CMPUT690 Term Project
Fingerprinting using Polynomial (Rabin's method)

Calvin Chan Hahua Lu

December 4, 2001

Executive Summary

The purpose of this project is to implement the Rabin's method of fingerprinting using irreducible polynomials. This project team has successfully implemented the method and test run on a given dataset. Test results show that the time efficiency of this method is comparable to the other well known hashing functions while outperfoming them in the sense of lower or even no collision occurences.

Table of Contents

Chapter 1

Introduction

1.1 Reference

1.1.1 Terms of Reference

This term project is carried out with the purpose of fulfilling the course requirement of CMPUT690.

CMPUT690 is a graduate course in Advanced Database Systems offered by the Department of Computing Science, Faculty of Science, University of Alberta. The instructor of the course is Professor Davood Rafiei.

1.1.2 Project Team Members

The project team consists of the following team members:

- Calvin Chan
- Hai-Hua Lu

1.2 Motivation

Traditionally, a hashing function is used to map a character string of undetermined length to a storage address. The mapping is done by computing the residue of that string, viewed as a large integer, modulo p - a prime number[1].

Since this is a mapping from an unbounded domain into a finite range of integers, it would be very difficult to devise a good hashing function that will produce a set of distinct mapped values with minimal chance of collisions. This is especially true when the set of character strings to be mapped is of a huge size.

In 1981, Michael O. Rabin published a paper [1] descibing a fingerprinting method using polynomials. It is claimed that the method provides efficient mapping that has little chance of collision even with huge dataset.

1.3 Objectives

In this project, it is intended to achieve the followings:

- to implement the Rabin's method and to test it against a dataset
- to examine the relationship between the percentage of collisions (distinct strings with the same fingerprint) and the degree of the chosen irreducible polynomial
- to compare the effectiveness of using Rabin's method with that of using hashing functions

Chapter 2

Rabin's method

2.1 Method Outline

2.1.1 The Scheme

Suppose the character string $\mathcal{A}$ is a bit string containing m bits $[b_1, \ldots, b_m]$. It is then associated to a polynomial of degree $(m-1)$ in indeterminate t as follows.

$$\mathcal{A}(t) = b_1 t^{m-1} + b_2 t^{m-2} + \ldots + b_{m-1} t + b_m$$

Given a polynomial $\mathcal{P}(t)$ of degree k,

$$\mathcal{P}(t) = a_1 t^k + a_2 t^{k-1} + \ldots + a_{k-1} t + a_k$$

the residue $f(t) = \mathcal{A}(t) \bmod \mathcal{P}(t)$ will be of degree $(k-1)$.

In Rabin's fingerprinting method, an irreducible polynomial is used for $\mathcal{P}(t)$. More details about irreducible polynomials can be found in section 2.2. Since we are dealing with bit strings, all the coefficients of $\mathcal{A}(t)$ are in $\mathbf{Z_2}$. Thus $\mathcal{P}(t)$ will be chosen using a_i's in $\mathbf{Z_2}$. With this in mind, the fingerprinting function is defined as follows:

Definition Given a character string $\mathcal{A}$, the fingerprint of $\mathcal{A}$ is

$$f(\mathcal{A}) = \mathcal{A}(t) \bmod \mathcal{P}(t) \tag{1}$$

2.1.2 Bound of Error

In his paper [1], Rabin finds a bound for error in the steps as follows:

1. The number of irreducible polynomial $\mathcal{P}(t)$ of degree k is
$$\frac{2^k - 2}{k} \approx \frac{2^k}{k}$$
.

2. Given a dataset $\mathcal{S}$ containing n character strings of maximum length m bits, construct a polynomial
$$\mathcal{Q}(t) = \prod_{\{\mathcal{A},\mathcal{B}\in\mathcal{S}\}} (\mathcal{A}(t) - \mathcal{B}(t))$$
.

3. If degree of $\mathcal{Q}(t) = degQ$, then
$$\begin{aligned} degQ &= \sum_{\{\mathcal{A},\mathcal{B}\in\mathcal{S}\}} deg\ (\mathcal{A}(t) - \mathcal{B}(t)) \\ &\leq \sum_{\{\mathcal{A},\mathcal{B}\in\mathcal{S}\}} max\{deg\ \mathcal{A}(t), deg\ \mathcal{B}(t)\} \\ &\leq \sum_{\{\mathcal{A},\mathcal{B}\in\mathcal{S}\}} m \\ &\leq n^2 m \end{aligned}$$

4. Maximum number of irreducible factors of degree k of $\mathcal{Q}(t) = \frac{degQ}{k} \leq \frac{n^2 m}{k}$.

5. For $f(\mathcal{A}) = f(\mathcal{B})$ given $\mathcal{A} \neq \mathcal{B}$, one must have $\mathcal{P}(t) \mid (\mathcal{A}(t) - \mathcal{B}(t))$ and $\mathcal{P}(t) \mid \mathcal{Q}(t)$.

6. Probability of making an error = probability of picking a factor of $\mathcal{Q}(t)$

7. The bound of error is estimated as
$$\begin{aligned} Pr\{f(\mathcal{A}) = f(\mathcal{B}) \mid \mathcal{A} \neq \mathcal{B}\} &= \frac{degQ}{k} \Big/ \frac{2^k - 2}{k} \\ &\approx \frac{degQ}{2^k} \\ &\leq \frac{n^2 m}{2^k} \end{aligned}$$

Rabin [1] suggests two ways to lower the probability of error:

- The probability of a wrong output will be lowered by increasing the value of k. This will require a larger word-length.
- The probability can also be lowered by using two different irreducible polynomials $\mathcal{P}_1(t)$ and $\mathcal{P}_2(t)$ of the same degree k. The algorithm is then run twice by interleaving steps, one time with $\mathcal{P}_1(t)$ and another time with $\mathcal{P}_2(t)$. Since the error probabilities are independent, the maximum probability of collision becomes
$$\frac{n^2 m}{2^{2k}}.$$

2.1.3 Properties and Characteristics

The general features are specialized for this project (domain in $\mathbf{Z_2}$) as follows[2]:

1. $\mathcal{A}(t) + \mathcal{B}(t) \equiv \mathcal{A}$ **XOR** $\mathcal{B}$
2. $\mathcal{A}(t)t \equiv \mathcal{A} << 1$
   proof

$$\begin{aligned} \text{If} \quad & \mathcal{A}(t) = a_1 t^m + a_2 t^{m-1} + \ldots + a_{m-1}t + a_m, \\ \text{then} \quad & \mathcal{A}(t)t = (a_1 t^l + a_2 t^{l-1} + \ldots + a_{l-1}t + a_l)t \\ & \quad = a_1 t^{l+1} + a_2 t^l + \ldots + a_{l-1}t^2 + a_l t + 0 \end{aligned}$$

3. $p_1 t^k \bmod \mathcal{P}(t)$ is equivalent to dropping the leading bit off $\mathcal{P}(t)$
   proof

$$\begin{aligned} \text{If } p_1 = 0, \text{ then} \quad & p_1 t^k \bmod \mathcal{P}(t) = 0 \\ \text{If } p_1 = 1 \text{ and} \quad & \mathcal{P}(t) = p_1 t^k + p_2 t^{k-1} + \ldots + p_{k-1}t + p_k, \\ \text{then} \quad & p_1 t^k = \mathcal{P}(t) - p_2 t^{k-1} - \ldots - p_{k-1}t - p_k \\ & p_1 t^k \bmod \mathcal{P}(t) = -p_2 t^{k-1} - \ldots - p_{k-1}t - p_k \\ & \quad = p_2 t^{k-1} + \ldots + p_{k-1}t + p_k = \mathcal{P}(t) - p_1 t^k \end{aligned}$$

4. $f(\mathcal{A} + \mathcal{B}) = f(\mathcal{A}) + f(\mathcal{B})$
5. $f(concat(\mathcal{A}, \mathcal{B})) = f(concat(f(\mathcal{A}), \mathcal{B}))$
6. If $\mathcal{B}$ is of length l, then

$$\begin{aligned} f(concat(\mathcal{A}, \mathcal{B})) &= \{\mathcal{A}(t)t^l + \mathcal{B}(t)\} \bmod \mathcal{P}(t) \\ &= f(f(\mathcal{A}) * f(t^l)) + f(\mathcal{B}) \end{aligned}$$

