$ (condition 3 is met).

We now form the word $y = y_1, y_2, y_3, y_4 = 1432$. The next step is to compare this word with the outer codewords in C_2. The user with the outer codeword with the smallest Hamming distance to y, is probably the guilty one. In this example it matches user 1's codeword in 4 positions. User 1 is part of the collusion, hence the distributor has found at least one member of the guilty collusion P.

Chapter 4

Other important schemes

This chapter will first outline some of the improvements published on the BS-CS scheme. Then a quick introduction to the pros and cons of some of the fingerprinting schemes that exist in addition to the BS-CS scheme will be given. Most of the codes from the literature are binary, hence all schemes mentioned here will be binary. For a detailed overview and comparison of most of the following schemes, see [Sch04a].

4.1 Improving the BS-CS scheme

First we will explain how we can improve the BS-CS scheme. The scheme uses an outer code due to [Che96], hence the inner code and the tracing algorithm are the new aspects introduced. The following can be done to improve the BS-CS scheme:

1. Improve error analysis.
2. Change outer code.
3. Change inner code.
4. Change decoding algorithm.

In the following subsections will mention some of the papers that give improvements on the codeword lengths.

4.1.1 Improve error analysis

In [Sch03a] Schaathun gives a new analysis on the error probability of the BS-CS scheme. The bounds given here proves that the BS-CS scheme is better than assumed, i.e. shorter codewords can be used. In addition to the analysis, list decoding is introduced as outer code decoding. This facilitate the tracing of more than one pirate, and it also make the error analysis simpler.

4.1.2 Change outer code

In [Sch04a] Schaathun introduce a new scheme which uses the BS-RS scheme as inner codes and Reed-Solomon codes as outer codes. AG codes are also tried as outer code, and it is shown that these codes with large distance are much better than random codes if the inner code can be made large enough. The Reed-Solomon codes can be decoded with the Guruswami-Sudan algorithm, with complexity $O(n)$.

4.1.3 Change decoding algorithm

In [SFM05] Schaathun and Fernandez-Muñez improve the decoding algorithm of the BS-CS scheme by using soft output from the inner decoder. This permits the use of significantly shorter codewords. The only excisting scheme with comparable or better rates is the Tardos scheme (Chapter 4.2.2), but this scheme is subject to adverse selection. The new decoding algorithm designed in [SFM05] has complexity $O(M \log M)$, because the weight has to be calculated and compared for each codeword. This complexity is typical for fingerprinting schemes.

4.2 Different schemes

4.2.1 The BBK scheme

In their paper [BBK03], Barg, Blakley and Kabatiansky present binary fingerprinting codes secure against size-t collusions which enable the distributor to recover at least one of the pirates with probability of error $\exp(-O(n))$ for $M = \exp(O(n))$.

The construction of binary fingerprinting codes in the BBK scheme employs concatenation of two codes. A long outer code C_2 and a shorter inner code C_1. The code C_2 is error-correcting with large minimum distance, while the code C_1 has a (t, t)-separating property. The use of separating codes is one of the new ideas of this scheme. The inner codes in BBK are not themselves collusion-secure with the decoding algorithm in use, but the minimum distance in the outer code is large enough not only to trace, but to correct for some inner decoding errors. The inner code can have a very high error rate, because the outer code can be made powerful enough to correct it. Since the outer codes must correct tracing errors from the inner decoding in addition to the tracing, the outer codes must be larger than traceability codes. The theory on $(2, 2)$-separating codes are limited. Unless better separating codes can be constructed, the BBK scheme is worse than exponential in t, hence it will give a long code even for a moderate t. The best BBK inner code known is duals of BCH codes[SH03]. These are especially good for small t. For a survey on separating codes see [sag94]. Reed-Solomon and AG codes are also suggested as outer codes for the BBK scheme.

Often a fingerprinting scheme needs a randomizing secret key to limit the information available to the pirates. The key used by the BBK scheme is much shorter than that of other known schemes. Only the mapping from the outer code alphabet onto the inner code must be kept secret. This mapping must be chosen at random for each block. The key size of the BBK scheme is $n_2 \log q! = O(\log M)$ bits. The key size increase as rapidly in the length of the inner codes for BBK, as it does in the total length for other codes, hence the inner codeword lengths alone is greater than the total lengths of other schemes.

The BBK scheme was the first one to introduce a decoding algorithm with complexity logarithmic in M. The problem with the BBK decoding is that the inner decoding depends heavily on t. Every possible t-set of codewords has to be considered, giving exponential complexity in t. So even for a moderately large t, the BBK decoding algorithm is likely to be slow.

4.2.2 The Tardos scheme

Tardos [Tar03] introduced a new probabilistic scheme where the codes for M users are ϵ-secure against t pirates with code length $O(t^2 \log(M/\epsilon))$. The main results in this paper are the construction of short fingerprinting codes, and the proof of a matching lower bound for the length of any fingerprinting code. This construction improves the codes proposed by the BS-CS scheme whose length is approximately the square of this length. The code length can be calculated for any M and ϵ. The length is extremely good for large t, but unfortunately this scheme is subject to adverse selection. Therefore the scheme often is used as an inner codes. Concatenating a Tardos code with a larger outer code, will much likely hide the information leading to adverse selection.

One of the most important aspects about the Tardos scheme is that the probability of accusing a given innocent user is independent of the *Marking Assumption* and the number of pirates t. Hence an over-sized pirate collusion can rarely frame anyone.

Tardos uses random codes, so the entire random code is a secret key. This will created a huge key space. Tardos inner decoding complexity depends on t only through the dependency on n. The decoding algorithm has complexity $O(t^4)$.

The results presented in this paper also imply that randomizing fingerprint codes over a binary alphabet are as powerful as over an arbitrary alphabet.

4.2.3 The LBH scheme

With the paper [LBH03] Le, Burmester and Hu construct a probabilistic fingerprint code where at least one colluder in a collusion of up to t pirates can be traced with high probability. They prove that this code is shorter than the BS-CS code, and that it is asymptotically optimal when t is constant.

The length of the construction is $n = ln(M/\epsilon)/g(t)$ for any $t \geq 2$, where $g(t)$ depends only on t. For a constant t, this improves on the BS-CS construction by a factor of $O(\ln 1/\epsilon)$. For the special case where $t = 3$, the construction gives codes of length $n = 9851 \ln(M/\epsilon)$, which is better than the results presented by Sebe and Domingo-Ferrer [SDF02b],[SDF02a] when $M > 6000$.

This scheme is good for a small t, but terrible for a large one. It is also susceptible to adverse selection, hence we often use it as an inner code for concatenation. Because the length formula incorporate the M and ϵ-values, it can use any sizes for these parameters.

As for the Tardos scheme, LBH also uses random codes, so the secret key is the entire random code. The decoding depends on a CND related algorithm, but here the Hamming metric is not used. The inner decoding complexity depends on t only through the dependency on n, so it has complexity exponential in t.

4.2.4 Dual Hamming codes

In [HJDF00] Herrera-Joancomarti and Domingo-Ferrer show that for $t = 2$, collusion security can be obtained using the error-correcting capacity of dual Hamming codes [MS77]. With this code as the outer code, the 2-secure fingerprinting codes obtained are much shorter than the 2-secure codes obtained via the BS-CS construction. For M users the codeword length of this proposal is only M, and the same technique described in BS-CS can be used to reduce the lengths to $\log^{O(1)}(M)$. CND can be used as outer decoding.

4.2.5 Scattering and Dual Hamming codes

In [SDF02b] and [SDF02a] Sebe and Domingo-Ferrer construct 3-secure codes that given a relatively small number of possible buyers are much shorter than the general BS-CS construction. The basic idea is to compose a new kind of code, which is called scattering code, with a dual Hamming code. The scattering codes where designed in order to fight three pirates. It is used as an inner code for concatenation to reveal the most frequent bit value among the pirates, regardless of the pirate strategy. E.g. if the pirates see two ones and a zero, then inner decoding outputs one with probability $1 - \epsilon$.

This paper also shows that Hamming codes offer collusion security, as long as the colluder' strategy can be controlled.

In [Sch04b] it is proven that the construction in [SDF02a],[SDF02b] is insecure against the optimal pirate strategy. Then it is shown how to build secure schemes using scattering codes as inner codes and separating codes as outer codes. This new construction has very good rates for a reasonable number of users.

4.2.6 Separating codes

In [Sch03b] Schaathun proves that an asymptotically good family of $(2, 2)$-separating codes is 2-secure with ϵ-error, where ϵ tends to zero with increasing code size. This code also removes the risk of accusing an innocent user. On the negative side, no efficient tracing algorithm for the code is designed. For an overview of separating systems, see [Sag94].

Part II

The Simulation

Chapter 5

Introduction

This part of the paper will discuss the simulations done on the BS-CS scheme. This introductory chapter gives some background information needed to understand the simulation, while Chapter 6 will give simulation results and all conclusions made. Chapter 7 will mention some open problems for future study.

5.1 Background

The BS-CS paper gives formulas for calculating the code parameters such that the scheme is secure with high probability. As mentioned in Chapter 4 these formulas have been improved over the years. [Sch03a] gives a new error analysis that permits the use of shorter codewords. A simulation on the BS-CS scheme is interesting because all the parameters given by these formulas are based on theoretical bounds, the scheme will be t-secure with ϵ-error. As far as we know, no simulations on the BS-CS scheme exists. Hence the difference between the error rate given by the bounds and the empirical estimate is unknown. We will later see that this difference is difficult to find due to imprecise theoretical bounds. But we don't necessarily need to compare the simulation estimate to the theoretical bounds. It can be just as interesting to study the behavior of the error rate and running time when adjusting the code parameters. The next chapter will use this strategy, and hopefully give new inside to the BS-CS scheme.

5.2 What is performance?

Before the simulation results are presented it is important to discuss what performance is, in relation to fingerprinting. The following are important performance parameters in a fingerprinting scheme (and therefore important for an distributor to be able to control):

- To be able to trace the pirates with high probability. The size of ϵ determines the probability.

- To be able to use as short fingerprints as possible. Shorter fingerprints require less embedding capacity.
- To be able to handle as many colluding pirates as possible. This parameter is a tricky one to set, as no codes are safe for an arbitrary t.
- Good running times, especially for the decoding algorithm.

In addition to these parameters a distributor must be able to control how many buyers the fingerprinting system shall support (M).

Some of these are conflicting goals. A good error probability will lead to longer codes and most likely longer running times. See Table 5.1 for an overview of how the goals behave in relation to each other. The $\searrow$ arrow (in the table) indicates that some, but not all, increases in the value will give better results (this observation is based on the results presented in Chapter 6).

Code parameter	Size of n	Size of ϵ	Dec. alg. time
Increase in q	$\uparrow$	$\downarrow$	$\uparrow$
Increase in r	$\uparrow$	$\searrow$	$\uparrow$
Increase in n_2	$\uparrow$	$\downarrow$	$\uparrow$

Table 5.1: Conflicting goals in fingerprinting.

A fingerprinting scheme should be evaluated for each M, according to the codeword length n, the error probability ϵ and the running time of the tracing algorithm. Hence we use a couple of M-values for the simulations. As we try different parameters we usually keep the n value constant to outline the behavior of the error probability ϵ. If n is kept constant and the error rate decreases, then we know that the code parameters used must be good. If n varies in size the situation would be unclear if ϵ and n goes in different directions. The running times given in Chapter 6 are the total running times of the program. But tables giving the decoding time are also presented.

5.3 Simulation facts

So what do we need in order to be able to run simulations on a fingerprinting scheme. We must of course implement the scheme. The C programming language was used for this task, and this implementation will be extensively covered in the implementation part of this paper. Here we just scratch the surface to give some information needed before we look at the simulation results in Chapter 6.

5.3.1 Theory and Practice

This subsection will differentiate between some of the expressions used in the BS-CS scheme contra some of the expressions used in the following chapters. The implementation follows the same principles as the BS-CS scheme, but some operations are not needed, while a different order of operation is used for others. These changes represent nothing new, they are only implementation decisions taken to support larger codes and better running times.

Code Generation
When we talk about code generation in the BS-CS scheme, we think of the generation of the inner and outer codewords, and the concatenation of these. But for an implementation this will not be effective. We don't need to create the full concatenated code book before a pirate collusion creates it's hybrid fingerprint. We only create and store the outer code book, and as the pirate collusion wishes to make a hybrid fingerprint, the inner codewords are generated based on the outer codewords. So what we call code generation in the simulation is just the generation of the outer code book. The creation of the inner codewords will be carried out when the pirate collusion makes it's hybrid fingerprint. And the creation of the concatenated code will never take place, as we now will see.

Making/Tracing of the Hybrid Fingerprint
The outer code book has been generated and it is time for a pirate collusion to create it's hybrid fingerprint. To do this the collusion will look at one inner codeword at a time, based on the numbers in the outer codeword. So instead of finding the feasible set of all the concatenated codewords in an pirate collusion, the program finds the feasible set of one concatenated codeword block at a time. After the feasible set is found for one block, the collusion generates a block of the hybrid fingerprint. The program then traces this block at once, using *Algorithm 1* (from the BS-CS scheme). This will return a number within the outer code alphabet, which is then given as input to the outer code decoding. This outer code decoding will be carried out after all blocks of the hybrid fingerprint are generated, and traced with the help of *Algorithm 1.* So essentially the implementation only breaks one large operation into many smaller parts. We split the operations on the concatenated code into operations on the inner codewords. Hence we never create the concatenated code.

Embedding and Random Permutation
In a real system we need to embed the fingerprint in a digital object, and then extract it before we use the tracing algorithm. This is not necessary for simulation purposes. Here our only interest is how to make and trace the hybrid fingerprint. Since we use the marking assumption, the embedding is irrelevant. The only marks that are detectable by an collusion are the marks that differ in their fingerprints. So we skip the embed/extract part of the scheme.

The same goes for the permutation on the fingerprints. The permutation shuffles the marks in the fingerprint to hide the correspondence between the marks and their positions in the fingerprint. This disguise is not needed for the simulation because the pirate strategy is a random generation of binary digits for the detectable marks. The permutation will not influence the simulation results in any way, so it is skipped.

5.3.2 Pirate Strategy

The BS-CS scheme is not designed with a particular pirate strategy in mind, it protects against any attack provided that the marking assumption holds. Under the marking assumption no strategy can raise the error probability of the scheme to more than ϵ. Hence none of the simple attacks described in Section 2.2.2 works against the BS-CS scheme, so all strategies used in the simulation should give estimates to the error probability no worse than the error bound specified in the BS-CS paper.

The pirate strategy used for the simulations is a random, independent generation of binary digits for all detectable marks. This strategy is the same as Yoshida [YII98] uses in his error analysis. Yoshida proves that the BS-CS scheme is much better than specified for this particular attack. It is important to remember that the results presented in Chapter 6 is based on this particular pirate strategy. Other strategies would most likely lead to other results.

Simulations on other pirate strategies are left as future work. See Chapter 7 for more on this.

5.3.3 The Simulation input

The simulation input can be divided in three: The variables that specify the amount of simulation-runs, the code parameters and the collusion-size parameter.

Simulation parameters:
We must be able to control how the simulation will behave. That is, how many times the given parameters will be tested by the simulation. We have two parameters that specify how extensive the simulation will be:

- The Number of Outer Code generations (NOC): This parameter controls how many times the program generates a new outer code.
- The Number of Pirate Collusions (NPC): This parameter controls how many pirate collusions that will make hybrid fingerprints on a given outer code.

