Analysis of Permutation Polynomials over Finite Field as Verifiable Delay Function

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Anurag Jha
(111901010)

Boda Pranav Aditya
(111901018)

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Analysis of Permutation Polynomials over Finite Field as Verifiable Delay Function**" is a bonafide work of **Anurag Jha** (**Roll No. 111901010**) and **Boda Pranav Aditya** (**Roll No. 111901018**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Srimanta Bhattacharya

Assistant/Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

We would like to extend our thanks to our professor Srimanta Bhattacharya for giving us the chance to work on this project, and we would want to express our gratitude.
We appreciate his advice and unwavering support, which have aided and encouraged us throughout the project.

Abstract

In this report, we first defined the fields before moving on to finite fields.

Then, we defined permutation polynomials and offered numerous theorems and findings pertaining to these permutation polynomials. Additionally, we defined irreducible polynomials. After that, we went on to explain the various codes we had created for Sage Math, including those for generating permutation polynomials, listing out all binomial permutation polynomials, and listing out all binomial permutation polynomials which also satisfy the property that f(x) + x is also a permutation polynomials.

Then we have devised a strategy for verifying permutation polynomial over finite field.

Then we have explained VDF and different approaches to measure code execution time.

Finally we have tried measuring the time elapsed in verifying permutation polynomial efficiently.

Contents

Chapter 1

Introduction

A field is a collection of elements that may perform addition and multiplication operations that are equivalent to those found in the real number system. **Finite fields** as the name suggests consists of finite number of elements. The cardinality of a field is called the order of field. The order of finite field is finite and are prime numbers or power of primes.
i.e. order of finite field = p or p^k [where p is prime number and k is a positive integer]
One very common example of finite field is integer mod p.
The characteristic of field is defined as the number of times we need to add the copies to get 0, for a field of order p^k, this value will be p.

If we have a finite field of order q, a polynomial $f \in F_q[x]$ is called a **Permutation polynomial** of F_q if induced mapping $x \mapsto f(x)$ is a permutation of F_q.
$\forall$ permutation in field, there is a permutation function attached to it.

Chapter 2

Permutation Polynomials

Theorem: 1 *A polynomial f(x) = $\sum_{i=0}^{d} a_i x^i$ over finite field F is permutation polynomial if it induces a bijection from F to F.*

Theorem: 2 *Fermat's little theorem states that if p is a prime number, then for any integer a, the number a^p–a is an integer multiple of p.*
i.e. $a^p \equiv a \ mod \ p$

Fermat's little theorem gives one of the simplest permutation polynomials modulo a prime p: $f(x) = x^p$, satisfying that for all x belonging to integers, $f(x) \equiv$ x (mod p).

$f \in F_q[x]$ is a permutation polynomial of F_q if and only if: f is onto, f is one-to-one, PP

1. The function f is onto.
2. The function f is one-to-one.
3. $\forall a \in F_q$, the equation $f(x) = a$ has solution in F_q
4. $\forall a \in F_q$, the equation $f(x) = a$ has a unique solution in F_q

Theorem: 3 *Every linear polynomial of the form ax + b with over F_q having $a \neq 0$ is a permutation polynomial of F_q.*

Theorem: 4 *x^k is a permutation polynomial of F_q if and only if the $gcd(k, q-1) = 1$*

Theorem: 5 ***Irreducible Polynomial*** *is a polynomial that cannot be factored into product of two non-constant polynomials over the same field.*

$x^2 + 1$ is an example of a degree 2 polynomial over F_7. This is because if we consider the polynomial to be $(x + a)(x + b)$, we get $x^2 + (a + b)x + ab$, there are no two numbers a,b in mod 7 such that (ab) mod $7 == 1$ and $(a + b)$mod $7 == 0$.

We can use this polynomial to create extension field such as F_{49}.

$F_{49} = \frac{F_7[x]}{x^2+1}$

Here instead of $x^2 + 1$, we can use any other irreducible polynomial in F_7 as well.

$x^2 + 1$ is not an irreducible polynomial of degree 2 polynomial over F_2. This is because if we consider the polynomial to be $(x + 1)(x + 1)$, we get $x^2 + (1 + 1)x + 1$.