7. 1-bit extension

$$\begin{aligned} \text{If} \quad & f([a_1, \ldots, a_l]) = (a_1 t^l + a_2 t^{l-1} + \ldots + a_{l-1}t + a_l) \bmod \mathcal{P}(t) \\ & \quad = r_1 t^{k-1} + r_2 t^{k-2} + \ldots + r_{k-1}t + r_k \\ \text{then} \quad & f([a_1, \ldots, a_{l+1}]) = (a_1 t^{l+1} + a_2 t^l + \ldots + a_{l-1}t^2 + a_l t + a_{l+1}) \bmod \mathcal{P}(t) \\ & \quad = (t(a_1 t^l + a_2 t^{l-1} + \ldots + a_{l-1}t + a_l) + a_{l+1}) \bmod \mathcal{P}(t) \\ & \quad = t(a_1 t^l + a_2 t^{l-1} + \ldots + a_{l-1}t + a_l) \bmod \mathcal{P}(t) + a_{l+1} \bmod \mathcal{P}(t) \\ & \quad = (r_1 t^k + r_2 t^{k-1} + \ldots + r_{k-1}t^2 + r_k t + a_{l+1}) \bmod \mathcal{P}(t) \\ & \quad = \underbrace{\underbrace{r_2 t^{k-1} + \ldots + r_{k-1}t^2 + r_k t + a_{l+1}}_{(f([a_1,\ldots,a_l])<<1),\ \text{input } a_{l+1}} + \underbrace{r_1 t^k \bmod \mathcal{P}(t)}_{\text{drop leading bit of } \mathcal{P}(t) \text{ conditioned on } r_1}}_{\mathbf{XOR}} \end{aligned}$$

2.2 Irreducible Polynomials

Irreducible polynomial is a topic in the "finite fields" area and is not within the scope of this project. However, it would definitely help if a short note about irreducible polynomials is included in this chapter. A full discussion is too broad to be included so only a simplified version is included.

2.2.1 Criteria

From literatures[3, 4] of related topics, one can identify the following characteristics of irreducible polynomials in $\mathbf{Z_2}$:

- The Galois Field $Z_2 = \{0, 1\}$.
- Given $\mathcal{P}(t) = a_1 t^k + a_2 t^{k-1} + \ldots + a_{k-1} t + a_k$ in $\mathbf{Z_2}$ of degree k, the following holds:
$$\begin{cases} a_1 = 1 \\ a_i \in Z_2, \forall\, i \in \{1, \ldots, k\} \end{cases}$$
- All polynomials $\mathcal{P}(t)$ with no constant term must be factorable.
- $\mathcal{P}(t)$ is irreducible if $\mathcal{P}(t)$ does not evaluate to 0 for all elements of $\mathbf{Z_2}$.

For
$$\mathcal{P}(t) = a_1 t^k + a_2 t^{k-1} + \ldots + a_{k-1} t + a_k$$
to be irreducible,one can conclude the following criteria based on the above characteristics:

1. $a_i \in \mathbf{Z_2}, \forall\, i \in \{1, \ldots, k\}$
2. $a_1 = 1$
3. $a_k = 1$
4. $\mathcal{P}(t) \neq 0, \forall\, t \in \mathbf{Z_2}$ or $\mathcal{P}(t) = 1, \forall\, t \in \mathbf{Z_2}$

2.2.2 Algorithm for generation

From the criteria set in the previous section, one can see that $\mathcal{P}(t) = 1$ when $t = 0$. Thus, 0 is definitely not a zero of $\mathcal{P}(t)$.

When $t = 1$, $\mathcal{P}(t)$ will evaluate to 0 when there is an even number of $a_i \neq 0$. Thus, there must be an odd number of $a_i \neq 0$ in order that $\mathcal{P}(t)$ does not evaluate to 0. In other words, $\mathcal{P}(t)$ is irreducible if there is an odd number of $a_i \neq 0$.

When $\mathcal{P}(t)$ is of degree k, it requires $k+1$ bits to represent the polynomial. In other words, a polynomial of degree 64 needs 65 bits. In C, word length is 32 bits. In order to make full use of the word size, it is decided to drop the MSB since it is always 1. The following algorithm is designed for the purpose of generating an irreducible polynomial in $\mathbf{Z_2}$ of degree k (multiples of 32). The result will be stored in 32-bit words W_j with the MSB dropped.

Pre-condition: $k \geq 0$ and $32 \mid k$.

1. int $r :=$ random number $\in \left[0, \lfloor \frac{k-1}{2} \rfloor - 1\right]$
2. $r := 2r + 1$
3. $A := int\ Array[0 \ldots k-1], \qquad A[i] := 0, \forall\ i \in \{0, \ldots, k-1\}$
4. $A[0] := 1$
5. For $i := 1 \ldots r,$ do
   repeat
   int $j :=$ random number $\in [0, k-2]$
   until $A[j+1] == 0$
   $A[j+1] := 1$
   $i := i + 1$
   enddo
6. $n := k$ div 32
7. For $j := 1, \ldots, n$ do
   $W_j := 0$
   For $i := 32 * j - 1, 32 * j - 2, \ldots, 32 * j - 32$ do
   $W_j := 2 * W_j + A[i]$
   enddo
   enddo

Post conditions:

1. sum $:= \sum_{j=1}^{k \text{ div } 32} W_j * 2^{j-1}$ representsin binary form $\mathcal{P}(t)$ with MSB dropped.
2. W_j, $j = 1, \ldots, n$ are the n 32-bit words storing sum,
3. $\mathcal{P}(t)$ (with MSB dropped) is represented in the form $[W_n, \ldots, W_2, W_1]$

2.3 An Implementation

Andrei Broder [5] has proposed an implementation scheme for Rabin's method for $k = 64$. Since no mathematical work supporting this scheme is included in the paper, it is difficult to follow and to extend to other values of k. In order to allow the readers to appreciate the scheme and to allow implementers to extend the application to other values of k, the mathematical work is reproduced in section 2.3.1.

2.3.1 Mathematical Work

2.3.1.1 Notations

Bits, Bytes and Words

$$w_i \mapsto i^{th} \text{ bit of } \mathcal{W}, \qquad W_i \mapsto i^{th} \text{ byte of } \mathcal{W}, \qquad \mathcal{L}[i] \mapsto i^{th} \text{ word of } \mathcal{L}$$

Given a byte $\mathcal{B}$, $\mathcal{B} = [b_1, b_2, \ldots, b_7, b_8]$
Given a 32-bit word $\mathcal{W}$,

$$\mathcal{W} = [w_1, w_2, \ldots, w_{31}, w_{32}] = [W_1, W_2, W_3, W_4]$$

Given a 64-bit value $\mathcal{D}$,

$$\begin{aligned}\mathcal{D} &= [d_1, d_2, \ldots, d_{63}, d_{64}] = [D_1, D_2, \ldots D_7, D_8]\\ \mathcal{D}[1] &= [D_1, D_2, D_3, D_4]\\ \mathcal{D}[2] &= [D_5, D_6, D_7, D_8]\end{aligned}$$

Given a 96-bit value $\mathcal{L}$,

$$\begin{aligned}\mathcal{L} &= [l_1, l_2, \ldots, l_{95}, l_{96}] = [L_1, L_2, \ldots L_{11}, L_{12}]\\ \mathcal{L}[1] &= [L_1, L_2, L_3, L_4]\\ \mathcal{L}[2] &= [L_5, L_6, L_7, L_8]\\ \mathcal{L}[3] &= [L_9, L_{10}, L_{11}, L_{12}]\end{aligned}$$

Associated polynomials
If $\mathcal{W} = [W_1, W_2, W_3, W_4]$, then

$$\begin{aligned}\mathcal{W}(t) &\doteq W_1(t)t^{24} + W_2(t)t^{16} + W_3(t)t^8 + W_4(t)\\ W_1(t) &\doteq w_1t^7 + w_2t^6 + w_3t^5 + w_4t^4 + w_5t^3 + w_6t^2 + w_7t + w_8\\ W_2(t) &\doteq w_9t^7 + w_{10}t^6 + w_{11}t^5 + w_{12}t^4 + w_{13}t^3 + w_{14}t^2 + w_{15}t + w_{16}\\ W_3(t) &\doteq w_{17}t^7 + w_{18}t^6 + w_{19}t^5 + w_{20}t^4 + w_{21}t^3 + w_{22}t^2 + w_{24}t + w_{24}\\ W_4(t) &\doteq w_{25}t^7 + w_{26}t^6 + w_{27}t^5 + w_{28}t^4 + w_{29}t^3 + w_{30}t^2 + w_{31}t + w_{32}\end{aligned}$$