Together these give the total number of simulation-runs. Please note that a set with NPC totally new collusions will be made for each outer code generation.

Example 9 *We decide to make 500 outer codes, and let 100 pirate collusions make a hybrid fingerprint on each of these outer codes. Then we have the parameters:*

$$NOC = 500, NPC = 100$$

Hence a total of $500 \cdot 100 = 50000$ *simulation-runs.*

The simulation parameters used in Chapter 6, and why these are chosen, are outlined in Section 6.2.

Code parameters:
The code parameters decide how the inner and outer code will look like, hence the input of different parameter sizes will provide us with the results we need in order to draw conclusions. The code parameters are:

1. The inner code parameters: The number of codewords q, and the replication factor r. The replication factor r and q together set the inner code length: $n_1 = r(q-1)$.
2. The outer code parameters: The number of codewords M, the alphabet size q, and the code length n_2.

The collusion size:
The number of pirates in an collusion must also be given as input to the program. This is needed in order to create the appropriate pirate collusion sizes. The collusion-size is denoted t.

5.3.4 The Simulation output

The purpose behind the simulation is to provide some overview on how good the scheme really is when we use various input parameters. The following are the main measures on the parameters, and therefore the simulation output:

1. The error rate: The amount of tracing errors made by the simulation.
2. The running time: The time used by the simulation.

To define an error in the simulation, we use the same notation as in Chapter 2, but for the simulation (and for the BS-CS scheme) exactly one user will be outputted as guilty. The two types of errors are:

1. Type I error: No user is returned as guilty.
2. Type II error: An innocent user is returned as guilty.

The simulation output does not distinguish between these two types. An error is here a combination of the two error types.

Example 10 *If we have a total of* 50000 *simulation-runs the maximum number of errors is of course* 50000*. That is, the tracing fails for all the hybrid fingerprints made. Such an error rate indicates the use of extremely bad parameters.*

If the running time had been equal for all input parameters, our only concern would be the fingerprint length and the error rate. But this is unfortunately not the case, hence the error rate and the running time are most often conflicting goals. See Section 5.2 for more on this. Chapter 6 will show that increased parameters, and therefore longer running times, not always give better error rates. This is especially true for the size of r.

5.4 Random number generator

An important aspect of the implementation is the use of a random number generator, or short RNG. This section will give an introduction, and then discuss what the simulation requires from an RNG. Finally the RNG choice is made and presented.

5.4.1 Introduction

Definition 9 *Randomness is a condition in which any individual event in a set of events has the same mathematical probability of occurrence as all other events within the specified set.*

Randomness and random numbers have always been used in one way or another, for example in dice games. But when the time came to introduce randomness into computers, problems emerged. It is very difficult to get a computer to do something by chance, because all computers follows the instructions made by programmers. No perfect solution exists, so no RNG is appropriate for all tasks. We can divide the RNGs in two main classes:

- Pseudo-random number generators.
- True random number generators.

Pseudo-random number generators (PRNG) are not truly random, but computed from mathematical formulas or taken from precalculated lists. The pseudo-random generators are predictable if we know where in the sequence the first number is taken from. Such a predictability can be both good and bad, depending on the purpose of the PRNG. Cryptographic application should avoid such generators if possible, because here the numbers should be truly unpredictable. Since a PRNG is fast, it is perfect for Monte Carlo simulations.

For cryptographic applications we usually use true random number generators (TRNG). True random numbers are typically generated by sampling and processing a source of entropy outside the computer. This source can be everything from a keystroke to noise from a radio. A TRNG will usually be much slower than a PRNG because more time is needed to process the extern entropy.

5.4.2 The requirements

The simulation needs an RNG at three stages:

1. Generation of the outer codes.
2. Generation of the pirate collusions.
3. Generation of the hybrid fingerprints.

Before we give the RNG requirements for the simulation, it is appropriate to state some general requirements for a RNG. The following criterions are often used to measure the performance of an RNG:

- Good Theoretical Basis: It is possible to prove mathematically how good the performance is. Theoretical tests is known to give very reliable predictions of the performance, but cannot guarantee good empirical performance (only a reliable forecast). Examples of such tests are the *Discrepancy test*, the *Spectral test* and the *Weighted spectral test*.
- Pass empirical statistical tests: These tests are prototypes of simulation problems. We test the RNG against various problems, and examine the quality of the output. No RNG passes all tests, hence we must decide the importance of the tests based on our use of the RNG. Examples of such tests are the famous *Diehard test*, the *Load test* and the *NIST test*.
- Long period: Long period before it repeats the numbers it generates. This will make it harder to predict the numbers generated.
- Efficient: Short running time.
- Repeatable: With the same seed, the RNG will create equal numbers. This is good for some applications and bad for others. It is important in scientific experiments, where it is necessary to be able to replicate the results made in order to view them as valid data.
- Portable: Easy to implement on different hardware and operating systems.

For the simulations in this paper we emphasize the criterions: Efficient and long period. The amount of numbers that will be created is high, so a slow RNG is out of the question. And because the amount is high, the period also is important. The size of the period needed will now be discussed. This calculation is done with numbers significantly higher than the actual simulation parameters, and it is independent of the number of bits in each number to be generated (a period of 10 will therefore generate 10 numbers, regardless of the number size, before it repeats itself).

All the parameters defined in Section 5.3.3 are important for the size of the period:

1. M: The number of codewords in each outer code is set to 2^{30}.
2. n_2: The outer codeword length is set to 2^{15}.
3. t: The size of the pirate collusion is set to 2^{10}.

4. NOC: The number of new outer codes generated is set to 2^{20}.
5. NPC: The number of pirate collusions on every new outer code is set to 2^{20}.

We first discuss the period needed by the generation of all the outer code books. For each outer codeword $M = 2^{30}$ we must generate $n_2 = 2^{15}$ numbers. Hence a NOC value of 2^{20} will give a total of $2^{30} \cdot 2^{15} \cdot 2^{20} = 2^{65}$ numbers to generate.

The generation of the pirate collusions needs a period of 2^{50}. For each new outer code generated (NOC = 2^{20}), generate a given number of collusions(NPC = 2^{20}) with $t = 2^{10}$ pirates in each. $2^{20} \cdot 2^{20} \cdot 2^{10} = 2^{50}$.

The generation of the hybrid fingerprint needs a period of 2^{55}. For each NOC, NPC collusions will generate a pirate fingerprint of length n_2. $2^{20} \cdot 2^{20} \cdot 2^{15} = 2^{55}$.

Hence $2^{65}+2^{50}+2^{55}$ numbers are generated. The real number of generations in the simulations will be less than this, but to be on the safe side we choose a RNG with a period better than this estimate. The next subsection will discuss some of the RNG tried, and the choice for the simulation. This choice will have a significantly larger period than strictly needed.

5.4.3 The choice

Both PRNG and TRNG generators have been tested. First we will discuss two of the generators considered, then the choice is presented.

rand()
The *rand()* generator is a standard C-library function, which returns a pseudo-random integer between 0 and a MAX constant. A seed value is specified with the function *srand()*. If the same seed value is given more than once as input, then the same number sequences is generated by *rand()*. The standard does not say anything about the period of this generator, so it is unclear if the period is large enough for the simulations presented in this paper. On the positive side, it is definitely a fast generator.

/dev/(u)random
The generators */dev/random* and */dev/urandom* are files in a unix system that is a source for random bytes generated by the kernel random number generator device. These generators create high quality random numbers for cryptographic purposes. The */dev/random* generator is a TRNG, while the */dev/urandom* generator is a mixture of a TRNG and a PRNG.

The */dev/random* interface only returns random bytes when the correct amount of entropy has been collected. So if there is too little entropy to produce the requested number of bytes, */dev/random* will block until more entropy can by collected. This blocking will lead to long running times when many random numbers are needed.

The */dev/urandom* interface returns bytes regardless of the amount of entropy available. If there is too little entropy, it switches over to the use of a hash function to produce more pseudo-random bits. Hence the */dev/urandom* generator is less safe, but much faster than */dev/random*.

Mersenne Twister - The choice
Since the period of the *rand()* generator is unknown to us, and the */dev/(u)random* generators are too slow, we picked a PRNG called *Mersenne Twister*, which is due to Makoto Matsumoto and Takuji Nishimura [MN98]. This generator is based on linear recursion, hence it is not cryptographically secure. A pseudo-random number generated by a linear recursion is insecure, since we can predict the rest of the output if we have a sufficiently long subsequence of the output. Fortunately the period of this generator is $2^{19937} - 1$ and a 623-dimensional equidistibution property is assured. Another great aspect with this generator is that it is very fast, and the use of memory is efficient (it consumes only 624 words of working area). *Mersenne Twister* has passed many stringent tests, including the *Diehard* test and the *Load* test.

Comparison of the generators
This comparison, as the only one in this paper, was executed on a Intel Pentium Centrino 1.4 laptop, with 512 MB ram. The reason for this is simple, the cluster used for all other simulations has no extern input, so the use of *dev/random* is impossible (*/dev/urandom* did not work either).

Generator	Generation of the outer code
rand()	4 sec.
/dev/random	-
/dev/urandom	8 min.
Mersenne Twister	5 sec.

Table 5.2: Running times for the RNGs, with parameters $M = 2^{12}$, $q = 30$, $n_2 = 1000$, NOC = 1 and NPC = 1

We see from the simulation that the *rand()* simulation is the fastest one, closely followed by the Mersenne Twister. */dev/urandom* uses significantly longer time, while */dev/random* blocks until input is given (so a good estimate on this generator is difficult to achieve).

Chapter 6

The results

This chapter will discuss the results found through the simulations on the BS-CS scheme. We will look at the error rates and running times given various code parameters, but first some formulas for calculating these parameters are given.

6.1 The formulas

The BS-CS scheme provides formulas for calculating the parameters needed to achieve a t-secure fingerprinting scheme with ϵ-error. Improvements on these bounds exist, among these the results presented by [Sch03a] (HGS). The following subsections will present and compare these formulas.

6.1.1 BS-CS formulas

The BS-CS formulas give a t-secure (n,M) code with ϵ error, if $q = 2t$ and

$$n_2 = \lceil 2t \log \tfrac{2M}{\epsilon} \rceil, \qquad r = \lceil 2q^2 \log \tfrac{4qn_2}{\epsilon} \rceil$$

$$n_1 = r(q-1), \qquad n = n_1 \cdot n_2$$

6.1.2 HGS formulas

The HGS formulas give a t-secure (n,M) code with ϵ error, if $q = 2t$ and

$$n_2 = \frac{max\{-\log \epsilon_1, \log M - \log \epsilon_2\}}{D(\frac{1}{t+1} || \frac{1}{2t})}, \qquad r = \lceil 2q^2(3 + 2\log t) \rceil$$

$$n_1 = r(q-1), \qquad n = n_1 \cdot n_2$$

where $\epsilon_1 = \epsilon/2$, $\epsilon_2 = \epsilon/2$

Here $D(x||y)$ is the relative entropy defined as follows:

$$D(\sigma||p) = \sigma \log \frac{\sigma}{p} + (1-\sigma) \log \frac{1-\sigma}{1-p}$$

6.1.3 Notes on the formulas

The two formula definitions given in the previous sections give different code lengths. This fact will first be outlined with an example, then a closer look at the r and n_2 parameters will be given:

Example 11 *Let* $\epsilon = 10^{-10}$, $t = 10$, $M = 2^t$ *and* $q = 2t$.
The BS-CS formulas give: $r = 39465$, $n_1 = 749835$, $n_2 = 885$, $n = 663603975$.
The HGS formulas give: $r = 7716$, $n_1 = 146604$, $n_2 = 2139$, $n = 313585956$.

We see that the parameters calculated by the HGS formulas are good improvements, with a total code length only half the size of the original formula. Later in this section we will see that the ϵ-value determines the relationship between the lengths.

The r parameter.
If we compare the formulas given in order to calculate r, we see that the formula in the BS-CS scheme contains the ϵ and n_2-values. That is, the size of r partly depends on the size of ϵ and n_2. For the HGS formula, r only depends on q and t. This fact will always favor the HGS formula, but the effect is more noticeable as the ϵ-value decreases. Table 6.1 shows that as the ϵ-value moves

ϵ	r (BS-CS)	r (HGS)
10^{-1}	14248	7716
10^{-5}	25635	7716
10^{-10}	39465	7716
10^{-15}	53120	7716

Table 6.1: r-values for BS-CS and HGS with $t = 10$ and $M = 2^t$

towards zero, the BS-CS r-value will increase while the HGS r-value is static. Even with an $\epsilon = 10^{-1}$, the r-value for the BS-CS formula is twice as high as the r-value calculated with the HGS formula.

Simulation results presented later in this chapter will state that the r-value calculated for both these formulas are unnecessary high. The security can be maintained with much smaller r-values.

The n_2 parameter.
The formulas for calculating the outer codeword length n_2 also give interesting differences. The HGS formula calculates the lowest r parameter, but for calculating the n_2 parameter the victory goes to the BS-CS formula. Table 6.2 shows some n_2 values for different sizes of ϵ. We see that the BS-CS numbers are less then half the size of the HGS numbers. And the difference between the numbers increases as the ϵ-value decreases.

ϵ	n_2 (BS-CS)	n_2 (HGS)
10^{-1}	287	693
10^{-5}	553	1336
10^{-10}	885	2139
10^{-15}	1217	2942
10^{-20}	1549	3745

Table 6.2: n_2 values for BS-CS and HGS formulas with $t = 10$ and $M = 2^t$.

ϵ	Formula	n_2	r	n
10^{-1}	BS-CS	287	14248	77694344
10^{-1}	HGS	693	7716	101596572
10^{-2}	BS-CS	353	17144	114984808
10^{-2}	HGS	854	7716	125199816
10^{-3}	BS-CS	420	20002	159615960
10^{-3}	HGS	1014	7716	148656456
10^{-4}	BS-CS	486	22828	210793752
10^{-4}	HGS	1175	7716	172259700
10^{-5}	BS-CS	553	25635	269346945
10^{-5}	HGS	1336	7716	195862944

Table 6.3: Sample lengths for $t = 10$ and $M = 2^t$.

The conclusion
The two approaches we have looked at here are the BS-CS scheme and an improvement. We see from Table 6.3 that the drawback of an large n_2 causes the HGS formulas to give longer code lengths when the ϵ-value is high. But as the ϵ-value decreases, the HGS formulas give better code lengths, and the difference in the lengths will grow as ϵ decreases. This is caused by the significant increase in the BS-CS r-values. For the BS-CS scheme both parameters grow, for the HGS scheme only n_2 grows.

The HGS formulas give a larger n_2-value and a shorter r than the original BS-CS scheme. We will later see that this is significantly more secure than the contrary.