$x^2 + 2x + 1$ polynomial over F_2 is $x^2 + 0x + 1$.

Theorem: 6 *(**Hermite criterion**) [1] A polynomial $f(x) \in Fq[x]$ is a permutation polynomial of Fq if and only if following two conditions hold:*

1. *In Fq, f(x) has only 1 root ; and*
2. *for each integer t with $1 \leq t \leq q - 2$ and $t \neq 0$ mod p, the reduction of $f(x)^t$ mod $(x^q - x)$ has degree $\leq$ q - 2.*

Let us try to prove that f has only 1 root in Fq if and only if $f(x)^{q-1}$ mod $(x^q - x) = \sum_{i=0}^{q-1} {b_i}^t x^i$, where ${b_{q-1}}^{(t)} = -\sum_{c \in Fq} f(c)^t$.

If f has only j roots in Fq, then:

${b_{q-1}}^{(q-1)} = -\sum_{c \in Fq} f(c)^{q-1} = -(q - j) =$ j.

Since $0 \leq j \leq q - 1$, we have:

${b_{q-1}}^{(q-1)} = 1$ if and only if $j = 1$

The Hermite criterion is a very powerful method to identify if a polynomial is permutation polynomial or not and we have implemented the same criterion in our codes as well.

Chapter 3

Sage codes

We have written numerous pieces of code that account for different permutation polynomial features and facets. Sage Documentation

3.1 Given a polynomial check if it is permutation polynomial or not

The link for the code can be found here: code

3.2 Generate all the permutation polynomials of Fq where $q = 2^n$ and coefficients are permutation of q

The link for the code can be found here: code

3.3 Find all permutation polynomials f(x) of form $x^r + ax^s$ where f(x) + x is also a permutation polynomial

Here: b = all values of F, r = all values of F,s = 0 to r

```
1
```

```
x = PolynomialRing(RationalField(), 'x').gen()  #variable x
polynomialsSet = set()  #set to store polynomials

def generatePolynomials(n):

    for r in range(n):
        for s in range(r):
            for b in range(n):
                f=x^r + b*(x^s) #generate all possible binomials
                polynomialsSet.add(f)   #append binomial to set

def checkPermutationPolynomial(f,q):
    k=f.roots()
    if(len(k)!=1):
        return 0
    for i in range(1,q-1):
        p=(f^i)%(x^q - x)
        if(p.degree() >= q-1):
            return 0
    return 1

# Driver Code
if __name__ == "__main__":

    n = 2^2
    arr = [None] * n

    generatePolynomials(n)

    countPP=0
    permutationPolynomialList=[]

```

```
34      for p in polynomialsSet:
35          f = p + x
36          if(checkPermutationPolynomial(p,4)==1 and
    checkPermutationPolynomial(f,4)==1): #if both f(x) and f(x) + x is PP
37              countPP=countPP+1
38              permutationPolynomialList.append(p)
39
40      print(countPP)  #Print number of permutation polynomials
41      print(permutationPolynomialList)    #Print permutation polynomials
```

Chapter 4

Verifying Permutational Polynomial over Finite Field

To verify permutational polynomial over finite filed and to find the mapping for a particular input , we need irreducible polynomial .

Permutational Polynomial is in the form of binomial $x^r + ax^s$.

To find mapping for a polynomial x in the field over permutational polynomial , we need to substitute the x in permutational polynomial function and then find modulo with irreducible polynomial .

4.1 FUNCTIONS USED IN THE CODE

4.1.1 poly_binary

Function takes a polynomial in field 2^n and degree n as input and outputs binary string of n length .

If polynomial contains element of degree i then binary string at ith index will be 1 , else 0.

The polynomial input will be of form $x^r + x^s + x^t$

```
string poly_binary(string s, int n)
```

```
{   // function converts polynomial to binary string of length n starting
    from x^0
    string ans = "";
    for (int i = 0; i < n; i++)
        ans += '0';
    int ind = 0;
    while (ind < s.length())
    {
        int k = s.find('^', ind);
        int k1 = s.find('+', ind);
        if (k1 == string::npos)
            k1 = s.length();
        int i = stoi(s.substr(k + 1, k1 - k - 1));
        ans[i] = '1';
        ind = k1 + 1;
    }
    return ans;
}
```

4.1.2 binary_poly

Function takes binary string as input and coverts it into polynomial with starting index of binary string indicating x^0.