Fingerprinting
$\mathcal{S}$ = processed string with fingerprint $F_c = f(\mathcal{S})$
$\mathcal{W} = [W_a, W_2, W_3, W_4]$ = next word to be processed
F_n = the new fingerprint = $f(concat(\mathcal{S}, \mathcal{W}))$

2.3.1.2 For k = deg($\mathcal{P}(t)$) = 32

$$F_c = [F_{c1}, F_{c2}, F_{c3}, F_{c4}]$$

By property 5 and 6 of section 2.1.3, we have

$$\begin{aligned}
F_n &= f(concat(\mathcal{S}, \mathcal{W})) \\
&= f(concat(F_c, \mathcal{W})) \\
&= (F_c(t)t^{32} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
&= (F_c(t)t^{32}) \bmod \mathcal{P}(t) + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
&= (F_{c1}(t)t^{56} + F_{c2}(t)t^{48} + F_{c3}(t)t^{40} + F_{c4}(t)t^{32} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
&= \mathsf{TA}(t) + \mathsf{TB}(t) + \mathsf{TC}(t) + \mathsf{TD}(t) + TW(t) \\
where \quad \mathsf{TA}(t) &= (F_{c1}(t)t^{56}) \bmod \mathcal{P}(t) \\
\mathsf{TB}(t) &= (F_{c2}(t)t^{48}) \bmod \mathcal{P}(t) \\
\mathsf{TC}(t) &= (F_{c3}(t)t^{40}) \bmod \mathcal{P}(t) \\
\mathsf{TD}(t) &= (F_{c4}(t)t^{32}) \bmod \mathcal{P}(t) \\
TW(t) &= \mathcal{W}(t) \bmod \mathcal{P}(t)
\end{aligned}$$

Thus, $F_n \doteq \mathsf{TW}$ **XOR** TA **XOR** TB **XOR** TC **XOR** TD (2)

2.3.1.3 For k = deg($\mathcal{P}(t)$) = 64

$$F_c = [F_{c1}, F_{c2}, \ldots F_{c7}, F_{c8}] = \big[F_c[1], F_c[2]\big]$$

By property 5 and 6 of section 2.1.3, we have

$$\begin{aligned}
F_n =& f(concat(\mathcal{S}, \mathcal{W})) \\
=& f(concat(F_c, \mathcal{W})) \\
=& (F_c(t)t^{32} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
=& (t^{32}\{t^{32}F_c[1](t) + F_c[2](t)\} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
=& (F_{c1}(t)t^{88} + F_{c2}(t)t^{80} + F_{c3}(t)t^{72} + F_{c4}(t)t^{64} + F_c[2](t)t^{32} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
=& \underbrace{\mathsf{TA}[1](t)t^{32} + \mathsf{TA}[2](t)}_{\text{from } F_{c1}(t)t^{88} \bmod \mathcal{P}(t)} + \underbrace{\mathsf{TB}[1](t)t^{32} + \mathsf{TB}[2](t)}_{\text{from } F_{c2}(t)t^{80} \bmod \mathcal{P}(t)} + \underbrace{\mathsf{TC}[1](t)t^{32} + \mathsf{TC}[2](t)}_{\text{from } F_{c3}(t)t^{72} \bmod \mathcal{P}(t)} \\
&+ \underbrace{\mathsf{TD}[1](t)t^{32} + \mathsf{TD}[2](t)}_{\text{from } F_{c4}(t)t^{64} \bmod \mathcal{P}(t)} + F_c[2](t)t^{32} + \mathcal{W}(t) \\
=& t^{32}\{F_c[2](t) + \mathsf{TA}[1](t) + \mathsf{TB}[1](t) + \mathsf{TC}[1](t) + \mathsf{TD}[1](t)\} \\
&+ \{\mathcal{W}(t) + \mathsf{TA}[2](t) + \mathsf{TB}[2](t) + \mathsf{TC}[2](t) + \mathsf{TD}[2](t)\} \\
=& F_n[1](t)t^{32} + F_n[2](t)
\end{aligned}$$

Thus, $F_n[1] \doteq F_c[2]$ **XOR** $\mathsf{TA}[1]$ **XOR** $\mathsf{TB}[1]$ **XOR** $\mathsf{TC}[1]$ **XOR** $\mathsf{TD}[1]$ (3)

$F_n[2] \doteq W$ **XOR** $\mathsf{TA}[2]$ **XOR** $\mathsf{TB}[2]$ **XOR** $\mathsf{TC}[2]$ **XOR** $\mathsf{TD}[2]$ (4)

2.3.1.4 For k = deg($\mathcal{P}(t)$) = 96

$$F_c = [F_{c1}, F_{c2}, \ldots F_{c11}, F_{c12}] = \big[F_c[1], F_c[2], F_c[3]\big]$$

By property 5 and 6 of section 2.1.3, we have

$$\begin{aligned}
F_n =& f(concat(\mathcal{S}, \mathcal{W})) = f(concat(F_c, \mathcal{W})) \\
=& (F_c(t)t^{32} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
=& (t^{32}\{t^{64}F_c[1](t) + t^{32}F_c[2](t) + F_c[3](t)\} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
=& (F_{c1}(t)t^{120} + F_{c2}(t)t^{112} + F_{c3}(t)t^{104} + F_{c4}(t)t^{96} + F_c[2](t)t^{64} + F_c[3](t)t^{32} + \mathcal{W}(t)) \bmod \mathcal{P}(t) \\
=& \underbrace{\mathsf{TA}[1](t)t^{64} + \mathsf{TA}[2](t)t^{32} + \mathsf{TA}[3](t)}_{\text{from } F_{c1}(t)t^{120} \bmod \mathcal{P}(t)} + \underbrace{\mathsf{TB}[1](t)t^{64} + \mathsf{TB}[2](t)t^{32} + \mathsf{TB}[3](t)}_{\text{from } F_{c2}(t)t^{112} \bmod \mathcal{P}(t)} \\
& + \underbrace{\mathsf{TC}[1](t)t^{64} + \mathsf{TC}[2](t)t^{32} + \mathsf{TC}[3](t)}_{\text{from } F_{c3}(t)t^{104} \bmod \mathcal{P}(t)} + \underbrace{\mathsf{TD}[1](t)t^{64} + \mathsf{TD}[2](t)t^{32} + \mathsf{TD}[3](t)}_{\text{from } F_{c4}(t)t^{96} \bmod \mathcal{P}(t)} \\
& + F_c[2](t)t^{64} + F_c[3](t)t^{32} + \mathcal{W}(t) \\
=& t^{64}\{F_c[2](t) + \mathsf{TA}[1](t) + \mathsf{TB}[1](t) + \mathsf{TC}[1](t) + \mathsf{TD}[1](t)\} \\
& + t^{32}\{F_c[3](t) + \mathsf{TA}[2](t) + \mathsf{TB}[2](t) + \mathsf{TC}[2](t) + \mathsf{TD}[2](t)\} \\
& + \{\mathcal{W}(t) + \mathsf{TA}[3](t) + \mathsf{TB}[3](t) + \mathsf{TC}[3](t) + \mathsf{TD}[3](t)\} \\
=& F_n[1](t)t^{64} + F_n[2](t)t^{32} + F_n[3](t)
\end{aligned}$$

Thus,
$$F_n[1] \doteq F_c[2]\ \mathbf{XOR}\ \mathsf{TA}[1]\ \mathbf{XOR}\ \mathsf{TB}[1]\ \mathbf{XOR}\ \mathsf{TC}[1]\ \mathbf{XOR}\ \mathsf{TD}[1] \quad (5)$$
$$F_n[2] \doteq F_c[3]\ \mathbf{XOR}\ \mathsf{TA}[2]\ \mathbf{XOR}\ \mathsf{TB}[2]\ \mathbf{XOR}\ \mathsf{TC}[2]\ \mathbf{XOR}\ \mathsf{TD}[2] \quad (6)$$
$$F_n[3] \doteq W\ \mathbf{XOR}\ \mathsf{TA}[3]\ \mathbf{XOR}\ \mathsf{TB}[3]\ \mathbf{XOR}\ \mathsf{TC}[3]\ \mathbf{XOR}\ \mathsf{TD}[3] \quad (7)$$