6.2 NOC and NPC

This section will give some examples on how the simulation is affected by the NPC and NOC variables. How will the simulation times and error rates behave if these variables are adjusted? Table 6.4 provide some inside. Here all adjustments give approximately the same total number of simulation-runs. For the case $t = 10$ the total number of simulation-runs are $5 \cdot 10^7$, for the case $t = 20$ it is $5 \cdot 10^4$. Table 6.4 shows that the adjustments mainly affect running

t	q	NOC	NPC	Errors	Error rate	Running time
10	20	10000	5000	2326	$4.65 \cdot 10^{-4}$	29.55 hours
10	20	7071	7071	2357	$4.71 \cdot 10^{-4}$	29.32 hours
10	20	5000	10000	2434	$4.87 \cdot 10^{-4}$	29.16 hours
20	100	500	100	24	$4.8 \cdot 10^{-4}$	21.17 hours
20	100	224	224	26	$5.2 \cdot 10^{-4}$	19.03 hours
20	100	100	500	27	$5.4 \cdot 10^{-4}$	17.53 hours

Table 6.4: $M = 2^t$, $n_2 = 200$, $r = 5$, and NOC $\cdot$ NPC simulation-runs.

times, this is especially true for the $M = 2^{20}$ case. It looks like a large NOC gives better error rates, and for this no good reason could be found (we have also done simulations on two other sets of NOC/NPC values, and they gave the same results). In theory the error rate should be independent of the number of collusions that creates hybrid fingerprints on each code. A good idea could be to examine this result more thoroughly, by executing more simulations on each of the rows in the table, and examine each result against the average value. Then a more trustworthy result would have been achieved.

As we increase the NOC parameter and decrease the NPC parameter, we will get longer running times. A look at the outer decoding will explain this. The input to the outer decoding will be the outer codewords returned from the inner decoding for all collusions. The outer decoding will then compare each of these codewords against all codewords in the code book. So when the NOC value is small and the NPC value is large, the outer decoding will be faster because more returned outer codes will be compared against the outer codewords in the code book at a time. Hence less runs (NOC runs) through the code book is necessary. This will give a more significant improvement for the case $M = 2^{20}$, because here the code book is large.

6.3 ϵ - The error rate

The codes which are t-secure with ϵ-error enables a distributor to capture a member of a collusion P with probability at least $1 - \epsilon$. A small ϵ is good for

the security, but bad for code lengths and running times.

An ϵ-value of 10^{-10} is often used in the fingerprinting literature to ensure codes with a low error probability, but this size has some drawbacks for simulation purposes:

- Running times will be long due to large parameters (some of the code parameters are calculated with ϵ as one of the variables).
- The number of simulation-runs will increase. The purpose behind the simulation is to find an estimate to the error probability. This estimate will most likely be much better than what the bounds specify. So with an $\epsilon = 10^{-10}$, we must run the simulation more than 10^{10} times.

The combination of these gives a simulation-run with infeasible running time. Hence the original goal of finding the difference between the theoretical bounds and the simulation estimate becomes a difficult one. The following example will outline this fact:

Example 12 *For the ϵ-value 10^{-10} the shortest code is calculated by the HGS formulas. We have $t = 10$, $M = 2^t$, $q = 2t$, $n_2 = 2139$ and $r = 7716$. One simulation-run with $NOC = 1$ and $NPC = 1$ will only take 5 seconds. But an estimate for a simulation with a total number of 10^{10}-runs is approximately 1522 years. This is of course impossible.*

6.3.1 Conclusion

The original goal was to compare the theoretical error bounds with the simulation estimate. But as shown, the use of the formulas give infeasible running times, even if the ϵ-value is high.

So instead of comparing theoretical bounds with simulation estimates we will choose parameters that outline the behavior of the error rates and the running times. Hence the new goal is to provide patterns that tell something about the relationship between the parameters and the error rates/running times returned. Is it for example more efficient to increase n_2, in error rates and running times, then to increase r? The answer to this and similar questions will be given later in this chapter.

6.4 M - Number of users

We use fingerprints to mark digital copies uniquely for every user. One of the most important aspects is therefore to ensure that the code used supports the number of potential buyers. In the real world an impressive number of buyers would be 10^9, but a more realistic number may be in the range $10^6 - 10^8$. It is vital to keep the number of users M as low as possible, because an increase in M will affect the simulation time negatively in several ways (Table 6.5 outlines the times used by outer code generation and tracing for M^{10} and M^{20}).

t	n_2	r	q	OC gen. time	Tracing time
10	287	14248	20	< 1 sec.	< 1 sec.
20	973	65825	40	83 sec.	40 sec.

Table 6.5: Outer code generation and tracing running times for different sizes of M. BS-CS formulas used with $M = 2^t$, $\epsilon = 10^{-1}$, NOC = 1 and NPC = 1.

- To support M number of buyers it is necessary to generate at least M outer codewords. As M grows, so does the time needed to generate the outer code.
- If we use the BS-CS or HGS formulas longer outer codewords will be created because here the n_2-value partly depends on the size of M. Hence more outer codeword numbers must be generated by the RNG (therefore the running time increases).
- If the outer codeword lengths increase, the number of inner codeword creations also increase, hence more time is needed to create the hybrid fingerprint. The M value itself does not affect the times needed to create the hybrid fingerprint, because we only compare t codewords to find the feasible set, not M codewords.
- A larger M will generate a larger outer code book, hence more time is needed by the tracing process. The entire code book, an $M \cdot n_2$ matrix must be compared to the hybrid fingerprint.

The M parameter is adjusted only to provide for more users, it will not give better error rates. In fact an higher M-value needs larger code parameters to be secure with the same probability as before. The M values used in the simulation are: $M = 2^{10}$ and $M = 2^{20}$. The case $M = 2^{30}$ was tried, but found infeasible due to long running times. The use of different M values will give us the opportunity to examine, and compare, the simulation results archived by several M values. This will make the conclusions drawn more trustworthy.

6.5 t - Size of pirate collusion

A pirate collusion is a collection of pirates. These pirates compare their copies in order to find marks that differ. They then set these detectable marks according to a chosen pirate strategy to create a hybrid fingerprint. As the collusion size grows, so does the possibility of detecting marks, hence a large collusion can most likely discover more marks than a small one. This means that as the collusion size t grows, so does the code parameters if we want to keep the code t-secure with a good error probability. The results given by Table 6.6 and Table 6.7 outline the consequences of increasing the collusion size while keeping all other parameter sizes. We see that the error rate increases drastically as the

t	n_2	r	q	Errors	Error rate	Running time
2	200	5	20	0	0	2.22 hours
5	200	5	20	0	0	2.24 hours
10	200	5	20	203	$4.1 \cdot 10^{-5}$	2.53 hours
20	200	5	20	1083331	$2.1 \cdot 10^{-2}$	4.39 hours
30	200	5	20	3316025	$6.6 \cdot 10^{-1}$	7.16 hours
40	200	5	20	4126900	$8.2 \cdot 10^{-1}$	10.57 hours

Table 6.6: Differences in error rates and running times when adjusting the collusion size t. $M = 2^{10}$, NOC = 5000 and NPC = 1000.

t	n_2	r	q	Errors	Error rate	Running time
5	200	5	40	0	0	26.30 hours
10	200	5	40	0	0	27.22 hours
20	300	5	40	3878	$7.8 \cdot 10^{-2}$	33.02 hours
40	300	5	40	48314	$9.6 \cdot 10^{-1}$	33.36 hours
60	300	5	40	49790	$9.9 \cdot 10^{-1}$	33.03 hours
80	300	5	40	49988	$9.9 \cdot 10^{-1}$	33.10 hours

Table 6.7: Differences in error rates and running times when adjusting the collusion size t. $M = 2^{20}$, NOC = 1000 and NPC = 500.

pirate collusion size grows marginally. The running time also increases, due to the fact that more codewords must be compared to create the feasible set. In the fingerprinting literature it is common to set t to be $\log M$, hence this is done for the remaining simulations in this chapter.

6.6 r - The replication factor

The replication factor r specifies the number of duplications in the inner code. And together with q it sets the inner codeword length, $n_1 = r(q-1)$. In this section we will show how adjustments in the r-value will affect the error rate and the running times. In theory a larger r will give better error probability, but this section will show that this is not entirely true.

6.6.1 The simulation tables

Variations in r. Table 6.8 and Table 6.9 give error rates and running times for the simulations done with various r-values. The n value is not held constant here, because adjusting only the r parameter better outlines the behavior of

r	Errors	Error rate	Running time
7716	159	$3.18 \cdot 10^{-5}$	667.45 hours
3858	167	$3.34 \cdot 10^{-5}$	331.34 hours
1000	157	$3.14 \cdot 10^{-5}$	86.65 hours
500	164	$3.08 \cdot 10^{-5}$	44.32 hours
100	189	$3.78 \cdot 10^{-5}$	10.53 hours
10	161	$3.22 \cdot 10^{-5}$	3.12 hours
9	191	$3.82 \cdot 10^{-5}$	3.11 hours
8	152	$3.04 \cdot 10^{-5}$	3.12 hours
7	174	$3.48 \cdot 10^{-5}$	3.13 hours
6	185	$3.70 \cdot 10^{-5}$	2.53 hours
5	198	$3.96 \cdot 10^{-5}$	2.53 hours
4	357	$7.14 \cdot 10^{-5}$	2.53 hours
3	821	$1.64 \cdot 10^{-4}$	2.35 hours
2	5093	$1.02 \cdot 10^{-3}$	2.35 hours
1	212839	$4.25 \cdot 10^{-2}$	2.35 hours

Table 6.8: $M = 2^{10}$, $t = 10$, $q = 2t$, $n_2 = 200$, NOC = 5000 and NPC = 1000.

r	Errors	Error rate	Running time
37261	162	$3.24 \cdot 10^{-3}$	525.35 hours
18631	173	$3.46 \cdot 10^{-3}$	279.10 hours
1000	158	$3.16 \cdot 10^{-3}$	37.32 hours
500	157	$3.14 \cdot 10^{-3}$	28.46 hours
100	144	$2.88 \cdot 10^{-3}$	23.63 hours
10	146	$2.92 \cdot 10^{-3}$	22.50 hours
9	147	$2.94 \cdot 10^{-3}$	22.55 hours
8	174	$3.48 \cdot 10^{-3}$	22.56 hours
7	148	$2.96 \cdot 10^{-3}$	22.61 hours
6	159	$3.18 \cdot 10^{-3}$	22.14 hours
5	221	$4.42 \cdot 10^{-3}$	22.10 hours
4	416	$8.32 \cdot 10^{-3}$	22.13 hours
3	949	$1.89 \cdot 10^{-2}$	22.14 hours
2	4883	$9.76 \cdot 10^{-2}$	22.06 hours
1	31449	$6.28 \cdot 10^{-1}$	22.12 hours

Table 6.9: $M = 2^{20}$, $t = 20$, $q = 80$, $n_2 = 200$, NOC = 500 and NPC = 100.

the error rate. The n_2 value is chosen to give suitable error rates and running times. The largest r-values 7716 and 37261 are the r-values calculated by the

HGS formula. The q-value used in the case $M = 2^{20}$ is 80. This value is also chosen to give suitable error rates and running times.

q vs. r. Table 6.10 and Table 6.11 give error rates and running times for the simulations when the sizes of q and r are modified, while keeping the total length n constant.

q	r	Errors	Error rate	Running time
1001	1	914	$1.82 \cdot 10^{-4}$	2.45 hours
501	2	19	$3.8 \cdot 10^{-6}$	2.18 hours
334	3	7	$1.4 \cdot 10^{-6}$	2.09 hours
251	4	5	$1 \cdot 10^{-6}$	2.07 hours
201	5	7	$1.4 \cdot 10^{-6}$	2.03 hours
168	6	6	$1.2 \cdot 10^{-6}$	2.01 hours
144	7	34	$6.8 \cdot 10^{-6}$	1.59 hours
126	8	61	$1.22 \cdot 10^{-5}$	1.59 hours
112	9	114	$2.28 \cdot 10^{-5}$	1.57 hours
101	10	191	$3.82 \cdot 10^{-5}$	1.57 hours

Table 6.10: $M = 2^{10}$, $t = 10$, $n_2 = 50$, NOC = 5000 and NPC = 1000.

q	r	Errors	Error rate	Running time
1001	1	1714	$3.42 \cdot 10^{-2}$	12.04 hours
501	2	137	$2.74 \cdot 10^{-3}$	11.52 hours
334	3	88	$1.76 \cdot 10^{-3}$	11.50 hours
251	4	138	$2.76 \cdot 10^{-3}$	12.05 hours
201	5	278	$5.56 \cdot 10^{-3}$	11.47 hours
168	6	704	$1.4 \cdot 10^{-2}$	11.49 hours
144	7	1484	$2.96 \cdot 10^{-2}$	12.06 hours
126	8	2758	$5.51 \cdot 10^{-2}$	12.04 hours
112	9	4835	$9.67 \cdot 10^{-2}$	12.06 hours
101	10	7284	$1.45 \cdot 10^{-1}$	12.03 hours

Table 6.11: $M = 2^{20}$, $t = 20$, $n_2 = 100$, NOC = 500 and NPC = 100.

n_2	r	Errors	Error rate	Running time
500	1	195	$3.9 \cdot 10^{-5}$	6.18 hours
250	2	392	$7.84 \cdot 10^{-5}$	3.11 hours
167	3	2574	$5.15 \cdot 10^{-4}$	2.10 hours
125	4	12540	$2.51 \cdot 10^{-3}$	1.53 hours
100	5	40550	$8.11 \cdot 10^{-3}$	1.31 hours

Table 6.12: $M = 2^{10}$, $t = 10$, $q = 21$, NOC = 5000 and NPC = 1000.

n_2	r	Errors	Error rate	Running time
500	1	521	$1.04 \cdot 10^{-2}$	54.15 hours
250	2	911	$1.82 \cdot 10^{-2}$	29.32 hours
167	3	3438	$6.87 \cdot 10^{-2}$	20.30 hours
125	4	9358	$1.87 \cdot 10^{-1}$	16.15 hours
100	5	17331	$3.46 \cdot 10^{-1}$	14.06 hours

Table 6.13: $M = 2^{20}$, $t = 20$, $q = 81$, NOC = 500 and NPC = 100.

n_2 vs r. Table 6.12 and Table 6.13 give error rates and running times for the simulations when the sizes of n_2 and r are modified, while keeping the total length n constant.

6.6.2 The analysis

Variations in r. First we examine the error rates given by Table 6.8 and Table 6.9. The formulas stated by the BS-CS and HGS papers give bounds on the size of r. If the r-value used is lower than this bound, then the codeword used is not proven to be t-secure with ϵ-error. But as the tables show, the error rate is approximately the same for $r = 5$ as for the large r-value (7716). This means that the bounds given by the two approaches are very unprecise, with the use of this particular pirate strategy. As Figure 6.1 outlines, the error rate is similar for most values of r. But as r passes 5 and reaches towards 1, the error rate drastically increases. This fact is occurring for both cases of M, hence it should be a trustworthy result. The most obvious advantage of a decrease in the r-value is the fact that the codeword length will decrease. The case $t = 10$ gives codeword length $n = 29320800$ if $r = 7716$, and length $n = 19000$ if $r = 5$. The decrease in codeword length will also give the advantage of shorter running times. From Table 6.8 and Table 6.9 we see that the amount of time saved with the use of a small r is huge.