The Polynomial Output will be of form $x^r + x^s + x^t$

```
string binary_poly(string s)
{   // function converts binary to polynomial form
    int n = s.length() - 1;
    string ans = "";
    for (int i = s.length() - 1; i >= 0; i--)
    {
        if (s[i] == '1')
        {
```

```
            ans += "x^";
            ans += to_string(i);
            ans += '+';
        }
    }
    if (ans == "")
        return "0";
    return ans.substr(0, ans.length() - 1);
}
```

4.1.3 add

Function takes two binary strings which need to be added as input and outputs the binary string after addition.

Addition in finite field of 2^n is same as XOR.

Strings with unequal length are padded with 0 and added.

```
// function takes two strings and perform xor
string add(string s, string t)
{
    int l1 = s.length();
    int l2 = t.length();
    if (l1 != l2)
    {
        if (l1 < l2)
        {
            while (l1 < l2)
            {
                s += '0';
                l1++;
            }
        }
        if (l1 > l2)
```

```
        {
            while (l1 > l2)
            {
                t += '0';
                l2++;
            }
        }
    }
    int count = 0;
    for (int i = 0; i < s.length(); i++)
    {
        count = 0;
        if (s[i] == '1')
            count++;
        if (t[i] == '1')
            count++;
        if (count == 1)
            s[i] = '1';
        else
            s[i] = '0';
    }
    return s;
}
```

4.1.4 multiply

Function takes two binary strings which need to be multiplied as input and outputs the binary string after multiplication.

```
string multiply(string s, string t)
{
    int l1 = s.length();
    int l2 = t.length();
```

```
    string ans = "";
    for (int i = 0; i < l1 + l2 + 1; i++)
        ans += '0';
    for (int i = 0; i < l1; i++)
    {
        for (int j = 0; j < l2; j++)
        {
            if (s[i] == '1' && t[j] == '1')
            {
                if (ans[i + j] == '0')
                    ans[i + j] = '1';
                else
                    ans[i + j] = '0';
            }
        }
    }
    int ind = ans.length() - 1;
    while (ans[ind] == '0')
        ind--;
    return ans.substr(0, ind + 1);
}
```

4.1.5 mod

Function takes two binary strings s,i and we need to find s mod i. i is the irreducible polynomial. For this function the strings are reversed and while loop is run until length of string s is less than the length of irreducible polynomial.

To achieve this , for every while loop the starting sub string of length of irreducible polynomial is added with the irreducible polynomial and string s is accordingly updated.

```
string mod(string s, string i)
{
    string s_r = s;
```

```
    string i_r = i;
    reverse(s_r.begin(), s_r.end());
    reverse(i_r.begin(), i_r.end());
    int ind = 0;
    while (i_r[ind] == '0')
        ind++;
    i_r = i_r.substr(ind, i_r.length() - ind);

    int l = i_r.length();
    while (s_r.length() >= l)
    {
        string temp = s_r.substr(0, l);
        string remain = s_r.substr(l, s_r.length() - l);
        temp = add(temp, i_r);
        int k = 0;
        while (temp[k] == '0' && k < temp.length())
            k++;
        if (k != temp.length())
            temp = temp.substr(k, temp.length() - k);
        else
            temp = "";
        s_r = temp + remain;
    }
    reverse(s_r.begin(), s_r.end());
    return s_r;
}
```

4.1.6 decode_pp

Permutational polynomial is in the form of binomial $x^r + ax^s$. We need to find the value of r,s and a to find the mapping. Vector of string containing these values is returned as output.

```
vector<string> decode_pp(string pp)
{   // given a pemutational polynomial of form y^s + b y^r
    // function returns containing r,b,s
    vector<string> ans;
    int ind = 0;
    int find = 0;
    while (ind < pp.length())
    {
        if (find == 0)
        {
            int k = pp.find('^', ind);
            int k1 = pp.find('+', ind);
            ans.push_back(pp.substr(k + 1, k1 - k - 1));
            ind = k1 + 1;
            find++;
        }
        if (find == 1)
        {
            int k = pp.find('(', ind);
            int k1 = pp.find(')', ind);
            ans.push_back(pp.substr(k + 1, k1 - k - 1));
            ind = k1 + 1;
            find++;
        }
        if (find == 2)
        {
            int k = pp.find('^', ind);
            ans.push_back(pp.substr(k + 1, pp.length() - k - 1));
            break;
        }
    }
    return ans;
```

```
}
```

4.2 MAIN FUNCTION

4.2.1 solve

Function takes input polynomial x , irreducible polynomial , permutational polynomial and n as inputs . Outputs the mapping of polynomial x.