2.3.1.5 Summary of Operations for Implementation

From works in this section (equations 2 to 7), the following is a summary of the essentials for the implementation of Rabin's method.

k = deg $\mathcal{P}(\mathbf{t})$	32
Fingerprint Value	$F_n = [F_{n1}, F_{n2}, F_{n3}, F_{n4}]$
Processes	$F_n \doteq TW$ **XOR** TA **XOR** TB **XOR** TC **XOR** TD
Basic Operations	$\mathsf{TA}(t) = (F_{c1}(t)t^{56}) \bmod \mathcal{P}(t)$ $\mathsf{TB}(t) = (F_{c2}(t)t^{48}) \bmod \mathcal{P}(t)$ $\mathsf{TC}(t) = (F_{c3}(t)t^{40}) \bmod \mathcal{P}(t)$ $\mathsf{TD}(t) = (F_{c4}(t)t^{32}) \bmod \mathcal{P}(t)$ $TW(t) = \mathcal{W}(t) \bmod \mathcal{P}(t)$

$\mathbf{k} = \deg \mathcal{P}(\mathbf{t})$	64
Fingerprint Value	$F_n = [F_{n1}, \ldots, F_{n8}] = \boxed{F_n[1], F_n[2]}$
Processes	$F_n[1] \doteq F_c[2]$ **XOR** $\mathsf{TA}[1]$ **XOR** $\mathsf{TB}[1]$ **XOR** $\mathsf{TC}[1]$ **XOR** $\mathsf{TD}[1]$ $F_n[2] \doteq W$ **XOR** $\mathsf{TA}[2]$ **XOR** $\mathsf{TB}[2]$ **XOR** $\mathsf{TC}[2]$ **XOR** $\mathsf{TD}[2]$
Basic Operations	$\mathsf{TA}(t) = (F_{c1}(t)t^{88}) \bmod \mathcal{P}(t)$ $\mathsf{TB}(t) = (F_{c2}(t)t^{80}) \bmod \mathcal{P}(t)$ $\mathsf{TC}(t) = (F_{c3}(t)t^{72}) \bmod \mathcal{P}(t)$ $\mathsf{TD}(t) = (F_{c4}(t)t^{64}) \bmod \mathcal{P}(t)$

$\mathbf{k} = \deg \mathcal{P}(\mathbf{t})$	96
Fingerprint Value	$F_n = [F_{n1}, \ldots, F_{n12}] = \boxed{F_n[1], F_n[2], F_n[3]}$
Processes	$F_n[1] \doteq F_c[2]$ **XOR** $\mathsf{TA}[1]$ **XOR** $\mathsf{TB}[1]$ **XOR** $\mathsf{TC}[1]$ **XOR** $\mathsf{TD}[1]$ $F_n[2] \doteq F_n[3]$ **XOR** $\mathsf{TA}[2]$ **XOR** $\mathsf{TB}[2]$ **XOR** $\mathsf{TC}[2]$ **XOR** $\mathsf{TD}[2]$ $F_n[3] \doteq W$ **XOR** $\mathsf{TA}[3]$ **XOR** $\mathsf{TB}[3]$ **XOR** $\mathsf{TC}[3]$ **XOR** $\mathsf{TD}[3]$
Basic Operations	$\mathsf{TA}(t) = (F_{c1}(t)t^{120}) \bmod \mathcal{P}(t)$ $\mathsf{TB}(t) = (F_{c2}(t)t^{112}) \bmod \mathcal{P}(t)$ $\mathsf{TC}(t) = (F_{c3}(t)t^{104}) \bmod \mathcal{P}(t)$ $\mathsf{TD}(t) = (F_{c4}(t)t^{96}) \bmod \mathcal{P}(t)$

2.3.2 Implementation Issues

To implement the Rabin's method of fingerprinting, several issues have to be dealt with. These include the choice of programming language, representation of the irreducible polynomial and the fingerprint value, and implementing the basic operations. These will be discussed in the subsequent sections.

2.3.2.1 Programming Language

It is decided to use C++ as the programming language for implementing Rabin's method for the following reasons:

1. Embedded SQL can be used with C language. This facilitates the access of the database storing snapshots of webpages and their URLs which may be used as the dataset for the experiments in this project. Embedded SQL also facilitates the direct manipulation of data in databases.

2. Language C permits low level programming which is required to implement the Rabin's method using bit shift and bit-wise exclusive OR operations.

3. Language C++ permits use of codes written in C and also allows object-oriented programming which may make coding in the project easier.

2.3.2.2 Representation

As discussed in section 2.2, the irreducible polynomial with the MSB dropped will be stored in several 32-bit words. This can also be used for storing the fingerprint value. When using $\mathcal{P}(t)$ of degree k, k bits will be required to store the fingerprint value.

In order to genrealize the codes, the following scheme is used to store the fingerprint value $f(\mathcal{S})$ and $\mathcal{P}(t)$:

1. The values will be stored in $n = \lceil \frac{k}{32} \rceil$ 32-bit words.
2. The n 32-bit words will be stored in a data structure (array).
3. The first element of the array is the word containing the LSB.

2.3.2.3 Bit Shift

From properties listed in section 2.1.3, one can see that most of the operations required for implementing Rabin's method need low level bits shifting and bitwise exclusive OR. However, these operations are available to 32 bit words only. Thus, a set of special procedures are required for implementing shifts for word size greater than 32 bits. The following blocks are written for $k > 32$ with `sWord` representing the data structure containing the word to be operated on.

```
void longLeftShift(int * sWord, int s) {
   int len = number of elements contained in sWord;
   int temp;
   int mask = 0x00000000;

   // generate mask for extracting s bits
   for (int i = 0; i < s; i++)
      mask = mask << 1 | 0x00000001;
   for (int i = len - 1; i > 0; i--) {
      // extract s highest bits from lower word
      temp = sWord[i - 1] >> (32 - s) & mask;
      // insert as s lowest bits in higher word
      sWord[i] = sWord[i] << s | temp;
   }
   sWord[0] = sWord[0] << s
}
```

2.3.2.4 Computing $\mathcal{W}(t) \bmod \mathcal{P}(t)$

Using the same notation used in section 2.3.1, W is a 32 bit word. Thus $\mathcal{W}(t)$ is of degree 31 and we have

$$\mathcal{W}(t) \bmod \mathcal{P}(t) = \mathcal{W}(t) \qquad \forall k \geq 32$$

As we are only dealling with

$$k \geq 32$$

in this project, the above relation holds for all computation in this project.

2.3.2.5 Computing $i \times t^8 \bmod \mathcal{P}(t)$

Another commonly found operation in the implementation is $i \times t^8 \bmod \mathcal{P}(t)$ where $0 \leq i \leq 255$ and i is stored in the lowest byte of $\mathcal{W}$. This can be achieved by applying property 7 of section 2.1.3 eight times. Once again, all $\mathcal{P}(t)$ representations are with MSB dropped. With `sWord` and `P` denoting $\mathcal{W}$ and $\mathcal{P}(t)$, the code block is as follows:

```
void multiT8(int * sWord, int * P){
   int len = number of elements contained in sWord;
   int needXOR;
   for (int i = 0; i < 8; i++) {
      // XOR conditioned on MSB
      needXOR =
      ((sWord[len - 1] & 0x80000000) == 0x80000000);
      // left shift 1 bit
      longLeftShift(sWord, 1);
      if( ( needXOR ) {
         for (int j = 0; j < len; j++) {
            sWord[j] = sWord[j] ∧ P[j];
         }
      }
   }
}
```

2.3.2.6 Computing $i \times t^{k-8} \bmod \mathcal{P}(t)$

Since i is an 8-bit value stored in a 32-bit word and the associated polynomial is of degree 7, $i \times t^{k-8} \bmod \mathcal{P}(t)$ can be found by putting the 8 bits as the 8 MSB of the result which is stored in `sWord`.

```
void byteShift(int *sWord, int i){
   int len = number of elements contained in sWord;
   sWord[len - 1] = i << 24;
   for (int j = 0; j < len - 1; j++) {
      sWord[j] = 0x00000000;
   }
}
```