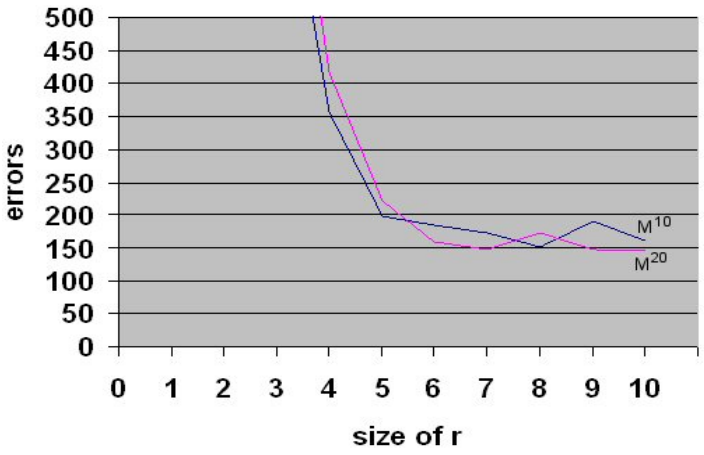


Figure 6.1: The size of r and the number of errors generated.

q **vs.** r**.** We now turn our attention to Table 6.10 and Table 6.11. These tables will outline a more precise estimate for the best r-value to choose. The tables show that the best cases are different for the two M-values. For $M = 2^{10}$ the best solution is to set $r = 4$ and $q = 251$, while the case $M = 2^{10}$ gives the optimal solution $r = 3$ and $q = 334$. But this result is not good enough to draw a final conclusion, because the r-values close to the optimal ones are only slightly worse. From the first tables considered we know that the error rates drastically increase if we set $r < 4$. This fact is also seen here as the error rate increases up to $r = 4$ (and $r = 3$), then it decreases even if the q-value continues to grow.

n_2 **vs.** r**.** We now look at Table 6.12 and Table 6.13. Here we see that the error rate gets better even if the r-value reaches 1. This means that an large n_2-value will be very important for the error rate.

A larger n_2 will also give significantly longer running times. As we double n_2 we also double the running time (the change from $r = 2$ to $r = 1$ will not affect the running times significantly).

Running times. Table 6.14 and Table 6.15 will outline the time used by different parts of the simulation as r grows.

- We see that the generation of the outer code is equal for both cases of r. This is due to the fact that the size of the outer code is independent of the r-value. For the case $M = 2^{20}$ the code book is larger, so longer time will be used to create it.
- The generation of the hybrid fingerprint is shown to be time consuming. And we can see that the dramatic difference between large and small r

Task	Time ($r = 5$)	Time ($r = 1000$)
Gen. of outer code	1.20 min.	1.20 min.
Gen. of hybrid fingerprint	58.35 min.	78.29 hours
Inner decoding	6.33 min.	8.21 hours
Outer decoding	1.51 hours	2.41 hours

Table 6.14: Running times for different parts of the simulation. $t = 10$, $M = 2^t$, $q = 2t$, $n_2 = 200$, $NOC = 5000$ and $NPC = 1000$.

Task	Time ($r = 5$)	Time ($r = 1000$)
Gen. of outer code	2.08 hours	2.08 hours
Gen. of hybrid fingerprint	20.21 min.	12.08 hours
Inner decoding	8.03 min.	1.01 hours
Outer decoding	18.34 hours	19.01 hours

Table 6.15: Running times for different parts of the simulation. $t = 20$, $M = 2^t$, $q = 80$, $n_2 = 200$, $NOC = 500$ and $NPC = 100$.

mainly is due to this generation (for the case $M = 2^{20}$ the outer code tracing is more time consuming).

- The inner decoding is fast for both cases of r, and for both cases of M. *Algorithm 1* take longer inner codes as input when $r = 1000$, hence the time needed will increase.
- The outer decoding will be approximately equal for both cases of r because the outer decoding is independent of r. The outer decoding depends only on the sizes of M and n_2, therefore more time are needed when $M = 2^{20}$.

The example below will outline the difference in inner code lengths for two different r-values, and why this is time consuming.

Example 13 *If* $r = 1000$ *and* $q = 20$ *then* $n_1 = 1000(20 - 1) = 19000$.
If $r = 10$ *and* $q = 20$ *then* $n_1 = 10(20 - 1) = 190$.
We see that the difference in inner codeword length is huge. And if we look at the total length, the time saved by cutting the r *becomes apparent. Let* $n_2 = 200$, *then* $19000 \cdot 200 = 3800000$ *for* $r = 1000$, *and* $190 \cdot 200 = 38000$ *for* $r = 10$.

6.6.3 The conclusion

Both cases of M show that the maximum value of r should be somewhere in the range of $5 - 10$. The fact that we can use smaller r-values will give shorter codewords, hence significantly better running times are achieved. And this

result also stress the fact that BS-CS and HGS formulas give bounds for r that are much higher than necessary. The BS-CS scheme is therefore much better than proven by these two approaches, given this particular pirate strategy. Yoshida [YII98] also pointed out that the r-value should be much lower than proven by the original scheme, but it is unknown to us if his bound is far from the results achieved here.

The observation done in this section indicates that the error rate will not be affected if we use a smaller r value with the simulations. The shorter running times achieved by this observation makes it possible to use larger values for the other code parameters in the simulations that follows.

6.7 n_2 and q

The parameter n_2 is the length of the outer codewords, it decides how many inner codewords the concatenated code is composed of. The numbers in the outer codeword must be within the alphabet q, where each number points to a codeword in the inner code. As mentioned the size of n_2 depends on the size of the ϵ-value for both the BS-CS and HGS formulas. This will cause the n_2-values calculated by the formulas to be infeasible for simulation purposes, hence we instead use n_2-values that give reasonable running times and error rates.

The q parameter is the glue that connects the inner code together with the outer code. For the inner code it determines the number of codewords, for the outer code it is the alphabet. So each unique number in the outer code points to one inner codeword. The q parameter also determines the length of the inner codeword, together with r ($n_1 = r(q-1)$). The literature often set q to be $2t$, but in this section the point is to examine the results made by different q-values, so here various q-values will be used.

6.7.1 The simulation tables

q vs. n_2. Table 6.16 and Table 6.17 give error rates and running times for the simulations when the sizes of n_2 and q are modified, while keeping the total length n constant.

n_2	q	Errors	Error rate	Running time
1000	6	79680	$1.59 \cdot 10^{-2}$	13.37 hours
500	11	157	$3.14 \cdot 10^{-5}$	6.31 hours
250	21	10	$2 \cdot 10^{-6}$	3.35 hours
125	41	12	$2.4 \cdot 10^{-6}$	2.16 hours
40	126	864	$2.73 \cdot 10^{-4}$	1.18 hours
20	251	14263	$2.85 \cdot 10^{-3}$	1.06 hours
10	501	113384	$2.26 \cdot 10^{-2}$	0.59 hours
5	1001	829917	$1.65 \cdot 10^{-1}$	0.55 hours

Table 6.16: $M = 2^{10}$, $t = 10$, $r = 5$, NOC = 5000 and NPC = 1000.

n_2	q	errors	Error rate	Running time
1000	21	5171	$1.03 \cdot 10^{-1}$	113.23 hours
500	41	640	$1.28 \cdot 10^{-2}$	59.66 hours
250	81	959	$1.91 \cdot 10^{-2}$	26.39 hours
160	126	2027	$4.05 \cdot 10^{-2}$	17.37 hours
125	161	3550	$7.1 \cdot 10^{-2}$	14.23 hours
80	251	7674	$1.53 \cdot 10^{-1}$	9.42 hours
40	501	18912	$3.78 \cdot 10^{-1}$	5.58 hours
20	1001	29100	$5.82 \cdot 10^{-1}$	4.11 hours

Table 6.17: $M = 2^{20}$, $t = 20$, $r = 2$, NOC = 500 and NPC = 100.

6.7.2 The analysis

n_2 vs. q. We now examine the results given in the Tables 6.16 and 6.17. The first observation is that an high n_2/low q will give better error rates and worse running times than the opposite. The use of a low n_2-value will not give good error rates even if the q-value is high. It is interesting to see that the best error rate is achieved when $q = 2t + 1$ for both cases of M. This indicates that the choice of $q = 2t$ taken in the literature probably is a vise one. If the q-value decreases below $2t$, the error rates will grow significantly even if the n_2-value is doubled.

Running times. Table 6.18 and Table 6.19 will outline which parts of the simulation that cause changes in running times when adjusting n_2 and q.

Task	Time($n_2 = 1000$ and $q = 6$)	Time($n_2 = 5$ and $q = 1001$)
Gen. of outer code	10.40 min.	2 sec.
Gen. of hybrid fingerprint	2.36 hours	42.15 min.
Inner decoding	14.06 min.	9.35 min.
Outer decoding	10.06 hours	3.10 min.

Table 6.18: Running times for different parts of the simulation. $t = 10$, $M = 2^t$, $r = 5$, NOC = 5000 and NPC = 1000.

Task	Time($n_2 = 1000$ and $q = 21$)	Time($n_2 = 20$ and $q = 1001$)
Gen. of outer code	11.26 hours	14.58 min.
Gen. of hybrid fingerprint	5.23 min.	2.53 min.
Inner decoding	8.12 min.	1.20 min.
Outer decoding	102.03 hours	3.16 hours

Table 6.19: Running times for different parts of the simulation. $t = 20$, $M = 2^t$, $r = 2$, NOC = 500 and NPC = 100.

- We see that the generation of the outer code is faster when n_2 is low. The code book is smaller, only $M \cdot n_2$. The outer code generation is independent of the q-value.
- The outer decoding will be faster when the n_2-value is low, because here the code book is small. The outer decoding is independent of q. It depends only on the size of M and n_2.
- We see that every part of the scheme use more time when n_2 is high.

6.7.3 The conclusion

The n_2 parameter is more important for error rates than the q parameter. But it is important to stress the fact that the q parameter should not be too low. An q-value below the standard $2t$ will give increased error rates even if we double n_2. Since the running times increase slowly for the q-value, it might be a good solution to increase the q-value in a real system.

6.8 A short comparison

This subsection will try to give an answer to one of the initial goals. Is the parameters calculated by the BS-CS formulas higher than necessary to provide security with a certain probability? In order to answer this question we compare the theoretical bound defined in the BS-CS scheme with an simulation estimate. All the parameters used here, except the r-value, are calculated from the formulas defined by the BS-CS scheme. The original r parameter gives infeasible simulation times, so this can not be used. But as we have shown, the original r-value gives no better error rates than an r-value of 5. Since a small r will make the simulation much faster, we choose to use $r = 5$. We see

n_2	q	r	t	Errors	Error rate	Running time
287	20	5	10	28	5.6×10^{-7}	41.33 hours

Table 6.20: $M = 2^t$, $\epsilon = 10^{-1}$, NOC = 10000 and NPC = 5000.

that with a total of $5 \cdot 10^7$ simulation-runs the number of errors returned are 28. The parameters used guarantees that no errors occurs, with probability $1 - \epsilon$. The result above shows that the probability of success is much better, $1 - 5.6 \times 10^{-7}$. Hence we see that the difference between the theoretical bound and the empirical estimate is significant, even after the r-value is 'corrected' according to our simulation results.

6.9 Summary

The most important result achieved in this paper is the observation that much shorter r-values can be used. This will give just as good error rates, while using much shorter codewords (will also result in better running times).

A small increase in q will give improved error rates with a low cost in running times. Hence it is recommended to use a q-value above the standard $q = 2t$. A q-value below $2t$ should be avoided (this will give a significant increase in the error rates).

An increase in the n_2-value is good for the error rates, but extremely bad for running times. So how to set this parameter will be a trade-off between long running times and good error rates.

If we compare the BS-CS and HGS formulas, we see that the HGS formulas are more secure because it gives larger n_2-values and shorter r-values than the BS-CS formulas. But since we know that shorter r-values can be used, with equal error probability, the HGS formulas will actually provide a more unprecise bound (higher n_2-value then BS-CS). Since Section 6.8 shows that the BS-CS formulas are unprecise, it is given that the HGS formulas must be

even more unprecise.

Chapter 7

Open problems

This chapter will discuss problems that are open for future study. Some of these problems was initially planed to be a part of this paper, but due to the time limit they are still unresolved.

7.1 Compare results against theoretical bounds

Several papers have stated improvements on the theoretical bounds presented by the BS-CS scheme. Among these are the results presented by Yoshida in [YII98]. His results is particularly interesting because he uses the same pirate strategy as employed by the simulations in Chapter 6. Therefore it would be very interesting to examine his results, and compare these to the simulation results made in this paper. Do the simulation estimate give results that are significantly better than Yoshida's results? Because [YII98] came to our attention at a late time (and also because the paper only exists in Japanese), this question is left as an open problem. Some of Yoshida's work is outlined in [Mur04] (this paper is in plain English!).

It could also be interesting to take a closer look at the formulas from the BS-CS scheme and the formulas from [Sch03a]. Are there any unnecessary requirements in the error analysis?

7.2 Compare several schemes

Results made by simulations on different fingerprinting schemes would be very interesting to examine. Chapter 6 gives the empirical results for the BS-CS scheme given a particular pirate strategy, but the empirical results of other schemes are still unknown. Therefore an implementation of another scheme, for example the BBK scheme, would allow us to compare the empirical results of the schemes. Such a comparison between schemes is of course interesting, hence it is left as an open problem.

Most of the program code listed in Appendix *A* is made with reuse in mind, hence it should not be difficult to implement different fingerprinting schemes.

7.3 Compare pirate strategies

The results in Chapter 6 are only based on one particular pirate strategy. The theoretical bounds presented by the BS-CS scheme ensures that the error rate is lower than a given number with high probability. But it is not clear how different pirate strategies would affect the error rate. The use of different pirate strategies can give significant variations in the error rate below the specified bound.

The implementation of other strategies should be relatively easy. We just have to remove the random generation function and specify the new pirate strategy.

Part III

The Implementation

Chapter 8

The program

This and the following chapters will describe the computer program implemented to achieve simulation results. First some general program information are given, then Section 8.3 will discuss the computer requirements. Disk and memory usage will here be emphasized. A description of the program modules and a short user manual will then be given. See Appendix A for the C code listings.

8.1 The purpose of the program

The *bsSim* program is an implementation of the BS-CS scheme. The purpose of this implementation is to collect simulation results which can be compared to the results given by the theoretical bounds. The *bsSim* program outputs the simulation running time and the amount of tracing errors made. Since the program tasks are few, the number of options the user can provide also are few. Therefore the user manual is rather simple.

8.2 Design and implementation

The original idea was to acquire simulation results with the use of large code parameters as input, hence efficiency was important. Even when the case $M = 2^{30}$ was abandoned this efficiency demand stayed vital. The choice of C as the programming language was the first decision made to support fast running times. More details on the C language will be given in the next subsection. The program was implemented from scratch, because no implementations of fingerprinting systems were found. Other important decisions was the choice of RNG and how to represent the inner and outer codes. Chapter 5.4 gives information on the RNG chosen, while the code representation is discussed in the sections 8.3, 9.4 and 9.5. All simulations were executed on a IBM e1350 cluster. This supercomputer has 172 AMD/opteron 250(2.4 Ghz) processors,

with 3 Gigabyte of memory per node (2 cpu's per node). The operating system on the cluster was Redhat Linux.

The program is split into modules that represent main tasks in the scheme, such as generation of the outer code and tracing of the hybrid fingerprint. Each of these modules will be explained in Chapter 9.

The program should be robust. Error checking is executed on input parameters, memory allocation and file communication. The user provides program input at the command-line.

8.2.1 The C language

C is a middle-level language. Hence it combines the advantages of a high-level language with the functionalism of the assembly language. C was initially meant to be a language for systems programming, so it has many features that can make it much harder to reuse or maintain. Among the disadvantages are:

- Lack of automatic memory management.
- It is platform dependent (it must be recompiled for every platform).
- The type checking is extremely bad.
- It has very weak support for modularization.
- The indirection operator (*) is described by the C creator as 'an accident of syntax'.