The polynomial x is substituted in permuational polynomial and modulo with irreducible polynomial.

```
string solve(string input, string irr, string pp, int n)
{
    // function takes input , irreducible and permutational polynomial and degree of field
    // returns the mapping in field by permutational polynomial

    string irr_b = poly_binary(irr, n);//irr expressed in form of y^r+b y^s
    string input_b = poly_binary(input, n);
    vector<string> pp_d = decode_pp(pp);
    int pp_r = stoi(pp_d[0]);
    int pp_s = stoi(pp_d[2]);
    string ppb = pp_d[1];
    string ppb_b = poly_binary(ppb, n);
    double max = floor(log2(pp_r));
    // for efficient caluculations all powers are caluculated in sums of powers of 2. ie 7=1+2+4
    vector<string> inputs;
    inputs.push_back(input_b);
    int temp = 0;
    while (max > 0)
    {
```

```
        max--;
        inputs.push_back(multiply(inputs[temp], inputs[temp]));
        temp++;
    }
    string input3;
    for (int i = 0; i < n; i++)
        input3 += '0';
    string input1 = input3;
    string input2 = input3;
    int ind = 0;
    while (pp_r > 0)
    {
        if (pp_r % 2 == 1)
        {
            input1 = add(input1, inputs[ind]);
        }
        pp_r /= 2;
        ind++;
    }
    ind = 0;
    while (pp_s > 0)
    {
        if (pp_s % 2 == 1)
        {
            input2 = add(input2, inputs[ind]);
        }
        pp_s /= 2;
        ind++;
    }
    string input4 = add(input1, multiply(input2, ppb_b));
    string ans = mod(input4, irr_b);
    return binary_poly(ans);
```

```
}
```

4.2.2 main

The main function takes command line inputs as n (degree of finite field) , irr (irreducible polynomial) and pp (permutational polynomial).

Code waits for user input polynomial to find the map and stops when user inputs -1.

solve function invokes when user gives input.

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

string solve(string input, string irr, string pp, int n);

int main(int argv, char **argc)
{
    int n = stoi(argc[1]); // degree in power of 2
    string irr = argc[2];  // irreducible polynomial
    string pp = argc[3];   // permutational polynomial
    cout << "Input :";
    string x;
    while (getline(cin, x))
    {
        if (x == "-1") // to break the input
            break;
        string ans = solve(x, irr, pp, n);
        cout << "Output: " << ans << endl;
        cout << "Input :";
    }
    return 0;
}
```

Chapter 5

Mesuring code execution time efficiently

5.1 VDF

Verifiable Delay Function (VDF) is a function whose computation requires running sequential steps and the result can be efficiently verified . VDF
VDF is always guaranteed with short **clock time** i.e. Any observer can verify the output in a very short amount of time. No matter how many processors the task is divided it is still guaranteed it takes the same amount of clock time to complete.
More formally VDF is made up of 3 algorithms:

1. **Setup(t, λ)** : This technique generates public parameters(pp) from delay parameter(t) and security parameter(λ). Along with other data needed for computation and evaluation, these public parameters define the domain and range of VDF.

2. **Eval(x, pp)** : This algorithm selects an input x from the domain. It outputs a value from the range (y). It also outputs π[short proof] optionally.

3. **Verify(π, x, y, pp** : efficiently verifies whether the output y is accurate given the input x.

VDF should be unique, sequential and efficiently verifiable.

We want to perform analysis of Permutation Polynomials over Finite Field as Verifiable Delay Function. To achieve this, we must create a method for precisely estimating the function's execution time. The several methods utilised for this are described in later sections.

5.2 Approaches to mesauring code execution time

Talk about various approaches, look at gfg and other websistes. Also write simple codes and show. Talk about why this is not very efficient. Understanding clock time vs. CPU time is necessary before we can comprehend various methods of measuring code execution time.

Wall time(clock time) is the amount of time that passed while the code was being executed. It is comparable to starting a stopwatch when we start an execution and stopping it when it is complete.

The length of time the CPU spent executing a program's instruction is known as CPU time.

5.2.1 time on linux

One of the simplest ways to measure code execution time on linux is by using the time command. If we run the executable with time command we get wall time and clock time of the code. The drawback of this method is that we cannot obtain the execution time of isolated parts of code. Here 'real' is wall time and 'user' is CPU time.

Fig. 5.1 time on simple code that takes average of million numbers