2.3.2.7 Pre-computed Table

In section 2.3.1, equations (2) to (7) indicates that TA, TB, TC and TD values are frequently used to compute the fingerprint values. Since a byte can only take values between 0 and 255, it is efficient to have all possible values of TA, TB, TC and TD to be pre-computed and stored in a table for future reference. Suppose the tables are known as TA, TB, TC and TD, we find the following relationships:

$$\begin{aligned}
\mathsf{TD}[i] &= i * t^{k} \bmod \mathcal{P}(t) = (i * t^{k-8}) * t^{8} \bmod \mathcal{P}(t) \\
\mathsf{TC}[i] &= i * t^{(k+8)} \bmod \mathcal{P}(t) = \mathsf{TD}[i] * t^{8} \bmod \mathcal{P}(t) \\
\mathsf{TB}[i] &= i * t^{(k+16)} \bmod \mathcal{P}(t) = \mathsf{TC}[i] * t^{8} \bmod \mathcal{P}(t) \\
\mathsf{TA}[i] &= i * t^{(k+24)} \bmod \mathcal{P}(t) = \mathsf{TB}[i] * t^{8} \bmod \mathcal{P}(t)
\end{aligned}$$

This can achieved by applying the code blocks discussed in the previous two sections repeatedly. Once again, with P denoting $\mathcal{P}(t)$, the code block for computing the tables (TA, TB, TC and TD), which are assumed to be globally accessible, is as follows:

```
void computeTable( int * P ) {
   int len = number of elements contained in sWord;
   for (int i = 0; i < 256; i++) {
      for (int j = 0; j < len; j++) {
         TD[i][j] = 0;
      }
      byteShift(TD[i], i);    // TD[i] = i × t^(k-8)
      multiT8(TD[i], P);      // TD[i] = i × t^8
      for (int j = 0; j < len; j++) {
         TC[i][j] = TD[i][j];
      }
      multiT8(TC[i], P);
      for (int j = 0; j < len; j++) {
         TB[i][j] = TC[i][j];
      }
      multiT8(TB[i], P);
      for (int j = 0; j < len; j++) {
         TA[i][j] = TB[i][j];
      }
      multiT8(TA[i], P);
   }
}
```

2.3.2.8 Fingerprinting (concatenating 4-character string)

According to work in section 2.3.1.5, each cycle of computation will take on 4 bytes. Suppose the next 4 characters to be concatenated are contained in 32-bit word W as [W1, W2, W3, W4]. Assuming that the previous fingerprint value is stored in `sWord` and $\mathcal{P}(t)$ representated by `P`, the function to compute the new fingerprint will be as shown below based on equations 2 to 7 and the assumption that the tables (`TA`, `TB`, `TC` and `TD`) are globally accessible:

```
void fp4(int * sWord, char W1, char W2, char W3,
char W4){
   int len = number of elements contained in sWord;
   int f1 = sWord[len - 1] >> 24 & 0xffff;
   int f2 = sWord[len - 1] >> 16 & 0xffff;
   int f3 = sWord[len - 1] >> 8 & 0xffff;
   int f4 = sWord[len - 1] & 0xffff;
   int W = (((W | W1) << 8 | W2) << 8 | W3) << 8 |
   W4;
   for (int j = len - 1; j > 0; j--) {
      sWord[j] = sWord[j - 1] ∧ TA[f1][j] ∧
      TB[f2][j] ∧ TC[f3][j] ∧ TD[f4][j];
   }
   sWord[0] = W ∧ TA[f1][0] ∧ TB[f2][0] ∧ TC[f3][0]
   ∧ TD[f4][0];
}
```

2.3.2.9 Fingerprinting (concatenating 1 character)

The sitauation that the end chunk of a string is not a 32-bit word can be handled by dealing with 1 character at a time using property 7 of section 2.1.3. Code block as follows:

```
void fp1(int * sWord, int * P, char S){
   int len = number of elements contained in sWord;
   int W, needXOR;
   for (int i = 0; i < 8; i++) {
      needXOR = (sWord[len - 1] & 0x80000000);
      W = S & 0x80;
      S = S << 1;
      longLeftShift(sWord, 1);
      sWord[0] = sWord[0] | (W >> 7);
      if ( needXOR ) {
         for (int j = len - 1; j >= 0; j --) {
            sWord[j] = sWord[j] ∧ P[j];
         }
      }
   }
}
```

2.3.3 Implementation Scheme

This scheme is designed for $k \geq 32$ based on discussions in the previous sections. The source codes of this implementation is included in the section 7.1.

Pre-condition: S is a non-zero length character string

1. Obtain $\mathcal{P}(t)$ of degree k and store as P
2. computeTable
3. set F = 0
4. int $n :=$ len(S) div 4
5. For $i := 1 \ldots n,$ do

   Perform
   fp4(*F, S[4*i-4], S[4*i-3], S[4*i-2], S[4*i-1])

   enddo
6. For each of the character c in the remaining chunk, do

   Perform fp1(F, P, c)

   enddo

Post condition: $f(S)$ is stored as F

Chapter 3

Hashing Algorithms

3.1 General Considerations

In this project, result of using Rabin's method will be compared with those using hasing functions. Four schemes have been chosen after evaluating several published hashing methods. they are described in the following sections.

3.2 Scheme djb2

Dan Bernstein [6] reported this algorithm many years ago. The code block is included below for reference. The codes are self-explanatory.

```
unsigned int djb2( char* str ) {
   unsigned int hash = 5381;
   int c;
   while ( c = *str++ )
      hash = ((hash << 5) + hash) + c;
      // hash * 33 + c
   return hash;
}
```

3.3 Scheme sdbm

It is reported [6] that this algorithm is created for the sdbm database library. The underlying function is

$$hash(i) = hash(i-1) * 65599 + str[i]$$

```
unsigned int sdbm( char* str ) {
   unsigned int hash = 0; int c;
   while ( c = *str++ )
      hash = c + (hash << 6) + (hash << 16) - hash;
   return hash;
}
```

3.4 Horner's Rule

Bruno Preiss [7] adopted the Horner's rule[1] to write the following hashing function.

```
unsigned int hhash( char* str ) {
   unsigned int hash = 0;
   int c;
   while ( c = *str++ )
      hash = hash << 7 ∧ c;
   return hash;
}
```

3.5 Conversion function url2pid()

This is a conversion function used in the database storing the target dataset for converting a character string into an 8 byte character string used as identifying key. The codes can be found in the section 7.2.

[1] By Horner's rule, the value $result = \sum_{i=0}^{n} a_i x^i$ can be found by the algorithm:
result := a_n; for i := n - 1,. . . ,1,0 result := result * x + a_i

Chapter 4

The Experiment

4.1 Test Objects

The experiment is designed to find out the following:

- the dependency of the percentage of collisions (distinct strings with the same fingerprint) and the choice of $\mathcal{P}(t)$ of the same degree k
- the relationship between the percentage of collisions (distinct strings with the same fingerprint) and the degree of $\mathcal{P}(t)$
- the effectiveness of using Rabin's method with that of using hashing functions
- the efficiency of Rabin's method in terms of CPU time required as compared with the time required using various hashing fuctions

4.2 Outline of Experiment

One of the tables stored in the research databases[1] contains a collection of the URL of more than 23.5 million web pages. These URLs will be the target strings to be fingerprinted. The records are also recorded in a text file with each record contained in one single line of text.

Each line of text contains data in the order as follows:

```
name of attributes  data format  descriptions
pid                 char(8)      page identity
url                 char(128)    first part of the URL
url2                char(872)    remaining part of the URL
```

[1] At the time of the project, it is located at /usr/bluesky-r1/cmput690/db2/

4.3 Details of the Experiment

The following tests will be performed on the dataset:

1. For each S of the URLs, find $f(S)$ using various $\mathcal{P}(t)$ of degree 32, 64 and 96.
2. For each S of the URLs, find djb2(S)
3. For each S of the URLs, find sdbm(S)
4. For each S of the URLs, find hhash(S)
5. For each S of the URLs, find url2pid(S)
6. Each of the finerprints and hash values obtained is stored in a separate text file
7. For each of the above tests, log the CPU time required
8. For each of the above tests, find out the number of duplicate values for distinct URLs using the sort function