But fortunately the language has a number of advantages too:

- Existing compilers are efficient.
- The lack of strong typing can also be an advantage, because it allows the programmer to do many things that probably would be caught as errors in a high-level language.
- Extremely easy to learn, with only 32 keywords.
- The operators for the bit operations are easily available and works on the integer level such that fast running times are guaranteed.

8.3 Computer usage

Before the BS-CS scheme was implemented, a choice had to be made: Which was most important, fast running times or the opportunity to provide for large codes? The second alternative was chosen, hence the outer code had to be stored at disk. This matter is discussed in detail in the following subsection. The choice of compatibility with large codes does not mean that the running times are unimportant. We will later see that the support for large codes only increases the running times slightly.

It is not common to differentiate between disk and memory, usually the term memory refers to all kinds of data storage on the computer. But here we separate disk from memory because a comparison of running times between the two will be given.

8.3.1 Disk

This section will examine what the program must store to disk, and how much disk capacity it needs given various parameters. The outer code book is the only part of the scheme that must be stored to disk. We could of course store the outer codewords in memory, but for large codes this would be infeasible. The following example will give some insight:

Example 14 *Let $t = 30$, $M = 2^t$, $q = 2t$ and $n_2 = 1000$. Then we need 804 GB to store the outer code.*

We see that the case $M = 2^{30}$ gives very large outer codes, even if the n_2-value is relatively small (compared to the values calculated by the BS-CS and HGS formulas). This case is not used in the simulation because it leads to infeasible running times.

Before we store an outer codeword to disk, we use an array as a buffer. The amount of numbers that will be generated and stored for one outer codeword are n_2, so this array stores each number until n_2 numbers are generated, then the array will be written to disk. This procedure will be repeated for all outer codewords. The buffer array has a block length of 32 bits. The variable $nrInBlock$ specifies the amount of numbers the program can store in one block. As the next example will show, the alphabet size q determines this.

Example 15 *If $q = 40$, all numbers in the outer code will be below 40, so we need maximum 6 bits to represent each number. Hence we store $\lfloor 32/6 \rfloor = 5$ numbers in each block.*

If the amount of storage space needed to save the outer code to disk is required, use the following formula:

$$\lceil n_2/nrInBlock \rceil \cdot M \cdot 32 = \text{Number of bits.}$$

The first part, $\lceil n_2/nrInBlock \rceil$, is the amount of blocks needed by the buffer array to represent the outer code. M is the number of supported users, while 32 is the block length.

Example 16 *Let $t = 20$, $M = 2^t$, $q = 2t$ and $n_2 = 1000$. Then each 32-bit block can store 5 numbers. The storage space needed is: $\lceil 1000/5 \rceil \cdot 2^{20} \cdot 32 = 6710886400$ bits $= 800$ MB.*

As we see from the example, the maximum number of bits we can store in one block is 32, but often some bits will be used to store irrelevant bits. The following example will outline why this is the case.

Example 17 *Let $t = 10$, $M = 2^t$, $q = 2t$ and $n_2 = 885$. Since $q = 20$ we need 5 bits to represent one number, hence each 32-bit block can store 6 numbers. The program needs 592 kB to store this code on disk. 2 bits in each block will not store relevant bits ($6 \cdot 5 = 30$). If all bits were used, the outer code file size would be 557 kB.*

When the buffer array is written to disk, this will include the bits that are irrelevant. As we have seen from the previous example, this is a waste of storage space. But as the program is primarily designed with a priority on efficiency, this code representation will be the best solution.

The values of the n_2 parameters generated by the BS-CS and HGS formulas are very different. From Table 8.1 we see that since n_2, the length of the outer

Formula	t	n_2	q	Storage space
BS-CS	10	885	20	592 kB
HGS	10	2138	20	1.39 MB
BS-CS	20	2169	40	1.7 GB
HGS	20	4512	40	3.5 GB
BS-CS	30	3854	60	3 TB
HGS	30	7629	60	5.9 TB

Table 8.1: BS-CS and HGS with $\epsilon = 10^{-10}$ and $M = 2^t$.

codewords, is longer for the HGS numbers, it will also require more storage space. In fact it requires approximately twice as much storage space as the BS-CS scheme.

8.3.2 Memory

This section will provide a memory usage overview. The outer code is as already mentioned stored at disk, but all other information will be stored in memory. The decrease in running time is one of the benefits from using the memory instead of disk. But in fact this disk communication is only a small bottleneck in the program. As Table 8.2 shows, it will only affect the overall running times slightly if the program stores the outer code in memory instead of at disk. The compatibility with large codes is more important than slightly faster running times, hence the outer code is stored to disk.

Table 8.3 outlines the difference between the memory usage of the BS-CS scheme and the HGS scheme. It is apparent that the HGS numbers give much better memory usage. The reason for this is simple. The r parameter given by the HGS formula is much smaller, hence the inner code will be shorter and smaller arrays are needed. The aspect that could favor the BS-CS memory usage is the short outer codewords, but since these are stored to disk the memory usage is unaffected.

The numbers in Table 8.3 are achieved with the NOP variable set to 1. This means that only one pirate collusion is used. If we set NOP = 1000, the memory usage will grow for both cases, but the increase is most significant for the HGS parameters. For the case $t = 20$ the BS-CS memory usage grows

Formula	OC storing	t	n_2	r	q	Time used
BS-CS	memory	10	885	39465	20	12 sec.
BS-CS	disk	10	885	39465	20	11 sec.
HGS	memory	10	2139	7716	20	5 sec.
HGS	disk	10	2139	7716	20	5 sec.
BS-CS	memory	20	2169	165197	40	18.07 min.
BS-CS	disk	20	2169	165197	40	18.48 min.
HGS	memory	20	4513	37261	40	-
HGS	disk	20	4513	37261	40	File size exceeded

Table 8.2: Running times when the outer code is stored to disk and memory. BS-CS and HGS formulas with $\epsilon = 10^{-10}$ and $M = 2^t$. $NOC = 1$ and $NPC = 1$.

Formula	t	Memory usage
BS-CS	10	1.5 MB
HGS	10	0.7 MB
BS-CS	20	17.5 MB
HGS	20	4.5 MB
BS-CS	30	82 MB
HGS	30	22 MB

Table 8.3: Memory usage for the BS-CS and HGS-schemes with $\epsilon = 10^{-10}$, $M = 2^t$ and $NOP = 1$.

to 25 MB, while the HGS memory usage grows to 21. The explanation: The tracing algorithm is executed once (one run through the outer code book), hence the program must collect the outer codewords for all collusions returned by the inner decoding (*Algorithm 1*) in an array. In the case of 1000 pirate collusions, this array will use more memory for the HGS formulas because the outer codewords are longer here (one array has size: n_2 multiplied with the NOC variable). So a growing number of pirate collusions will favor the BS-CS memory usage.

8.3.3 CPU

The program depends on the use of bitwise operators like AND, OR and XOR, hence a fast CPU is very important for the simulation running times. Especially the RNG, which is a significant part of a simulation-run, relies heavily on these operators.

The program does not use parallel algorithms because the simulations will not benefit from this. We can divide a simulation run between many processors by simply running the program more than one time on the cluster, and then add the number of errors achieved.

8.4 The header file

bonehShaw.h contains common macros, constants, error messages, exit codes, configuration settings and all the function definitions. The struct *codeParam*, which stores all the code parameters and properties is defined here.

Chapter 9

The modules

This chapter will give details on the program modules, hence it serves as a technical documentation. Most of the modules and functions are designed with reuse in mind, so they easily can be adapted to work with other programs. The following section will outline how the modules interact.

9.1 Overview

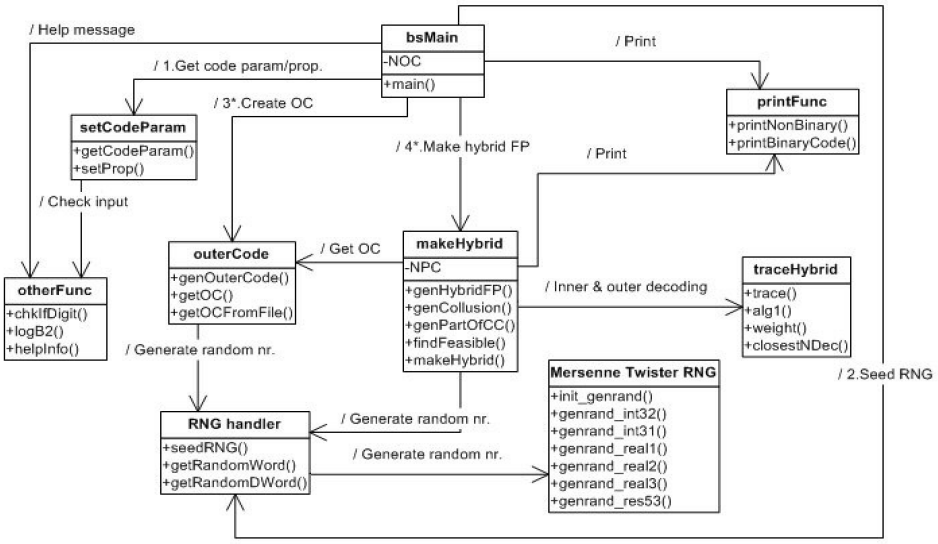


Figure 9.1: Module interaction.

Figure 9.1 shows all the interaction between the modules in the program. The main function is located in the module *bsMain*, and from this function the first 4 function calls are made. The * indicates that the function can be called several times (this is determined by the size of the NOC variable).

9.2 Program control

This module includes the main function, hence the program behavior will be controlled from here.

Module: *bsMain*
File: *bsMain.c*
We initiate the RNG, and a timer which is used to calculate the running time of the program. At the command-line the user specifies the program options. This information will be registered in flags, as the following example outlines.

Example 18 *If only outer code generation is needed, use the option* -o. *This choice will be stored in the flag: flagOC.*

The variable NOC (specified in *bonehShaw.h*) decides how many outer codes that will be created (the new one overwrites the old).

9.3 Setting the parameters

This module contains functions that read command-line input and calculate code properties. This information is saved in the struct *codeParam* defined in the header file.

Module: *setCodeParam*
File: *setCodeParam.c*
The function *getCodeParam()* reads the code parameters from the command-line. Some error-handling routines are implemented here. Wrong number of arguments will generate an error message, and the function *chkIfDigit()* will check if the parameters given really are digits. The collusion-size parameter t is not forced input, but only outer code generation will work if it is skipped. It is needed for creation/tracing of the hybrid fingerprint.

The function *setProp()* calculate and stores code properties. An example of such a property is the information on how many blocks required by an array to store the outer or inner codewords.

9.4 Handling the outer code

The outer code is one of the building blocks in the BS-CS fingerprinting scheme. This module handles all operations on this code.

Module: *outerCode*
File: *oc.c*
The primary function in this module is *genOuterCode()*. This function generates and stores the outer code to a file. An array called *outercode* is used to store the outer codewords for one user. When all numbers for one outer

codeword are generated and stored in the array, it will be written to file. This procedure is then repeated for all users. Some details on how an outer codeword is stored will now be given: Every block in the array has room for more than one number. How many is determined by the size of q. Usually the blocks will not be filled. See section 8.3.1 for examples.

The function *getOC()* first calls the function *getOCFromFile()*, which given a particular user number, returns this user's outer codeword. Since the latter function stores more than one number in one block, the *getOC()* function is used to extract the numbers and then store them in an array with only one number in each block (it is not strictly necessary to use this additional array, but it will greatly reduce complexity at later stages).

9.5 Generate hybrid fingerprint

To generate a hybrid fingerprint the program must, among other things, find the feasible set and generate the hybrid fingerprint. A lot of arrays are needed, hence an array overview will be given later in this subsection. First the most important functions will be mentioned.

Module: *makeHybrid*
File: *makeHybrid.c*
To create the hybrid fingerprint the function genHybridFP() is called. A fingerprint is a concatenation of several inner codewords. In the process of creating the hybrid fingerprint the program will look at each inner codeword (block) separately. First it finds the feasible set for the current inner codeword, then a block of the hybrid fingerprint is created. After one of the n_2 blocks of the hybrid fingerprint is created, the program sends it to the function *alg1()*, which is *Algorithm 1* (the inner decoding) in the BS-CS scheme. All the n_2 blocks will be handled in the same manner. The variable NPC (specified in *bonehShaw.h*) tells the program how many pirate collusions that will be created on each outer code. So when all the hybrid fingerprint blocks are made, and the inner decoding has been executed on each of them, the program jump directly to the next collusion. After this is done for all collusions, the *trace()* function described in subsection 9.6 is executed.

Creation of pirate collusions are handled by the function *genCollusion()*. It uses the RNG to randomly choose a pirate collusion of size t. The program checks that each user in an collusion is unique.

The function *getPartOfCC()* returns the inner codeword that corresponds to a given number in the outer code. The inner codewords are generated based on the outer code numbers. The next example shows how this work.

Example 19 *If* $q = 4$, $r = 3$ *and the number in the outer codeword (OCN) is* 2*, then the inner codeword* 000000111 *will be generated. The length* 9 *is given by the formula* $r(q-1)$*, while the number of zero's is determined like this: The*

OCN number is multiplied with r, hence we have $2 \cdot 3 = 6$. Therefore the 6 first positions will be zero's.

The *findFeasible()* function uses the current inner codewords for all pirates in an collusion to find the feasible set.

The *makeHybrid()* function generates a given number of 64-bit blocks randomly. This number of blocks equals the number of blocks in the inner code. After the generation, a test ensures that the hybrid fingerprint block is valid. We use the feasible set and one of the pirate codewords to check the bit positions for correctness. The following example will outline how this work.

Example 20 *We have a pirate collusion of size $t = 2$. Here pirate A has the inner codeword* 000111111*, while pirate B has the codeword* 000000000*. Hence the feasible set is* 000111111 *(pirates can detect the last* 6 *positions).*

A hybrid fingerprint is generated randomly: 10101110*. The program then checks the validity of the fingerprint against the feasible set. We see that position* 1 *and* 3 *are not valid because both pirate codewords have the value* 0 *in these positions. Hence these positions must be flipped according to one of the pirate codewords. All other positions are valid, and therefore kept.*

Array overview:

- everyPos: 2D array. After a block of the hybrid fingerprint is generated, this block will be given as input to the inner decoding algorithm *alg1()*. The *everyPos* array stores the outer codeword numbers returned from this algorithm. It is 2 dimensional because it stores the outer codewords returned for all pirate collusions. This is needed for effective outer decoding.
- feasible: This array stores the current feasible set block after all the pirates have compared their current inner codewords.
- hybridFP: This array stores the current hybrid fingerprint block generated.
- pirates: 2D array. This array contains all the pirates for all collusions. This is needed in order to determine if the tracing process returns the correct guilty users.
- piratesCol: 2D array. Here the current inner codeword for all pirates in one collusion is stored. These inner codewords will later be compared to create the current feasible set block.
- pirOC: This array stores all the outer codewords for one pirate collusion. This is needed to determine which inner codewords to use when creating the feasible set and the hybrid fingerprint.

9.6 Trace hybrid fingerprint

The tracing procedure can essentially be divided in two steps: The inner decoding and the outer decoding. This module will describe these tracing steps.