```
anurag@anurag:~/B.Tech/BTP$ time ./test
Average of million numbers between 0-10 is: 5.00211

real    0m0.027s
user    0m0.023s
sys     0m0.005s
```

5.2.2 clock()

Another method is to use time.h library and use the function clock(). We start measuring time by calling the clock() before the main code logic and when the main code logic finished, we call the clock() again. The difference in these variables divided by $CLOCKS\_PER\_SEC$ will give us the time elapsed. The output will differ based on the operating system i.e. it will give CPU time on linux, and wall time on windows as output.

5.2.3 clock_get_time()

We can improve this method by using clock_get_time. The code used in section 5.2.1 now integrated with clock_get_time() will look like:

```
#include<bits/stdc++.h>
#include<time.h>
using namespace std;

int sum(int a, int b){
  return a+b;
}

int main(){
  int s=0;
  srand(time(0));
  struct timespec begin, end;

  // Start measuring time
    clock_gettime(CLOCK_REALTIME, &begin);

  for(int i=0;i<1000000;i++){
```

```
18      int x = rand()%11;
19      s=sum(s,x);
20    }
21
22    float avg = s;
23    avg = avg/1000000;
24
25    // Stop measuring time and calculate the elapsed time
26      clock_gettime(CLOCK_REALTIME, &end);
27
28    long sec = end.tv_sec - begin.tv_sec;
29      long ns = end.tv_nsec - begin.tv_nsec;
30      double timeElapsed = sec + ns*1e-9;
31
32    cout<<"\nAverage of million numbers between 0-10 is: "<<avg<<endl;
33    printf("\nTime Elapsed: %.6f seconds.\n", timeElapsed);
34    return 0;
35  }
```

This Code measures the wall time, if we want to measure the CPU time, we just have to replace CLOCK_REALTIME with CLOCK_PROCESS_CPUTIME_ID.

5.3 Benchmarking code using RDTSC

The purpose of this section is to use the RDTSC instruction to calculate the number of clock cycles needed to run a certain piece of C/C++ code under Linux on a processor with a standard Intel architecture.

The basic goal is to count the number of clock cycles elapsed during code execution. This is highly helpful in the context of CPU benchmarks and code optimizations.

Each cycle is recorded by the timestamp counter on Intel CPUs. The per-core timestamp register of the devices stores the value of the timestamp counter, which the RDTSC may

access. Let us take a look at simple code snippet that uses RDTSC to measure code cycles elapsed.

```
unsigned cylesow, cycles_high, cycles_low1, cycles_high1;

asm volatile
(
    "RDTSC\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t": "=r" (cyclesHigh), "=r"
    (cyclesLow):: "%rax", "%rbx", "%rcx", "%rdx"
);
//function whose time has to be measured comes here
asm volatile
(
    "RDTSC\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    "CPUID\n\t": "=r" (cyclesHigh1), "=r"
    (cyclesLow1):: "%rax", "%rbx", "%rcx", "%rdx"
);