Chapter 5

Results and Analyses

5.1 Test Results

When the compiled program is executed, results were recorded in various output log files. These files are processed and merged into one single result file. The file contents are included below for analysis. In order to have a fair comparison in time efficiency, only the CPU time for each process is measured.

```
Size of dataset = 23743961

**************************************

The polynomial is of degree 32
Irreducible polynomial used is 1 4D96487B
Total CPU time for 32 bit fingerprinting = 1617.080000 seconds

Size after removing duplicates = 23678578 f32_1.txt

**************************************

The polynomial is of degree 32
Irreducible polynomial used is 1 2DC7EEB3
Total CPU time for 32 bit fingerprinting = 1583.680000 seconds

Size after removing duplicates = 23678346 f32_2.txt

**************************************

The polynomial is of degree 32
Irreducible polynomial used is 1 1100C021
Total CPU time for 32 bit fingerprinting = 1504.040000 seconds
```

```
Size after removing duplicates = 23678200 f32_3.txt

****************************************

The polynomial is of degree 32
Irreducible polynomial used is 1 53BCFEDB
Total CPU time for 32 bit fingerprinting = 1506.460000 seconds

Size after removing duplicates = 23678797 f32_4.txt

****************************************

The polynomial is of degree 32
Irreducible polynomial used is 1 00401003
Total CPU time for 32 bit fingerprinting = 1404.200000 seconds

Size after removing duplicates = 23678626 f32_5.txt

****************************************

The polynomial is of degree 64
Irreducible polynomial used is 1 460C8808 10028043
Total CPU time for bit fingerprinting = 2005.600000 seconds

Size after removing duplicates = 23743961 f64_1.txt

****************************************

The polynomial is of degree 64
Irreducible polynomial used is 1 7523C013 A96DD7FF
Total CPU time for 64 bit fingerprinting = 1775.430000 seconds

Size after removing duplicates = 23743961 f64_2.txt

****************************************

The polynomial is of degree 64
Irreducible polynomial used is 1 7FABFBF6 5FFFFEFF
Total CPU time for 64 bit fingerprinting = 1804.740000 seconds

Size after removing duplicates = 23743961 f64_3.txt

****************************************

The polynomial is of degree 64
Irreducible polynomial used is 1 01751A98 4D90AF27
```

```
Total CPU time for 64 bit fingerprinting = 1846.320000 seconds

Size after removing duplicates = 23743961 f64_4.txt

**************************************

The polynomial is of degree 64
Irreducible polynomial used is 1 77FFFFFF FFDFFFBF
Total CPU time for 64 bit fingerprinting = 1884.140000 seconds

Size after removing duplicates = 23743961 f64_5.txt

**************************************

The polynomial is of degree 96
Irreducible polynomial used is 1 0200100A A0300111 2125200D
Total CPU time for bit fingerprinting = -1934.287296 seconds

Size after removing duplicates = 23743961 f96_1.txt

**************************************

The polynomial is of degree 96
Irreducible polynomial used is 1 6FF7FFFF FFFE797F FFFFFFFF
Total CPU time for 96 bit fingerprinting = 1960.210000 seconds

Size after removing duplicates = 23743961 f96_2.txt

**************************************

The polynomial is of degree 96
Irreducible polynomial used is 1 62500258 258182C6 D985013D
Total CPU time for 96 bit fingerprinting = 1888.950000 seconds

Size after removing duplicates = 23743961 f96_3.txt

**************************************

The polynomial is of degree 96
Irreducible polynomial used is 1 7FFFFFFF FBFFFFFF FFEBFFFF
Total CPU time for 96 bit fingerprinting = 1872.290000 seconds

Size after removing duplicates = 23743958 f96_4.txt

**************************************

The polynomial is of degree 96
```

```
Irreducible polynomial used is 1 01040008 00466C86 04001109
Total CPU time for 96 bit fingerprinting = 1989.230000 seconds

Size after removing duplicates = 23743961 f96_5.txt

****************************************

Hashing result
Hashing algorithm used : djb2
Total CPU time for hashing = 1068.070000 seconds

Size after removing duplicates = 23677811 hash1.dat

****************************************

Hashing result
Hashing algorithm used : sdbm
Total CPU time for hashing = 1134.640000 seconds

Size after removing duplicates = 23678248 hash2.dat

****************************************

Hashing result
Hashing algorithm used : Horner’s rule
Total CPU time for hashing = 1273.920000 seconds

Size after removing duplicates = 724143 hash3.dat

****************************************

Hashing result
Hashing algorithm used : url2pid
Total CPU time for hashing = 2121.483520 seconds

Size after removing duplicates = 23736299 dhash.dat

****************************************
```

5.2 Analyses

5.2.1 Expected Errors

Given the context of this experiment, the number of collisions expected in each of the experiments can be calculated according to the bound of error (2.1.2) proposed by Rabin in his paper [1] as follows:

$n =$ number of character strings $= 23,743,961$

$m =$ maximum length of strings in bits $= 8 \times 1,000 = 8,000$

For $k = 32, Pr \leq 1.05 \times 10^{11}\%$, maximum collisions $= 23,743,961$

For $k = 64, Pr \leq 24.45\%$, maximum collisions $= 5,805,368$

For $k = 96, Pr \leq 5.69 \times 10^{-9}\%$, maximum collisions $= 1.35 \times 10^{-3} \approx 0$

In case $k = 32$, the bound of error is meaningless since it is over 1. This is an indication that the value of k should be changed according to the expected total number of candidates for fingerprinting in order to have a quality output.

5.2.2 Actual Results

As the results file is lengthy, the data are processed and organized into tabular form for easy reference.

algorithm	run	CPU time (sec)	collisions	% collisions
32-bit fingerprinting	1	1,617.08	65,383	0.275%
	2	1,583.68	65,615	0.276%
	3	1,504.04	65,761	0.277%
	4	1,506.46	65,164	0.274%
	5	1,404.20	65,335	0.275%
64-bit fingerprinting	1	2,005.60	0	0%
	2	1,775.43	0	0%
	3	1,804.74	0	0%
	4	1,846.32	0	0%
	5	1,884.14	0	0%
96-bit fingerprinting	1	-1,934.29	0	0%
	2	1,960.21	0	0%
	3	1,888.95	0	0%
	4	1,872.29	2	$\approx$ 0%
	5	1,989.23	0	0%
hashing - djb2	1	1,068.07	66,150	0.279%
hashing - sdbm	1	1,134.64	65,713	0.277%
hashing - Horner's rule	1	1,273.92	23,019,818	96.95%
hashing - url2pid	1	2,121.48	7,662	0.032%

5.2.3 Comparisons

5.2.3.1 Note of abnormality

The negative time recorded for the first run with 96 bits fingerprinting method is due to the fact that the codes for that run was not written with the expectation of counter overflow. This was fixed in the subsequent runs.

5.2.3.2 Collisions

All schemes used in this project are performing well with the exception of the Horner's rule which is therefore not included in this discussion any more. As expected, the fingerprint methods using 64 bits or more give promising results. Fingerprint method employing only 32 bits is simliar to the other hashing functions and thus has errors of the same order of magnitude as the others. The url2pid algorithm seems to be edging the other functions of similar base.

5.2.3.3 Time efficiency

While the url2pid algorithm gives a better result (among those 32-bit based algorithms), it is the slowest of all. The fingerprint methods perform consistently independent of the degree of the irreducible plynomial. Of the 32-bit based functions, the two hashing functions namely djb2 and sdbm are the most efficient ones. This explains why they have good reputation within the community.

5.2.4 Fingerprint method

The following observations can be made from the tabulated test results:

1. The number of collisions is independent of the choice of the irreducible polynomial used for hashing provided that they are of the same degree
2. The expression for estimating the upper bound of errors is not violated in the contetext of the experiment.
3. The performance of the method improves with the increase of the degree of the irreducible polynomial used

Chapter 6

Conclusion

6.1 Conclusions

The Rabin's method of fingerprinting using irreducible polynomials has been studied and implemented. As a result of the implementation and experiments with a given dataset, the followings are verified:

1. The result of application of the Rabin's method is independent of the choice of irreducible polynomial

2. The probability of error (different strings having the same fingerprint) decreases with the increase of the degree of the irreducible polynomial used

3. The probability of error is bounded by the expression provided in his paper [1] in which Rabin released this fingerprinting method

6.2 Related and Future Work

6.2.1 Related Work

The famous MD4 and MD5 message-digest algorithms are direct applications of this fingerprinting method. Both algorithms employ 128-bit fingerprinting with MD4 being faster. They use padding at the end of the string to avoid single character concatenation. Each string is padded to a length equal to a multiple of 512 bits. A 16-word (32 bit word) block is processed each cycle.