Module: *traceHybrid*
File: *traceHybrid.c*
The function *alg1()* is as already mentioned *Algorithm 1* from the BS-CS scheme. This function is called from the *for* loop in the function *genHybridFP()* (module *makeHybrid*). The inner decoding starts each time a block of the hybrid fingerprint is created, and it returns the number from the outer code alphabet which was most likely used to generate this current hybrid fingerprint block.

The function *weight()* is used by *alg1()* to calculate the weight of particular bit positions in a hybrid fingerprint block. The number of 1's for a particular area on the hybrid fingerprint are added together to form the weight.

After the outer codeword is returned from *alg1()* (based on all the hybrid fingerprint parts) for all collusions, the *trace()* and *closestNDec()* functions will compare these codes to the outer codewords in the code book. The user, for every collusion, which has the outer codeword that matches the returned pirate outer codeword in most positions (smallest Hamming distance), will be returned. If the returned user does not exist in the current collusion, then the program updates the error variable.

9.7 The Random Number Generator

The random number generator (RNG) is used by the program to generate the outer codes, the pirate collusions and the hybrid fingerprints.

Module: *Mersenne Twister RNG*
File: *mt19937ar.c*
The program is not hard coded to work with only one particular RNG, so it should be easy to change RNGs. Only the functions in the RNG handler must be adjusted. In this case the excellent generator *Mersenne Twister* is chosen. The generator is implemented in the file *mt19937ar.c* (this implementation is the work of the *Mersenne Twister* creators).

9.8 The RNG handler

All program calls to the chosen RNG will go through this module.

Module: *RNGhandler*
File: *rng.c*
When the program requires random numbers it will always call the functions

in this file. Here it is specified which RNG the program uses. This will make it very easy to change RNG.

9.9 Other functions

This module contains functions that have nothing in common, they are collected here because no other logical placing was appropriate.

Module: *otherFunc*
File: *otherFunc.c*
The *chkIfDigit()* function will check if a given character is a digit. It is used by the program to check if the parameters given at the command-line are digits. The *logB2()* function calculates the base 2 logarithm for a given number. The *help()* function prints information on where the user can acquire help.

9.10 Debugging

This module contains functions that are made for debugging purposes. When debugging we must use small parameters as input, because large parameters will be impossible to read due to screen limitations.

Module: *printFunc*
File: *printFunc.c*
The function *printNonBinaryCode()* reads all the outer codewords from the file *outercode*, and prints them on the screen.

The function *printBinaryCode()* prints the inner codewords on the screen. Also blocks of the feasible set or block of the hybrid fingerprint will be displayed with this function.

9.11 More implementation details

This section will further explain the most important operations in the program. Pseudo-code is used in order to describe the operations.

9.11.1 Generate and store the outer code

This subsection will outline how the outer code is generated and stored.

1. block length = the amount of numbers one outer codeword block can store
2. for($i = 1$ to the total number of users, M)
3. for($j = 1$ to the number of blocks in the outer codeword array)
4. if last outer codeword block, set block length=length of last block
5. for($k = 1$ to block length)

6. generate and store outer codeword number at:
7. a) the rightmost positions of the block, if first number in block
8. b) the next free bit positions, if not first number
9. block length = the amount of numbers one outer codeword block can store
10. write current outer codeword to file

An outer codeword for each user is now generated and stored at disk.

9.11.2 Generate hybrid fingerprint

Here we show how the hybrid fingerprint is created.

1. for($i = 1$ to the number of pirate collusions, NPC)
2. generate and store collusion i
3. for($j = 1$ to the number of pirates in collusion, t)
4. get the outer codeword for pirate j in collusion i
5. for($j = 1$ to the length of the outer codeword, n_2)
6. for ($k = 1$ to the number of pirates in the collusion, t)
7. get the inner codeword block j for pirate k
8. find the feasible set for inner codeword j
9. make the hybrid fingerprint block

The inner decoding (*alg1()*) is executed each time after step 9 (in the same for loop) if tracing (-t) is given as an option.

Chapter 10

Using bsSim

This chapter provides an description on the use of the *bsSim* program. The chapter is split into parts according to the task at hand. The last section will provide an error handling overview.

All program input are given at the command-line. The syntax is:

`[tor]$ bsSim -[OPTIONS]` n_2 M q r `[Optional` t`]`

10.1 Generate outer code

If the task is to create the outer code, and nothing else, we write an *-o* in the $[OPTIONS]$ field. In addition, we must provide the code parameters. The size of the collusion t is not needed, hence we give 4 parameters as input. If the t parameter is provided, the program simply ignores it. The output *Outercode generated and stored* will be printed each time the program creates a new outer code. How many times it appears are determined by the NOC variable. The size of the other variable, NPC, is irrelevant here, because no hybrid fingerprints are created. The outer code will be stored in the *outercode* file. The *total running time* will differ from the *OC running time* if NOC > 1.

```
[tor]$ bsSim -o 160 1024 20 5
Code parameters:  C_1 = (n_1, q) = (95, 20), C_2 = (n_2, M)_q = (160, 1024)_20,
r = 5, t = 0, n = 15200
Outercode generated and stored (Printed NOC times)
Generate OC running time = 0 seconds
Total running time = 0 seconds
```

10.2 Make hybrid fingerprint

To create a hybrid fingerprint in addition to the outer code, we use the two options *-o* and *-m*. Since the program here generates a hybrid fingerprint, we

need to specify the size of the pirate collusion (the t parameter must be given as input). If we only give the -m option as input, the outer code generation will be skipped. This will only work if the user already has created the outer code (with the same size). So even if NOC > 1, all the pirate collusions will use the same outer code in the process of creating their hybrid fingerprints.

```
[tor]$ bsSim -om 160 1024 20 5 10
Code parameters:  C_1 = (n_1, q) = (95, 20), C_2 = (n_2, M)_q = (160, 1024)_20,
r = 5, t = 10, n = 15200
Outercode generated and stored (Printed NOC times)
Generate OC running time = 0 seconds
Total running time = 0 seconds
```

10.3 Trace hybrid fingerprint

To trace the hybrid fingerprint created, use the options -o and -t. The option -t includes the creation of the hybrid fingerprint (option -m), because no tracing is possible if no hybrid fingerprint exists. If we only choose to give -t as input, the generation of the outer code will be skipped. To achieve complete simulation results, the options -ot must be provided. We see from the output that given these options the program returns the running time and the number of tracing errors made.

```
[tor]$ bsSim -ot 160 1024 20 5 10
Code parameters:  C_1 = (n_1, q) = (95, 20), C_2 = (n_2, M)_q = (160, 1024)_20,
r = 5, t = 10, n = 15200
Outercode generated and stored (Printed NOC times)
Total error:  0
Generate OC running time = 0 seconds
Total running time = 0 seconds
```

10.4 Other functions

We can give two additional options on the command-line:

Debugging The -p option is a debugging function. It prints simulation information to the screen. If given together with the -o option, the program will print all the outer codewords created. If given together with the -t option, it will print the pirate collusion's feasible set, hybrid fingerprint, and some tracing information. This print function will be useless with large codes, because the information printed on screen will be to extensive for the human mind to grasp.

```
[tor]$ bsSim -op 10 10 10 5
```

Code parameters: $C_1 = (n1, q) = (45, 10)$, $C_2 = (n2, M)_q = (10, 10)_{10}$, $r = 5$, $t = 0$, $n = 450$
Outercode generated and stored
Codeword $1 = 2188344570$
Codeword $2 = 0230692898$
Codeword $3 = 4115448737$
Codeword $4 = 3512517805$
Codeword $5 = 2498985607$
Codeword $6 = 1382158442$
Codeword $7 = 9468456449$
Codeword $8 = 6300604921$
Codeword $9 = 3255138292$
Codeword $10 = 6163096997$
Generate OC running time = 0 seconds
Total running time = 0 seconds

Help information The -*h* option is a help function. If this option is given, all other options are ignored. It will print a help message, which tells the user where additional help can be found.

```
[tor]$ bsSim -h
Read the manual for help
```

10.5 Error messages

This section will give a summary of the existing error messages. The first three are related to wrong input, while the last two can have more than one explanation and are therefore more difficult to explain.

No options and/or parameters given. This will give a simple message, telling the user the correct syntax. Wrong number of parameters will also trig this message.

```
[tor]$ bsSim
Correct syntax:  progName [OPTIONS] n2 M q r [Optional t]
```

One or more of the code parameters provided is, or includes, non-digits. The error message given will tell the user to provide only digits as code parameters.

```
[tor]$ bsSim -o 160 1024 20 5d
n2, M, q, r and t(optional) must be numbers.
Correct syntax:  progName [OPTIONS] n2 M q r [Optional t]
```

The option -*m* or -*t* is provided, but no collusion-size parameter t is given. A message telling the user to include the t parameter is printed. This message will also be printed if t is greater than M or smaller than 2.

```
[tor]$ bsSim -ot 160 1024 20 5
The number of colluding pirates(t) must be given as input (M > t > 1).
Correct syntax:  progName [OPTIONS] n_2 M q r [Optional t]
```

The program cannot open the file *outercode* for writing or reading.

```
Unable to open the file outercode.txt
```

The program can not allocate the necessary memory required.

```
Error on malloc
```

Part IV

The Appendices

Appendix A

The source code

This appendix will list the program code. Each section corresponds to a module in the program (the first section gives the header file).

A.1 The header file

```
1  /* bonehShaw.h                        */
2  /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7  #include <limits.h>
8  #include <math.h>
9  #include <fcntl.h>
10
11 /* Typedef */
12 typedef unsigned long word;
13 typedef unsigned long long dword;
14
15 typedef struct codeParam {
16   word n1;
17   word q;
18   word n2;
19   word M;
20   word t;
21   long r;
22   double ICepsilon;
23   double expr;
24   int nrOfBits;
25
26   int nrOfBlocksIC;
27   int lenLastBlockIC;
28   int nrOfBlocksOC;
29   int lenLastBlockOC;
30   int nrInBlockOC;
```

```
31    int nrOfBytes;
32  }cParam;
33
34  /* Define constants */
35  #define BS_WORD 32
36  #define BS_DWORD 64
37  #define ULLONG_MAX 0xffffffffffffffffULL
38
39  /* Define macros */
40  #define CLE printf("Correct syntax:  progName [OPTIONS] n2" \
41                     " M q r [Optional t]\n"), exit(1)
42  #define EOM(code)  ((code == NULL) ? printf("Error on malloc"), exit(1): NULL)
43  #define UOF printf("Unable to open the file %s",TXTFILE), exit(1)
44
45  /* Define config_options */
46  #define TXTFILE "outercode.txt"
47  /* The number of outer codes we want to create */
48  #define NOC 225
49  /* Nr of collusions we want to create on one outer code */
50  #define NPC 225
51
52  /* Functions */
53  cParam getCodeParam(int argc,char *argv[]);
54  int chkIfDigit(char argv[]);
55  cParam setProp(cParam p);
56  void genOuterCode(cParam param);
57  void printNonBinaryCode(cParam param);
58  int trace(long **everyPos, word **pirates, cParam p, int flagP);
59  void getPartOfCC(dword *innerCode, word *outerCode,
60          cParam p, word userNr, int OCpos);
61  void printBinaryCode(dword *concatCode, cParam p);
62  int genHybridFP(cParam p, int flagP, int flagT);
63  void makeHybrid(dword *hybridFP,dword **pirateCol,
64                  dword *feasible, cParam p, int flagP);
65  void findFeasible(dword *feasible, dword **pirateCol, cParam p, int flagP);
66  long alg1(dword *hybridFPblock, cParam p, int flagP);
67  void closestNDec(long **position,int *guiltyUsers,cParam p,int flagP);
68  word weight(dword *hybridBlock, cParam p, int s);
69  void getOC(int userNr, word *numbers, cParam p);
70  void getOCFromFile(word *code,word userNr, cParam p);
71  void genCollusion(word *pirates, cParam p);
72  double logB2(double nr);
73  void helpInfo(void);
74  void seedRNG(void);
75  dword getRandomDWord(void);
76  word getRandomWord(word q);
77  void init_genrand(unsigned long s);
78  unsigned long genrand_int32(void);
79  double genrand_real2(void);
80
81  /* End of bonehShaw.h */
```

A.2 The main program

```
1  /* bsMain.c                          */
2  /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4  #include "bonehShaw.h"
5
6  int main(int argc, char *argv[])
7  {
8    int i,c,total,error, flagOC,flagP,flagM,flagT;
9    cParam p;   /* Code parameters */
10   time_t start,end, startOC, endOC;
11
12   (void) time(&start);
13   total=error=flagOC=flagP=flagM=flagT=0;
14
15   while(--argc>0 && (*++argv)[0] == '-')
16     while (c = *++argv[0])
17       switch(c) {
18         case 'o': /* Generate outer code */
19           flagOC=1; break;
20         case 'm': /* Generate hybrid fingerprint */
21           flagM=1; break;
22         case 't': /* Trace hybrid fingerprint */
23           flagM=1; flagT=1; break;
24         case 'p': /* Print */
25           flagP=1; break;
26         case 'h': /* Help info */
27           helpInfo(); return 0;
28       }
29
30   /* Get all code parameters. Store in struct */
31   p = getCodeParam(argc, argv);
32
33   printf("Code parameters: C1=(n1,q)=(%u,%u), C2=(n2,M)_q=(%u,%u)_%u."\
34    "\n\t\t r=%d, t=%u, n=%u\n", p.n1, p.q, p.n2, p.M, p.q, p.r,p.t,p.n1*p.n2);
35
36   /* Set all code properties. Store in struct */
37   p = setProp(p);
38
39   seedRNG();
40
41   /* Run the program a number of times... */
42   for(i=0;i<NOC;i++) {
43     if(flagOC==1) { /* Generate outer code */
44       if(i==0)         (void) time(&startOC);
45       genOuterCode(p);
46       if(i==0) (void) time(&endOC);
47
48       if(flagP==1)  /* Print outer code */
49         printNonBinaryCode(p);
50     }
51
```

```
52      if(flagM==1) { /* Generate hybrid fingerprint */
53        if(p.t < p.M && p.t > 1) /* Check if t is given as argument  */
54                error = genHybridFP(p, flagP, flagT);
55        else{
56          printf("The number of colluding pirates(t) must be given"\
57                 " as input. (M > t > 1)\n");
58          CLE;
59        }
60      }
61
62      total=total+error;
63    }
64
65    if(flagT==1 && p.t>1)
66      printf("Total error: %d\n", total);
67
68    if(flagOC==1)
69      printf("Generate OC running time = %d seconds\n",(int)endOC-(int)startOC);
70    (void) time(&end);
71    printf("Total running time = %d seconds\n",(int)end-(int)start);
72
73    return 0;
74  }
75
76  /* End of bsMain.c */
```

