Analysis of Permutation Polynomials over Finite Field using ipp-crypto library

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 using ipp-crypto library**" 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 express our sincere gratitude to my project guide, Srimanta Bhattacharya, for their invaluable guidance, motivation, and constant support throughout the course of our project. His insightful comments, suggestions, and feedback have been instrumental in shaping our understanding of the subject matter and refining the quality of our work. His unwavering support and encouragement have been a source of inspiration and motivation for us. We are extremely grateful to sir for his guidance and expertise and for the invaluable learning experience he has provided us with."

Abstract

The project aims to explore various aspects of finite field arithmetic supported by the ipp-crypto library and their applications in analyzing permutation polynomials.

The project begins with an introduction to big number arithmetic followed by a detailed study of prime number functions in ippcp. Next, finite field arithmetic is introduced, and different functions useful in initializing finite field context and arithmetic operations are explored. Extension field functions are also discussed in detail.

The project involves developing code that supports various properties of permutation polynomials to facilitate their analysis. In addition, the project evaluates the effectiveness of the developed code by comparing it with the previously used Sage code from the previous semester and plotting the corresponding time values. This project provides a comprehensive understanding of permutation polynomials in finite fields and showcases the practical usefulness of the ipp-crypto library in their analysis.

Contents

Chapter 1

Introduction

1.1 Project aim

Verifying permutation polynomials in extended fields is crucial for ensuring the security of cryptographic applications especially VDF, and efficient verification methods are necessary for scalability and practical implementation. The main aim of the project is to use Intel IPP crypto library to optimize analysis of permutation polynomials.

1.2 Previous Work

The progress made on the project in previous semester include:

- Studying Finite Field Arithmetic:
  - 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 field consists of finite number of elements.
- Studied the order of a field and characteristic of a field and their applications.

- Studied permutation polynomials:
  - Given 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.
  - Established and verified various theorems including:
    - $f \in F_q[x]$ is a permutation polynomial of F_q if and only if: f is onto, f is one-to-one, $f(x) = a$ has solution for all a, unique solution.
    - Every linear polynomial of the form ax + b with over F_q having $a \neq 0$ is a permutation polynomial of F_q.
    - x^k is a permutation polynomial of F_q if and only if the $gcd(k, q-1) = 1$
- Studied Irreducible polynomials and their applications.
- Studied and verified Hermite Criterion and established its importance to verify permutation polynomials. 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) polynomial has single 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.
- Wrote numerous pieces of code that account for different permutation polynomial features and facets.
  - Given a polynomial check if it is permutation polynomial or not. code
  - Generate all the permutation polynomials of Fq where $q = 2^n$ and coefficients are permutation of q. code
  - Find all permutation polynomials f(x) of form $x^r + ax^s$ where f(x) + x is also a permutation polynomial.

- Verified Permutational Polynomial over Finite Field : To find mapping for a polynomial x in the field over permutational polynomial , we wrote a code that substitutes the x in permutational polynomial function and then find modulo with irreducible polynomial. code
- Studied VDF(Verifiable Delay Function)
- Tested and compared different approaches to measure execution time of code efficiently. These include:
  - time command on linux
  - clock()
  - clock_get_time()
  - benchmarking using RDTSC The code for evaluation of different approaches: Benchmarking_comparison
- Tried improving the benchmarking using RDTSCP, CPUID, and LFENCE instructions and compared results.
- Produced a piece of code that verifies permutation polynomial over finite field and benchmarks the execution process efficiently.

. The link for the code can be found here: code

Chapter 2

Intel Integrated Performance Primitives Cryptography

2.1 ipp-crypto

Intel's Integrated Performance Primitives Programmers can use a collection of cryptographic functions provided by the software library Cryptography (Intel IPPCP). It is a part of the larger Intel Integrated Performance Primitives (Intel IPP) library, which provides computer programmes with features that are enhanced for a range of multimedia and signal processing applications. It provides kernel mode compatibility, thread safe design and security.

IPP Cryptography offers a wide range of cryptographic algorithms, including key exchange protocols, symmetric and asymmetric encryption, message digests, and digital signatures. For high-speed cryptographic tasks, the library leverages hardware acceleration where it is available and is optimised for Intel Processors. It works with a range of operating systems and development environments, including Windows, Linux, macOS, and Android.

The ipps prefix is attached to the functions. This makes it easier to distinguish between the functions of the Intel IPP interface and the other functions.