6.2.2 Future Work

Due to the constraint of resources, only a limited number of runs have been performed. To conform to the Principle of Large Numbers, more runs should be executed to obtain results to verify the above observations made in the conclusion section.

Besides, 2 collisions were found in the fourth run using polynomial of degree 96, it is not expected. The situation should be examined to find out the cause of the problem. It would be good to know if the errors are just coincidence or erratic program codes or some other reasons. However, this cannot be done in time to meet the submission deadline of this project.

Chapter 7

Source Codes

7.1 Source Codes for Fingerprinting

```
/*************************************************************************
 * Authors       :  Calvin Chan & Haihua Lu
 * Course Project:  CMPUT 690 -- Advanced Database System, December, 2001
 * Instructor    :  Dr. Davood Rafiei
 *
 * Filename      :  fingerPrint.cc
 *
 * Purpose       :  to run finger printing test on pages.txt
 *
 * Usage         :  fp <degree> <output file name> < input stream
 *                  if no output file name is given, "outfp.txt" will be used
 *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <math.h>
#include <time.h>

/***** Type Definitions *****/
typedef unsigned int usInt;

/***** Forward Declarations *****/

// for generating irreducible polynomial of degree k
// wCount being the number of 32 bit words required to hold P(t)
usInt* generateIrreduciblePoly(short k, short *wCount);

// for generating pre-computed table TA, TB, TC, TD
void computTable
```

```
  (usInt **TA, usInt **TB, usInt **TC, usInt **TD, usInt *P, short wCount);

// for left shifting values s bits
void longLeftShift(usInt *words, short wCount, short s);

// for computing i*t^8 mod P(t)
void multiT8(usInt *words, usInt *P, short wCount);

// for computing i*t^(k-8) mod P(t)
void byteShift(usInt *words, short wCount, short value);

// for computing f(concat(S, 1 byte))
void fp1(usInt *fP, usInt *P, short wCount, char s);

// for computing f(concat(S, 4 bytes))
void fp4(usInt *fP, short wCount, short s1, short s2, short s3, short s4);

// return a random number in the range [0, 1]
double getArandom();

/***** Global Variables *****/
usInt **TA, **TB, **TC, **TD;

/***** Application Entry *****/
int main( int argc, char * argv[] ) {

  // setting up
  FILE* logFile;
  FILE* outFile;
  char* fName;
  short wCount, i, j, k;
  usInt* P;

  // open the log and output files to record statistics and results
  if ( (logFile = fopen("result.log", "a")) == NULL ) {
    fprintf( stderr, "Error opening log file, exiting...\n" );
    exit(1);
  };

  fName = argc > 2 ? strdup(argv[2]) : strdup("outfp.dat");
  if ( (outFile = fopen(fName, "w")) == NULL ) {
    fprintf( stderr, "Error opening output file, exiting...\n" );
    exit(1);
  };

  // retrieve the degree for P from parameter, default to 64
  // and generate the polynomial, write to log file
  k = argc > 1 ? atoi( argv[1] ) : 64;

  P = generateIrreduciblePoly(k, &wCount);

  fprintf(logFile, "\nThe polynomial is of degree %d\n", k);
  fprintf(logFile, "Irreducible polynomial used is ");
  for ( j = wCount - 1; j >= 0; j-- ) {
```

```
  for (i = 28; i >= 0; i -= 4) fprintf(logFile, "%X", P[j] >> i & 0xf);
  fprintf(logFile, " ");
}
fprintf(logFile, "\n");

// allocate memory for tables and generate tables
TA = (usInt **)malloc(256*sizeof(usInt*));
TA[0] = (usInt *)malloc(256*wCount*sizeof(usInt));

TB = (usInt **)malloc(256*sizeof(usInt*));
TB[0] = (usInt *)malloc(256*wCount*sizeof(usInt));

TC = (usInt **)malloc(256*sizeof(usInt*));
TC[0] = (usInt *)malloc(256*wCount*sizeof(usInt));

TD = (usInt **)malloc(256*sizeof(int*));
TD[0] = (usInt *)malloc(256*wCount*sizeof(usInt));

if((!TA)||(!TB)||(!TC)||(!TD)){
  fprintf(stderr, "Memory allocate error in main--1\n");
  exit(1);
}

if((!TA[0])||(!TB[0])||(!TC[0])||(!TD[0])){
  fprintf(stderr, "Memory allocate error in main--2\n");
  exit(1);
}

computTable(TA, TB, TC, TD, P, wCount);

// actual process
int n, len;
char pid[8];
char url[1001];
char url2[873];
usInt* fP = (usInt *)malloc(wCount*sizeof(usInt)); // for storing fingerprint

if(!fP) {
  fprintf(stderr, "Memory allocate error in main--3\n");
  exit(1);
}

// clean up for next round
pid[0]  = '\0';
url[0]  = '\0';
url2[0] = '\0';

// do some statisitics on time used
double totalTime = 0;
clock_t t0 = clock();
long record = 0;

// loop until the file is done
while ( scanf("\"%[^\"]\",\"%[^\"]\",\"%s\n", pid, url, url2) > 0 ) {
```

```
    url2[strlen(url2)-1] = '\0';
    strcat(url, url2);
    record++;

    for ( j = 0; j < wCount; j++ ) fP[j] = 0;

    len = strlen(url);
    n = len / 4;

    for ( j = 1; j <= n; j++ ) {
      fp4(fP, wCount, url[4*j-4], url[4*j-3], url[4*j-2], url[4*j-1]);
    }

    for ( j = 4*n; j < len; j++) {
      fp1(fP, P, wCount, url[j]);
    }

    // output the result
    for ( j = wCount - 1; j >= 0; j-- ) {
      fprintf(outFile, "%X ", fP[j]);
    }
    fprintf(outFile, "\n");

    // clean up for next round
    pid[0]  = '\0';
    url[0]  = '\0';
    url2[0] = '\0';

    if(record % 1000000 == 0){
       clock_t t1 = clock();
       totalTime = totalTime + (double)(t1-t0)/CLOCKS_PER_SEC;
       t0 = clock();
    }

  }

  clock_t t1 = clock();
  totalTime = totalTime + (double) (t1 - t0) / CLOCKS_PER_SEC;
  fprintf( logFile,
   "Total CPU time for %d fingerprinting = %f seconds\n", k, totalTime);

  fprintf( logFile, "\n***************************************\n" );

  free(P); free(fP);
  free(TA[0]); free(TB[0]); free(TC[0]); free(TD[0]);
  free(TA); free(TB); free(TC); free(TD);

  // close the files
  fclose(logFile);
  fclose(outFile);

  return 1;
}
```

```
/***** Methods *****/
usInt * generateIrreduciblePoly(short k, short *size) {
  // precondition
  if((k<0)||(k % 32 != 0)){
    cout<<"Error in Generating Irreducible Polynomial, k - value"<<endl;
    exit(1);
  }

  //Seed the random-number generator with current time so that
  //the numbers will be different every time we run.
  srand( (unsigned)time( NULL ) );
  double temp;
  for(int n=0; n<5; n++) {
  //I found the first generated random does not change,
  //so I choose the fifth one
    temp = getArandom(); //temp is [0, 1)
  }
  int r = (int) ( temp * (floor((k-1)/2.0)-1));
  r = 2*r + 1; // r is a random odd value within [1, k-2]

  usInt *A;//A array is going to store the poly. coefficents.
  A = (usInt *)malloc(k*sizeof(usInt)); //allocate memory
  if(!A) {
    cout<<"Memory allocate error for Irreducible Polynomial--1"<<endl;
    exit(1);
  }

  //initialize the array
  A[0] = 1; //A[k-1] = 1;
  short i;

  for(i=1; i<= k-1; i++) // should be i <= k - 1 instead of i < k - 1
     A[i] = 0;

  for(i=1; i<=r; i++){
     while(1){
         temp = getArandom(); //temp is [0, 1)
 short j = (short)(temp * (k-2)); // j is [0, k-2)

 if(A[j+1]==0){
    A[j+1] = 1;
    break;
 }
     }//end of while loop
  }//end of for i loop

  *size = k/32;

  usInt *W;
  W = (usInt *)malloc((*size)*sizeof(usInt)); //allocate memory
  if(!W) {
    cout<<"Memory allocatet error for Irreducible Polynomial--2"<<endl;
    exit(1);
```

```
  }

  for(short j=0; j<*size; j++){
     W[j] = 0;
     for(i=32*(j+1)-1; i>=32*(j+1)-32; i--)
W[j] = 2*W[j] + A[i];
  }
  return W;
}

void computTable
  (usInt **TA, usInt **TB, usInt **TC, usInt **TD, usInt *P, short wCount) {

  short i;
  for(i=1; i<256; i++) {
    TA[i] = TA[i-1] + wCount; //allocate address
    TB[i] = TB[i-1] + wCount; //allocate address
    TC[i] = TC[i-1] + wCount; //allocate address
    TD[i] = TD[i-1] + wCount; //allocate address
  }

  for(i=0; i<256; i++) {
    for(short j=0; j<wCount; j++) {
      TA[i][j] = 0; //initialize
      TB[i][j] = 0; //initialize
      TC[i][j] = 0; //initialize
      TD[i][j] = 0; //initialize
    }
  }

  for(i=0; i<256; i++) {
    short j;
    for(j=0; j < wCount; j++) {
      TD[i][j] = 0;
    }

    byteShift(TD[i], wCount, i);

    multiT8(TD[i], P, wCount);

    for(j=0; j < wCount; j++) {
      TC[i][j] = TD[i][j];
    }

    multiT8(TC[i], P, wCount);

    for(j=0; j < wCount; j++) {
      TB[i][j] = TC[i][j];
    }

    multiT8(TB[i], P, wCount);

    for(j=0; j < wCount; j++) {
      TA[i][j] = TB[i][j];
```

```
    }

    multiT8(TA[i], P, wCount);

  }
}

void multiT8(usInt *words, usInt *P, short size)
{
  int needXOR;
  short i, j;

  for(i=0; i<8; i++){

    // XOR is needed only when the MSB = 1
    needXOR = ((words[size-1] & 0x80000000) == 0x80000000);

    // left shift 1 bit
    longLeftShift(words, size, 1);

    if ( needXOR ){
      for(j=size-1; j>=0; j--){
        words[j] = words[j] ^ P[j];
      }
    }
  }
}

void byteShift(usInt *words, short len, short value)
{
  short i;
  usInt temp = value;
  words[len - 1] = value << 24;

  for(i = len - 2; i >= 0; i--){
      words[i] = 0x00000000;
  }
}

void longLeftShift(usInt *words, short len, short s)
{
  usInt temp;
  usInt mask = 0x00000000;
  short i;
  for (i = 0; i < s; i++)     //create a mask for s LSB
    mask = mask << 1 | 0x00000001;

  for (i = len - 1; i > 0; i--) {
    temp = words[i-1] >> (32 - s) & mask; // get a MSB from low
    words[i] = words[i] << s | temp; // set as s LSB in high
  }

  words[0] = words[0] << s; // shift the low word
}
```

```
void fp1(usInt *fP, usInt *P, short wCount, char s) {

  usInt W;
  int needXOR;
  short i, j;

  for ( i = 0; i < 8; i++ ) {
    W = s & 0x80; // retrieve the MSB of s
    s = s << 1;   // shift for next round
    needXOR = ( (fP[wCount-1] & 0x80000000) == 0x80000000 );

    longLeftShift(fP, wCount, 1);

    fP[0] = fP[0] | (W >> 7);

    if ( needXOR ) {
      for( j = wCount - 1; j >= 0; j--) {
fP[j] = fP[j] ^ P[j];
      }
    }
  }
}

void fp4(usInt *fP, short wCount, short s1, short s2, short s3, short s4) {

  usInt W = 0;
  short i, j;
  unsigned short f1, f2, f3, f4;

  W = (((W | s1) << 8 | s2 ) << 8 | s3 ) << 8 | s4;

  f1 = (fP[wCount - 1] >> 24 ) & 0xff;
  f2 = (fP[wCount - 1] >> 16 ) & 0xff;
  f3 = (fP[wCount - 1] >> 8 ) & 0xff;
  f4 = fP[wCount - 1] & 0xff;

  for( j = wCount - 1; j > 0; j--) {
    fP[j] = fP[j-1] ^ TA[f1][j] ^ TB[f2][j] ^ TC[f3][j] ^ TD[f4][j];
  }
  fP[0] = W ^ TA[f1][0] ^ TB[f2][0] ^ TC[f3][0] ^ TD[f4][0];

}

double getArandom() {
  return rand()/(double)RAND_MAX;
}
```