A.3 Set parameters

```
1   /* setCodeParam.c                    */
2   /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4   #include "bonehShaw.h"
5
6   /* Get code parameters from command-line */
7   cParam getCodeParam(int argc,char *argv[])
8   {
9     cParam p;
10
11    if(argc < 4 || argc > 5)
12      CLE; /* Wrong nr of arguments */
13
14    p.n2 = chkIfDigit(argv[0]); /* Outer code - length */
15    p.M = chkIfDigit(argv[1]);  /* Outer code - #codewords */
16    p.q = chkIfDigit(argv[2]);  /* Outer code - alphabet
17                                   Inner code - #codewords */
18    p.r = chkIfDigit(argv[3]);  /* Inner code - replication factor */
19    p.n1 = p.r * (p.q-1);       /* Inner code - length */
20
21    /* Check for collusion-size parameter */
22    p.t = (argc == 5) ? chkIfDigit(argv[4]) : 0;
23
```

```
24    /* ICepsilon and expr are used in the tracing algorithm alg1() */
25    //p.ICepsilon = pow(2,logB2(2*p.q)-(p.r/(2 * pow(p.q,2)))); /* T.V.1 */
26    p.ICepsilon = ((2 * p.q) * pow(2,-(p.r/(2 * pow(p.q,2))))); /* T.V.1 */
27    p.expr = (logB2((2*p.q)/p.ICepsilon)); /* Constant */
28
29    return p;
30  }
31
32  /* Calculate code properties */
33  cParam setProp(cParam p)
34  {
35    int i;
36
37    /* Find nr of bits we need for one nr in the outer code */
38    for(i=0;i<BS_WORD;i++)
39      if ((p.q & (word)1<<i) > 0)
40        p.nrOfBits = i+1;
41
42    /* # numbers we can store in each block(in each unsigned long)
43       of the Outer code */
44    p.nrInBlockOC = BS_WORD/p.nrOfBits;
45
46    /* Set number of blocks in Outer code */
47    p.nrOfBlocksOC = (p.n2 % p.nrInBlockOC) == 0 ?
48                     p.n2/(p.nrInBlockOC) : (p.n2/(p.nrInBlockOC))+1;
49
50    /* Set size of last block in Outer Code  */
51    p.lenLastBlockOC = p.n2-((p.nrOfBlocksOC-1)*p.nrInBlockOC);
52
53    /* Set number of blocks in Inner code */
54    p.nrOfBlocksIC = (p.n1 % BS_DWORD) == 0 ?
55                     p.n1/BS_DWORD : (p.n1/BS_DWORD) + 1;
56
57    /* Set size of last block in Inner Code  */
58    p.lenLastBlockIC = p.n1-(BS_DWORD*(p.nrOfBlocksIC-1));
59
60    return p;
61  }
62
63  /* End of setCodeParam.c */
```

A.4 Generate/read outer code

```
1   /* oc.c                              */
2   /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4   #include "bonehShaw.h"
5
6   /* Generate and store the outer code. */
7   void genOuterCode(cParam p)
8   {
```

```
9     int i,j,k, blockLen;
10    FILE *fp;
11    word *outerCode;
12
13    EOM((outerCode = malloc(p.nrOfBlocksOC * sizeof(word))));
14
15    if ((fp = fopen(TXTFILE, "w")) == NULL)
16      UOF; /* Unable to open the file */
17
18    blockLen = p.nrInBlockOC;
19
20    for(i=0;i<p.M;i++) {  /* For all users */
21      for(j=0;j<p.nrOfBlocksOC;j++) { /* For the nr of blocks we need  */
22        if (j == p.nrOfBlocksOC - 1)  /* If last block */
23          blockLen = p.lenLastBlockOC;
24
25        for(k=0;k<blockLen;k++) { /* For the number of numbers in block */
26          if(k==0)  /* First number in this block */
27            outerCode[j] = getRandomWord((word)p.q) << (BS_WORD-p.nrOfBits);
28          else
29            outerCode[j] ^=
30              (getRandomWord((word)p.q) << (BS_WORD-((k+1)*p.nrOfBits)));
31        }
32      }
33      blockLen = p.nrInBlockOC;
34      fwrite(outerCode, p.nrOfBlocksOC, sizeof(word),fp);
35
36      if(i==0)
37        if((fp = freopen(TXTFILE,"a",fp)) == NULL)
38          UOF; /* Unable to reopen the file */
39    }
40    printf("Outercode generated and stored\n");
41
42    fclose(fp);
43    free(outerCode);
44  }
45
46  /* Extract the numbers from a particular user's outer code */
47  void getOC(int userNr, word *numbers, cParam p)
48  {
49    int i,j,k,t,lenBlock;
50    word mask, *outerCode;
51
52    lenBlock = p.nrInBlockOC;
53    t = 0;
54
55    EOM((outerCode = malloc(p.nrOfBlocksOC * sizeof(word))));
56
57    /* 'mask' is used to mask out the current bits. First we
58       set 'mask' such that we can get the first number */
59    mask = ULONG_MAX << (BS_WORD-p.nrOfBits);
60
61    getOCFromFile(outerCode, userNr, p);
62
```

```
63    for(j=0;j<p.nrOfBlocksOC;j++) { /* For all blocks */
64      if (j==p.nrOfBlocksOC-1)
65        lenBlock = p.lenLastBlockOC;
66
67      /* For all numbers in the current block */
68      for(k=0;k<lenBlock;k++,t++)
69        numbers[t] =(outerCode[j] &
70           (mask >>(p.nrOfBits*k))) >> (BS_WORD-(p.nrOfBits*(k+1)));
71    }
72
73    free(outerCode);
74  }
75
76  /* Get the outer codeword for a particular user */
77  void getOCFromFile(word *code, word userNr, cParam p)
78  {
79    FILE *fp;
80    fp = fopen(TXTFILE,"r");
81
82    fseek(fp, userNr*p.nrOfBlocksOC*sizeof(word),SEEK_SET);
83    fread(code, p.nrOfBlocksOC, sizeof(word),fp);
84
85    fclose(fp);
86  }
87
88  /* End of oc.c */
```

A.5 Make hybrid fingerprint

```
1   /* makeHybrid.c                       */
2   /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4   #include "bonehShaw.h"
5
6   /* Make hybrid fingerprint */
7   int genHybridFP(cParam p, int flagP, int flagT)
8   {
9     int i,j,k,l,m,error;
10    dword *feasible, *hybridFP, **pirateCol, *arrayptr;
11    word **pirates, *arrayptr2;
12    long **everyPos,*arrayptr1;
13    word **pirOC, *arrayptr3;
14
15    error=0;
16
17    EOM((feasible = malloc(p.nrOfBlocksIC * sizeof(dword))));
18    EOM((hybridFP = malloc(p.nrOfBlocksIC * sizeof(dword))));
19
20    /* Make pirate codebook -> Allocate memory for the array */
21    EOM((arrayptr = malloc(p.t * p.nrOfBlocksIC  * sizeof(dword))));
22    /* Allocate room for the pointers to the rows */
```

```
23    EOM((pirateCol = malloc(p.t * sizeof(dword *))));
24    /* Make array to store pirate outer codes */
25    EOM((arrayptr3 = malloc(p.t * p.n2 * sizeof(word))));
26    EOM((pirOC = malloc(p.t * sizeof(word *))));
27    /* and now we 'point' the pointers */
28    for (i=0;i<p.t;i++) {
29      pirateCol[i] = arrayptr + (i * p.nrOfBlocksIC);
30      pirOC[i] = arrayptr3 + (i * p.n2);
31    }
32
33    /* Store the outer code numbers returned from alg1...for all the pirColls */
34    EOM((arrayptr1 = malloc(NPC * p.n2  * sizeof(long))));
35    EOM((everyPos = malloc(NPC * sizeof(long))));
36    /* Make pirate array */
37    EOM((arrayptr2 = malloc(NPC * p.t  * sizeof(word))));
38    EOM((pirates = malloc(NPC * sizeof(word))));
39    for (i=0;i<NPC;i++) {
40      pirates[i] = arrayptr2 + (i * (p.t));
41      everyPos[i] = arrayptr1 + (i * (p.n2));
42    }
43
44    /* Create a collusion -> make part of hybridfp -> trace part */
45    for(i=0;i<NPC;i++) {
46      genCollusion(pirates[i],p);
47
48      for(j=0;j<p.t;j++) /* Get pirate outer code */
49        getOC(pirates[i][j],pirOC[j],p);
50
51      /* For all digits in the outer code */
52      for(j=0;j<p.n2;j++) {
53        /* Get one of the inner codeblocks for all pirates */
54        for(k=0;k<p.t;k++) {
55          getPartOfCC(pirateCol[k], pirOC[k],p, k, j);
56
57          if(flagP==1) {
58            printf("\nUser %d IC corresponds to nr in position %d in OC "
59                   ,pirates[i][k], j);
60            printBinaryCode(pirateCol[k], p);
61          }
62        }
63
64        findFeasible(feasible, pirateCol, p, flagP);
65        makeHybrid(hybridFP,pirateCol,feasible,p, flagP);
66
67        if(flagT==1)  /* If tracing is chosen -> inner decoding */
68          /* Find and return the outer code nr from Algorithm1.*/
69          everyPos[i][j] = alg1(hybridFP,p, flagP);
70      }
71    }
72
73    if(flagT==1))  /* If tracing is chosen -> outer decoding */
74      error = trace(everyPos, pirates, p, flagP);
75
76    free(everyPos);
```

```
77    free(arrayptr);
78    free(arrayptr1);
79    free(pirateCol);
80    free(feasible);
81    free(hybridFP);
82    free(pirates);
83
84    return error;
85  }
86
87  /* Generate a random pirate collusion of size t */
88  void genCollusion(word *pirates, cParam p)
89  {
90    int i,j;
91    word randnum;
92
93    for(i=0;i<p.t;i++) {
94      pirates[i] = getRandomWord(p.M); /* Select pirate in the range: [0..M-1] */
95      /* Test to ensure that all pirates in the collusion are unique */
96      for(j=0;j<i;j++)
97        if(pirates[j] == pirates[i])
98          i=j=i-1;
99    }
100 }
101
102 /* Returns one of the inner codewords that
103 make up the concatenated code for a user  */
104 void getPartOfCC(dword *innerCode, word *outerCode,
105                                  cParam p, word userNr, int OCpos)
106 {
107   int i, nrOfNull,changeBlocks, lastBlock;
108
109   /* Set innerCode according to the number in OC */
110   nrOfNull = p.r * outerCode[OCpos];
111
112   /* How many blocks must be changed  */
113   changeBlocks = (nrOfNull % BS_DWORD) == 0 ?
114                nrOfNull/BS_DWORD : (nrOfNull/BS_DWORD) + 1;
115
116   for(i=changeBlocks;i<p.nrOfBlocksIC;i++)
117     innerCode[i] = ULLONG_MAX;
118
119   lastBlock =  nrOfNull - (BS_DWORD * (changeBlocks -1));
120
121   for(i=0;i<changeBlocks;i++) {
122     if (i==changeBlocks-1)
123       innerCode[i] = ULLONG_MAX^(ULLONG_MAX<<(BS_DWORD-lastBlock));
124     else
125       innerCode[i] = (dword)0;
126   }
127 }
128
129 /* Find feasible set */
130 void findFeasible(dword *feasible, dword **pirateCol, cParam p, int flagP)
```

```
131 {
132   int i, j, k;
133
134   if(flagP==1)printf("\nFinding feasible set for this inner code...");
135   for(i=0;i<p.t;i++)  /* for every codeword...(part of concat code)*/
136   /* and for every part of that inner codeword */
137     for(j=0;j<p.nrOfBlocksIC;j++)
138       for(k=i+1;k<p.t;k++) { /*...compare against the rest */
139         if (k == i+1 && i == 0) /* No value in feasible[j] */
140           feasible[j] = pirateCol[i][j] ^ pirateCol[k][j];
141         else
142           feasible[j] |= pirateCol[i][j] ^ pirateCol[k][j];
143       }
144
145   if(flagP==1) {
146     printf("Feasible set found!\n");
147     printf("Feasible: ");
148     printBinaryCode(feasible, p);
149     printf("\nCreating part of hybrid fingerprint...");
150   }
151 }
152
153 /* Make part of hybrid fingerprint */
154 void makeHybrid(dword *hybridFP,dword **pirateCol,
155                                 dword *feasible, cParam p, int flagP)
156 {
157   int i;
158
159   for(i=0;i<p.nrOfBlocksIC;i++) { /* For all blocks */
160     /* Generate random block(Part of the hybrid fingerprint) */
161     hybridFP[i] = getRandomDWord();
162
163     /* Check if hybrid fingerprint is valid -> If not, correct it */
164     hybridFP[i] =
165             ((pirateCol[0][i] ^ hybridFP[i]) & (~feasible[i])) ^ hybridFP[i];
166   }
167
168   if(flagP==1) {
169     printf("Part of hybrid fingerprint created!\n");
170     printf("HybridFP: ");
171     printBinaryCode(hybridFP, p);
172   }
173 }
174
175 /* End of makeHybrid.c */
```

A.6 Trace hybrid fingerprint

```
1 /* traceHybrid.c                    */
2 /* 2005: Tor Røneid <torr@ii.uib.no> */
3
```

```
4  #include "bonehShaw.h"
5
6  /* Find 'guilty' users, and check if they really are guilty */
7  int trace(long **everyPos, word **pirates, cParam p, int flagP)
8  {
9    int i,j, error, pirFound, *guiltyUsers;
10
11   error=pirFound=0;
12   EOM((guiltyUsers = malloc(NPC * sizeof(int))));
13
14   /* Find guilty users for all collusions by using CND */
15   closestNDec(everyPos,guiltyUsers,p, flagP);
16
17   /* Check if the 'guilty' users returned really are guilty */
18   for(i=0;i<NPC;i++) {
19     for(j=0;j<p.t;j++)
20       if (guiltyUsers[i] == pirates[i][j])
21         pirFound = 1;
22
23     if(pirFound==0){
24       error++;
25     pirFound=0;
26   }
27
28   free(guiltyUsers);
29   return error;
30 }
31
32 /* This is the decoding algorithm for the inner code in the BS-CS scheme.
33    We apply alg1 to each of the n_2 components of the HybridFP  */
34 long alg1(dword *hybridFPblock, cParam p, int flagP)
35 {
36   int s,w,k,flag;
37   long pos;
38
39   if(flagP==1) printf("ICepsilon:%0.15f\n",p.ICepsilon);
40   flag = 1;
41
42   /* Check condition 1 of alg 1. Check d first positions */
43   w = weight(hybridFPblock, p, 1);
44   if(flagP==1) printf("first cond. weight = %u\n", w);
45   if (w > 0)
46     pos = 0;
47   else {
48     /* Check condition 2 of alg 1. Check d last positions */
49     w = weight(hybridFPblock, p, p.q-1);
50     if(flagP==1) printf("second cond. weight = %u\n", w);
51     if (w < p.r)
52       pos = p.q-1;
53     else {
54       /* Check condition 3 of alg 1. */
55       for(s=2; s<p.q;s++) {
56         k = weight(hybridFPblock, p, s);    //B_s
57         w = weight(hybridFPblock, p, s-1);  //B_(s-1)
```

```
58          k = k + w;                              //B_s U B_(s-1)
59
60          if(flagP==1) printf("third cond. weigth. k = %u\tw = %d\n", k, w);
61          if (w < (double)k/2 - sqrt(((double)k/2 * p.expr))) {
62            pos = s-1;
63            flag = 0;
64            s = p.q;
65          }
66        }
67      }
68    }
69    return pos;
70  }
71
72  /* Calculate the weight of a particular pattern(r length) in hybridFP */
73  word weight(dword *hybridBlock, cParam p, int s)
74  {
75    int i,r, pos, startBlock, nrOfNrBlock, check, length, count;
76    word w;
77    dword mask;
78
79    w = 0;
80    count = mask = 1;
81    r = p.r;
82    pos = p.r * s;
83
84    while(r>0) {
85      startBlock = (pos % BS_DWORD) == 0 ?
86                   (pos / BS_DWORD) - 1 : pos / BS_DWORD;
87
88      nrOfNrBlock = pos - (BS_DWORD * startBlock);
89
90      /* Set number of positions to check */
91      check = (r >= nrOfNrBlock) ? nrOfNrBlock : r;
92
93      /* If count equals 2, then we must read block from
94         rightmost position. If count equals 1
95         (first block we look at), then we must set the
96         correct position to read from */
97      if(count == 1) {
98        count++;
99        length=(r<nrOfNrBlock)?BS_DWORD - nrOfNrBlock:BS_DWORD - check;
100     }
101     else
102       length = 0;
103
104     for(i=0;i<check;i++)
105       if ((hybridBlock[startBlock]  &   (mask << length + i)) != 0)
106         w++;
107
108     pos = pos - check;
109     r = r - check;
110   }
111
```

```
112   return w;
113 }
114
115 /* Find 'guilty' users for all collusions, by using closest neighbour decoding */
116 void closestNDec(long **everyPos, int *guiltyUsers, cParam p, int flagP)
117 {
118   int i,j,k,counter;
119   int *prev;
120   word *outercode;
121
122   EOM((outercode = malloc(p.n2 * sizeof(word))));
123   EOM((prev = malloc(NPC * sizeof(int))));
124
125   if(flagP==1) {
126     printf("\n\nOuter codeword found: ");
127     for(i=0;i<NPC;i++){
128       guiltyUsers[i] = -1;
129       prev[i] = 0;
130
131       for(j=0;j<p.n2;j++)
132         printf("%d", everyPos[i][j]);
133     }
134     printf("\n\nCompare it with user codewords...\n");
135   }
136   else
137     for(i=0;i<NPC;i++){
138       guiltyUsers[i] = -1;
139       prev[i] = 0;
140     }
141
142   counter = 0;
143
144   for(i=0;i<p.M;i++) {  /* For all users */
145     /* Get digits from the outerCode (For the current user) */
146     getOC(i, outercode, p);
147
148     /* Check how many positions that match */
149     for(j=0;j<NPC;j++) {  /* For all collusions */
150       for(k=0; k<p.n2; k++)   /* For all positions in outer code */
151         if (everyPos[j][k] == outercode[k])
152           counter++;
153
154       /* If this user codeword has more positions that match, store this user */
155       if (counter > prev[j]) {
156         guiltyUsers[j] = i;
157         prev[j] = counter;
158       }
159       counter = 0;
160     }
161   }
162
163   free(outercode);
164   free(prev);
165 }
```

```
166
167 /* End of traceHybrid.c */
```