start = ( ((uint64_t)cyclesHigh << 32) | cyclesLow );
end = ( ((uint64_t)cyclesHigh1 << 32) | cyclesLow1 );
uint64_t timeTaken = end - start;
printf("\n function execution time %d times is %d clock cycles\n", n,
   timeTaken);
```

The low-order 32 bits of the timestamp register are loaded into EAX and the high-order 32 bits are loaded into EDX by the RDTSC instruction. The register value is reconstructed and saved into a local variable using a bitwise OR. We can call our function whose time we need to measure. In order to determine how many clock cycles have passed since the initial read, read the timestamp register again (RDTSC). At the specific timings of the

RDTSC calls, start and finish had saved the timestamp register values. The difference in these values can be used to compute the clock cycles elapsed.

It's important to keep in mind that most Intel CPUs allow code to be performed out of sequence when benchmarking programmes that use the RDTSC instruction. The CPUID instruction must come before both RDTSC instructions in order to prevent this out-of-order execution. This forces the CPU to execute all of the previous instructions in the code before the program may proceed. One drawback of using this is that a significant amount of variation is indissolubly tied to the execution of the CPUID instruction. This implies that we give up measurement resolution when utilising CPUID in order to guarantee instruction serialisation.

Additionally, we must discover a mechanism to guarantee CPU exclusivity by turning off preemption and hard interrupts.

5.4 Comparison of different approaches

To compare different approaches stated so far, I have wriiten a code. The link for the same can be found here: Benchmarking_comparison

The output obtained is as follows:

```
FOR 1048576 iterations, the results are as follows:

For CLOCK_PROCESS_CPUTIME_ID:
Time taken = 1648.124258 msec
Time taken per call = 1.571774 micro sec

For CLOCK_THREAD_CPUTIME_ID:
Time taken = 1579.306167 msec
Time taken per call = 1.506144 micro sec

For CLOCK_REALTIME:
Time taken = 40.119150 msec
Time taken per call = 0.038261 micro sec

For CLOCK_MONOTONIC:
Time taken = 41.616828 msec
Time taken per call = 0.039689 micro sec

For RDTSC:
Time taken = 79.977108 msec
Time taken per call = 0.076272 micro sec

Time taken for counter using get_time() = 114.995980 msec
Time taken for counter using rdtsc = 114.000000 msec
```

5.5 Improvements in bench-marking

Using the RDTSCP command is one of the enhancements. Before reading the counter, the RDTSCP command waits for all preceding instructions to complete execution. However, additional instructions in the source code that follow the RDTSCP may start running before the read operation is completed, causing contamination of the code being measured. The LFENCE command is another option. Each load-from-memory instruction that was delivered before to the LFENCE command undergoes a serialising procedure. Specifically, LFENCE does not run until all preceding instructions have finished locally, and no subsequent instruction starts execution until LFENCE completes.
Another approach is to use CPUID before the first RDTSC instruction and to use RDTSCP instead of second RDTSC and to use CPUID after the 2 move instructions.

This code prevents instructions above and below the RDTSC instruction from being executed out of sequence by the first CPUID. Due to the fact that CPUID is before RDTSC, it has no impact on the measurement. RDTSCP confirms the completion of necessary code execution. The last CPUID call ensures that the barrier is implemented once more, preventing any CPUID-prior instruction from being executed before. By doing this, calling CPUID in between reads is avoided and this ensures that variance introduced by CPUID does not affect the benchmarking.

The improved code looks like:

```
asm volatile
    (
        "CPUID\n\t"
        "LFENCE\n\t"
        "RDTSC\n\t"
        "mov %%edx, %0\n\t"
        "mov %%eax, %1\n\t": "=r" (cycles_high), "=r"
        (cycles_low):: "%rax", "%rbx", "%rcx", "%rdx"
    );
```

```
//function whose time is to be measured
asm volatile
(
    "RDTSCP\n\t"
    "LFENCE\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    "CPUID\n\t": "=r" (cycles_high1), "=r"
    (cycles_low1):: "%rax", "%rbx", "%rcx", "%rdx"
);
//Now measure the clock cycles elapsed in a similar way as before.
```

References: intel_manual

Chapter 6

Conclusion and Future Work

Now that we have a method to verify permutation polynomial over finite field(section 4) and a method to measure time of execution efficiently(section 5), the next step is to combine the results.

We have written an integrated code that verifies permutation polynomial over finite field and provides accurate description of the time elapsed in the execution process.

The link for the code can be found here: code

As part of future work, further research on improving time measurement by finding ways to disable hardware interrupts and preemption can be studied.

References

[1] C. J. Shallue, "Permutation polynomials of finite fields," 2012. [Online]. Available: https://arxiv.org/abs/1211.6044