7.2 Source Codes for “url2pid”

The source codes are contained in a file named “functions.c”.

```
#include <string.h>
#include <math.h>
#include <sqludf.h>

double hash1(char *str) {
/*
  Hash function for chaning urls into PIDs.
*/
  double val = 0;
  int i = 0;
  char *ptr,*ptr2;

  ptr = str;
  if (*str) ptr++;
  ptr2 = ptr;
  if (*ptr) ptr2++;
  while (*str) {
    val += pow(10,i%12)*(pow(abs(*str),3) + (*ptr)*(*ptr) + abs(*ptr2));
    str++;
    ptr++;
    if (*ptr2) ptr2++;
    i++;
    val = fmod(val,pow(10,18));
  }

  return val;
}

double hash2(char *str) {
/*
  Secondary hash function for turning urls into PIDs.
*/
  double val = 0;
  double pro = 1;
  int i;

  while (*str) {
    for (i=1;(i<=5) && (*str);i++) {
      pro *= abs(*str);
      str++;
    }
    val += pro;
    val = fmod(val,pow(10,17));
    pro = 1;
  }
  return val;
}

double get_pid(char *url,char *pid, int n) {
```

```
/*
  Takes the url and generates a pid using a hash function. The pid is
  returned through the parameter as a string. The double value of the
  pid is also returned as the return value.
*/

  char temp[16];
  double val = 0;
  double val2,dbl;
  int i = 0;

  while (n > 1) {
    val += hash2(url);
    n--;
  }

  val += hash1(url);
  val = fmod(val,pow(93,8));  /* 93^8 combinations, want the number in
less than
 that */
  val2 = val;

  for (i=7;i>=0;i--) {
    dbl = fmod(val,93);
    dbl = floor(dbl);
    pid[i] = (int)dbl;
    pid[i] += 33;
    if (pid[i] == '"') pid[i] = 126;  /* pid's can't contain " */
    val /= 93;
  }
  pid[8] = 0;

  return val2;
}

/* returns the first pids */
void url2pid1 (
  char *url,/* input */
  char *pid,/* output */
  short *nullUrl,/* indicator variable for the input */
  short *nullPid,/* indicator variable for the output */
  char *sqlstate,
  char *fnName,
  char *specificName,
  char *message
  )
{
  int n = 1;

  if (*nullUrl) {
    *nullPid = -1;
    return;
  }
```

```
  get_pid(url, pid, n);

  return;
}

/* returns a table of pids */
void url2pids (
  char *url,            /* input */
  char *pid,            /* output */
  short *nullUrl,       /* indicator variable for the input */
  short *nullPid,       /* indicator variable for the output */
  char *sqlstate,
  char *fnName,
  char *specificName,
  char *message,
  SQLUDF_SCRATCHPAD   *scratchpad,/* declared in sqludf.h */
  SQLUDF_CALL_TYPE    *calltype /* declared in sqludf.h */
  )
{
  int *pad = (int *)scratchpad->data;   /* scratchpad */

  switch (*calltype) {
    case SQL_TF_OPEN: *pad = 0; break;
    case SQL_TF_FETCH:
      while (*pad < 3) {
        (*pad)++;
        get_pid(url, pid, *pad);
        strcpy(sqlstate, "00000");
        return;
      }
      strcpy(sqlstate, "02000");
      break;
    case SQL_TF_CLOSE: break;
  }
}
```

Bibliography

[1] Michael O. Rabin. Fingerprinting by random polynomials. Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[2] Michael O. Rabin. Discovering repetitions in strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*. Springer-Verlag, 1985.

[3] Bill Cherowitzo. http://www-math.cudenver.edu/˜wcherowi/courses/finflds.html. webpage.

[4] Michael O. Rabin. Probabilistic algorithms in finite fields. In *SIAM Journal of Computing*, volume 9, pages 273–280, 1980.

[5] Andrei A. Broder. Some applications of Rabin's fingerprinting method. In A. De Santis R. Capocelli and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag, 1993.

[6] http://www.cs.yorku.ca/˜oz/hash.html. webpage.

[7] Bruno Preiss. http://www.pads.uwaterloo.ca/bruno.preiss/books/opus5/html /page220.html#eqnhashingstring. webpage.