A.7 The RNG handler

```
1  /* rng.c                              */
2  /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4  #include "bonehShaw.h"
5
6  void seedRNG(void)
7  {
8    init_genrand((unsigned)time(NULL));
9  }
10
11 /* Generates a 32 bit number in the range [0..q-1] */
12 word getRandomWord(word q)
13 {
14   return (genrand_real2() * q);
15 }
16
17 /* Generates a random 64 bit number */
18 dword getRandomDWord(void)
19 {
20   dword x;
21
22   /* Generate two 32 bit numbers. Concatenate to one 64 bit */
23   x = ((dword)genrand_int32() << 32) + (dword)genrand_int32();
24   return x;
25 }
26
27 /* End of rng.c */
```

A.8 Mersenne Twister RNG

```
1  /* mt19937.c */
2  /*
3     A C-program for MT19937, with initialization improved 2002/1/26.
4     Coded by Takuji Nishimura and Makoto Matsumoto.
5
6     Before using, initialize the state by using init_genrand(seed)
7     or init_by_array(init_key, key_length).
8
9     Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
10    All rights reserved.
11
12    Redistribution and use in source and binary forms, with or without
13    modification, are permitted provided that the following conditions
14    are met:
15
```

```
16        1. Redistributions of source code must retain the above copyright
17           notice, this list of conditions and the following disclaimer.
18
19        2. Redistributions in binary form must reproduce the above copyright
20           notice, this list of conditions and the following disclaimer in the
21           documentation and/or other materials provided with the distribution.
22
23        3. The names of its contributors may not be used to endorse or promote
24           products derived from this software without specific prior written
25           permission.
26
27      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
28      "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
29      LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
30      A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
31      CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
32      EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
33      PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
34      PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
35      LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
36      NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
37      SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
38
39
40      Any feedback is very welcome.
41      http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
42      email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
43  */
44
45  #include <stdio.h>
46
47  /* Period parameters */
48  #define N 624
49  #define M 397
50  #define MATRIX_A 0x9908b0dfUL   /* constant vector a */
51  #define UPPER_MASK 0x80000000UL /* most significant w-r bits */
52  #define LOWER_MASK 0x7fffffffUL /* least significant r bits */
53
54  static unsigned long mt[N]; /* the array for the state vector  */
55  static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
56
57  /* initializes mt[N] with a seed */
58  void init_genrand(unsigned long s)
59  {
60      mt[0]= s & 0xffffffffUL;
61      for (mti=1; mti<N; mti++) {
62          mt[mti] =
63              (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
64          /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
65          /* In the previous versions, MSBs of the seed affect   */
66          /* only MSBs of the array mt[].                        */
67          /* 2002/01/09 modified by Makoto Matsumoto             */
68          mt[mti] &= 0xffffffffUL;
69          /* for >32 bit machines */
```

```
70      }
71  }
72
73  /* generates a random number on [0,0xffffffff]-interval */
74  unsigned long genrand_int32(void)
75  {
76      unsigned long y;
77      static unsigned long mag01[2]={0x0UL, MATRIX_A};
78      /* mag01[x] = x * MATRIX_A  for x=0,1 */
79
80      if (mti >= N) { /* generate N words at one time */
81          int kk;
82
83          if (mti == N+1)   /* if init_genrand() has not been called, */
84              init_genrand(5489UL); /* a default initial seed is used */
85
86          for (kk=0;kk<N-M;kk++) {
87              y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
88              mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
89          }
90          for (;kk<N-1;kk++) {
91              y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
92              mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
93          }
94          y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
95          mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
96
97          mti = 0;
98      }
99
100     y = mt[mti++];
101
102     /* Tempering */
103     y ^= (y >> 11);
104     y ^= (y << 7) & 0x9d2c5680UL;
105     y ^= (y << 15) & 0xefc60000UL;
106     y ^= (y >> 18);
107
108     return y;
109 }
110
111 /* These real versions are due to Isaku Wada, 2002/01/09 added */
112 /* generates a random number on [0,0x7fffffff]-interval */
113 long genrand_int31(void)
114 {
115     return (long)(genrand_int32()>>1);
116 }
117
118 /* generates a random number on [0,1]-real-interval */
119 double genrand_real1(void)
120 {
121     return genrand_int32()*(1.0/4294967295.0);
122     /* divided by 2^32-1 */
123 }
```

```
124
125 /* generates a random number on [0,1)-real-interval */
126 double genrand_real2(void)
127 {
128     return genrand_int32()*(1.0/4294967296.0);
129     /* divided by 2^32 */
130 }
131
132 /* generates a random number on (0,1)-real-interval */
133 double genrand_real3(void)
134 {
135     return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
136     /* divided by 2^32 */
137 }
138
139 /* generates a random number on [0,1) with 53-bit resolution*/
140 double genrand_res53(void)
141 {
142     unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
143     return(a*67108864.0+b)*(1.0/9007199254740992.0);
144 }
145
146 /* End of mt19937.c */
```

A.9 Debugging functions

```
1  /* printFunc.c                        */
2  /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4  #include "bonehShaw.h"
5
6  /* Print Outer code (Non-binary) */
7  void printNonBinaryCode(cParam p)
8  {
9    int i,j,k, blockLen;
10   word mask, *code;
11   FILE *fp;
12
13   if ((fp = fopen(TXTFILE,"r")) == NULL)
14     UOF; /* Unable to open the file */
15
16   EOM((code = malloc(p.nrOfBlocksOC * sizeof(word))));
17
18   blockLen = p.nrInBlockOC;
19
20   /* 'mask' is used to mask out the current bits. First we
21      set 'mask' such that we get the first number */
22   mask = ULONG_MAX << (BS_WORD-p.nrOfBits);
23
24   for(i=0;i<p.M;i++) {  /* For all users */
25     printf("Codeword %i = ",i+1);
```

```
26      fread(code, p.nrOfBlocksOC, sizeof(word),fp);
27
28      for(j=0;j<p.nrOfBlocksOC;j++) { /* For all blocks */
29        if (j==p.nrOfBlocksOC-1)
30          blockLen = p.lenLastBlockOC;
31
32        for(k=0;k<blockLen;k++)  /* For all numbers in the current block */
33          printf("%u",(code[j] &
34              (mask >>(p.nrOfBits*k))) >> (BS_WORD-(p.nrOfBits*(k+1))));
35      }
36      printf("\n");
37      blockLen = p.nrInBlockOC;
38    }
39
40    fclose(fp);
41    free(code);
42  }
43
44  /* Print binary code */
45  void printBinaryCode(dword *innerCode, cParam p)
46  {
47    int i,k,j,t,bits, len, count;
48    char *str;
49
50    EOM((str = malloc(p.n1)));
51
52    k=0;
53    len = BS_DWORD;
54
55    for(i=0;i<p.nrOfBlocksIC;i++) {
56      if(i==p.nrOfBlocksIC-1)
57        len=p.lenLastBlockIC;
58
59      for(j=0;j<len; j++,k++)
60        str[k] = innerCode[i]&((dword)1<<(BS_DWORD-1-j))?'1':'0';
61    }
62
63    str[k] = '\0';
64
65    printf("%s - Length %i\n", str, strlen(str));
66
67    free(str);
68  }
69
70  /* End of printFunc.c */
```

A.10 Other functions

```
1   /* otherFunc.c                        */
2   /* 2005: Tor Røneid <torr@ii.uib.no> */
3
4   #include "bonehShaw.h"
```

```
5
6  /* Check command-line input. Must be a digit */
7  int chkIfDigit(char argv[])
8  {
9    int i;
10
11   for(i=0;i<strlen(argv);i++)
12     if(!isdigit(argv[i])) {
13       printf("n2, M, q, r and t(optional) must be numbers.\n");
14       CLE;
15     }
16
17   return atoi(argv);
18 }
19
20 /* Calculate base 2 logarithm */
21 double logB2(double nr)
22 {
23   return log10(nr)/log10(2);
24 }
25
26 void helpInfo(void)
27 {
28   printf("Read the manual for help\n");
29 }
30
31 /* End of otherFunc.c */
```

APPENDIX A. THE SOURCE CODE

Bibliography

[AP98] R.J. Anderson and F.A.P. Petitcolas. On the limits of steganography. *IEEE Journal on Selected Areas in Communications, Special Issue on Copyright and Privacy Protection*, 16(4):474–481, may 1998.

[BBK03] A. Barg, G.R. Blakly, and G.A. Kabatiansky. Digital fingerprinting codes: Problem statements, constructions, identifications of traitors. *IEEE Trans. Inform. Theory*, 49(4):852–865, apr 2003.

[BMP85] G.R. Blakley, C. Meadows, and G.B. Purdy. Fingerprinting long forgiving messages. *Proceedings of Crypto '85, Springer-Verlag*, pages 180–189, 1985.

[BS95] D. Boneh and J. Shaw. Collusion-secure fingerprinting for digital data. *Advances in Cryptology: Proceedings of Crypto '95, Springer-Verlag*, pages 452–465, 1995.

[BS98] D. Boneh and J. Shaw. Collusion-secure fingerprinting for digital data. *IEEE Trans. Inform. Theory*, 44(5):1897–1905, sep 1998.

[CFN94] B. Chor, A. Fiat, and M. Naor. Tracing traitors. *In Advances in Cryptology - CRYPTO '94*, 839 of Springer Lecture Notes in Computer Science:257–270, 1994. Springer-Verlag.

[Che96] Y. M. Chee. Turàn-type problems in group testing, coding theory and cryptography. *PhD thesis, University of Waterloo, Canada*, 1996.

[HJDF00] J. Herrera-Joancomarti and J. Domingo-Ferrer. Short collusion-secure fingerprints based on dual binary hamming codes. *Electronic Letters*, 36:1697–1699, sep 2000.

[HK99] F. Hartung and M. Kutter. Multimedia watermarking techniques. *Proceedings of the IEEE*, 87(7):1079–1107, jul 1999.

[LBH03] T.V. Le, M. Burmester, and J. Hu. Short c-secure fingerprinting codes. *In Proceedings of the 6th Information Security Conference*, oct 2003.

[LL00a] J. Löfvenberg and T. Lindkvist. A general description of pirate strategies in a binary fingerprinting system. *Report LiTH-ISY-R-2259*, 2000.

[LL00b] T. Lindkvist and J. Löfvenberg. Some simple pirate strategies. *Report LiTH-ISY-R-2258*, 2000.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. jan 1998.

[MS77] F.J. MacWilliams and N.J.A. Sloane. The theory of error-correcting codes. 1977.

[Mur04] H. Muratani. Optimization and evaluation of randomized c-secure crt code defined on polynomial ring. *Information Hiding 2004*, 3200:282–292, 2004.

[PS96] B. Pfitzmann and M. Schunter. Asymmetric fingerprinting. *Advances in Cryptology - EUROCRYPT '96, Springer-Verlag*, pages 85–95, 1996.

[PS99] B. Pfitzmann and A.R. Sadeghi. Coin-based anonymous fingerprinting. *Advances in Cryptology - EUROCRYPT '99, Springer-Verlag*, pages 150–164, 1999.

[PW97] B. Pfitzmann and M. Waidner. Anonymous fingerprinting. *Advances in Cryptology - EUROCRYPT '97, Springer-Verlag*, pages 88–102, 1997.

[Sag94] Y. L. Sagalovich. Separating systems. *Problems of Information Transmission*, 30(2):105–123, 1994.

[Sch03a] H.G. Schaathun. The boneh-shaw fingerprinting scheme is better than we thought. *Technical Report 256, Department of Informatics, University of Bergen*, nov 2003.

[Sch03b] H.G. Schaathun. Fighting two pirates. *In Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 2643 of Springer Lecture Notes in Computer Science:71–79, may 2003.

[Sch04a] H.G. Schaathun. Binary collusion-secure codes: Comparison and improvements. *Technical Report 275, Department of Informatics, University of Bergen*, 2004.

[Sch04b] H.G. Schaathun. Fighting three pirates with scattering codes. *Technical Report 263, Department of Informatics, University of Bergen*, jan 2004.

[SDF02a] F. Sebé and J. Domingo-Ferrer. Scattering codes to implement short 3-secure fingerprinting for copyright protection. *Electronic Letters*, 38:958–959, aug 2002.

[SDF02b] F. Sebé and J. Domingo-Ferrer. Short 3-secure fingerprinting codes for copyright protection. *In ACISP 2002*, volume 2384 of Springer Lecture Notes in Computer Science:316–327, 2002.

[SFM05] H.G. Schaathun and M. Fernandez-Muñoz. Boneh-shaw fingerprinting and soft decission decoding. *In Information Theory Workshop*, sep 2005. Rotorua, NZ.

[Tar03] G. Tardos. Optimal probabilistic fingerprinting codes. *In Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, 2003.

[Wag83] N.R. Wagner. Fingerprinting. *In Proceedings of the 1983 Symposium on Security and Privacy*, 1983.

[YII98] J. Yoshida, K. Iwamura, and H. Imai. A coding method for collusion-secure watermark and less decline. *In:SCIS'98*, Number 10.2A, 1998.