2.2 Building Intel IPP cryptography

- Intel C++ compiler is one of the essential components for creating Intel IPP cryptography and of the oneAPI toolkit. Intel oneAPI is a powerful toolkit for developers looking to create high-performance applications that can run on a variety of hardware architectures.
- clone the Intel IPP Cryptography repository from github.
- Set the appropriate environment variables to point to the location of the Intel IPP library and the compiler.
- Run the appropriate build script based on the platform.
- Include the appropriate header files and link against the "ippcrypto" library.
- Use CMake as an alternative technique to discover the IPP crypto library that has been installed on the system. Variables like IPPCRYPTO_ROOT_DIR, IPPCRYPTO_INCLUDE_DIRS, IPPCRYPTO_LIBRARIES will be defined if it is discovered.

```
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build$ make all
[  6%] Built target example_sms4-128-cbc-decryption
[ 18%] Built target example_rsa-3k-pss-sha384-type1-signature
[ 25%] Built target example_aes-256-ctr-decryption
Scanning dependencies of target example_aes-256-ctr-encryption
[ 28%] Building CXX object CMakeFiles/example_aes-256-ctr-encryption.dir/aes/aes-256-ctr-encryption.cpp.o
[ 31%] Linking CXX executable example_aes-256-ctr-encryption
[ 31%] Built target example_aes-256-ctr-encryption
[ 43%] Built target example_rsa-1k-oaep-sha1-type2-decryption
[ 56%] Built target example_dsa-dlp-sha-1-verification
[ 68%] Built target example_dsa-dlp-sha-256-verification
[ 81%] Built target example_rsa-1k-oaep-sha1-encryption
[ 87%] Built target example_sms4-128-cbc-encryption
[100%] Built target example_rsa-1k-pss-sha1-verification
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build$ ./example_aes-256-ctr-encryption
ippCP AVX2 (l9) 2021.7.0 (11.5 ) (-)
Features supported by CPU       by Intel® Integrated Performance Primitives Cryptography
----------------------------------------
  ippCPUID_MMX        = Y       Y       Intel® Architecture MMX technology supported
  ippCPUID_SSE        = Y       Y       Intel® Streaming SIMD Extensions
  ippCPUID_SSE2       = Y       Y       Intel® Streaming SIMD Extensions 2
  ippCPUID_SSE3       = Y       Y       Intel® Streaming SIMD Extensions 3
  ippCPUID_SSSE3      = Y       Y       Supplemental Streaming SIMD Extensions 3
  ippCPUID_MOVBE      = Y       Y       The processor supports MOVBE instruction
  ippCPUID_SSE41      = Y       Y       Intel® Streaming SIMD Extensions 4.1
  ippCPUID_SSE42      = Y       Y       Intel® Streaming SIMD Extensions 4.2
  ippCPUID_AVX        = Y       Y       Intel® Advanced Vector Extensions (Intel® AVX) instruction set
  ippAVX_ENABLEDBYOS  = Y       Y       The operating system supports Intel® AVX
  ippCPUID_AES        = Y       Y       Intel® AES instruction
  ippCPUID_SHA        = N       N       Intel® SHA new instructions
  ippCPUID_CLMUL      = Y       Y       PCLMULQDQ instruction
  ippCPUID_RDRAND     = Y       Y       Read Random Number instructions
  ippCPUID_F16C       = Y       Y       Float16 instructions
  ippCPUID_AVX2       = Y       Y       Intel® Advanced Vector Extensions 2 instruction set
  ippCPUID_AVX512F    = N       N       Intel® Advanced Vector Extensions 512 Foundation instruction set
  ippCPUID_AVX512CD   = N       N       Intel® Advanced Vector Extensions 512 Conflict Detection instruction set
  ippCPUID_AVX512ER   = N       N       Intel® Advanced Vector Extensions 512 Exponential & Reciprocal instruction set
  ippCPUID_ADCOX      = Y       Y       ADCX and ADOX instructions
  ippCPUID_RDSEED     = Y       Y       The RDSEED instruction
  ippCPUID_PREFETCHW  = Y       Y       The PREFETCHW instruction
  ippCPUID_KNC        = N       N       Intel® Xeon Phi™ Coprocessor instruction set
```

Chapter 3

Big Number Arithmetic

One of the fundamental aspects of modern cryptography is the use of large prime numbers in encryption and decryption. When two parties want to communicate securely, they use a mathematical algorithm to encode their messages using a large prime number, or a set of prime numbers. These prime numbers are used to create a key that is shared between the two parties, and this key is used to encrypt and decrypt messages.

Big number arithmetic is necessary for handling these large prime numbers and for performing the calculations involved in generating and using encryption keys. Without this mathematical tool, it would be impossible to create the complex encryption schemes that are used to protect sensitive data in today's digital world.

3.1 Big number functions

3.1.1 ippcpInit_BN()

ippcpInit_BN() is a function that initializes the Big Number arithmetic functionality in the IPP Cryptography library. This function creates the internal data structures needed to execute arithmetic operations on big numbers as well as allots memory for the (BN) context. Example code for ippcpInit_BN() would look like:

```
IppsBigNumState *bn; // Declare a BN variable
Ipp32u bnData[] = { 0x12895678, 0x9abbaef0 };
int bnSize = sizeof(bnSata) / sizeof(bnData[0]);
// Allocate memory for the BN variable
bn = ippsBigNumAlloc(bnSize);
if (bn == NULL) {
    // error Handling
}
// Set the BN variable to the desired value
IppStatus status = ippsSet_BN(bnData, bnSize, bn);
if (status != ippStsNoErr) {
    // error Handling
}
// Use the BN variable for further cryptographic operations
```

3.1.2 ippsSet_BN()

ippsSet_BN() is a function that can be used to set a big number to a specific value. Example code for *ippcpInit_BN*() would look like:

```
IppsBigNumState *bn; // Declare a BN variable
Ipp32u bnData[] = { 0x12895678, 0x9abbaef0 };
int bnSize = sizeof(bnSata) / sizeof(bnData[0]);
// Allocate memory for the BN variable
bn = ippsBigNumAlloc(bnSize);
if (bn == NULL) {
    // error Handling
}
// Set the BN variable to the desired value
IppStatus status = ippsSet_BN(bnData, bnSize, bn);
if (status != ippStsNoErr) {
    // error Handling
}
// Use the BN variable for further cryptographic operations
```

The size of the BN in bytes is calculated using the *sizeof*() operator and passed to the function as *bn_size*. ippsBigNumAlloc() allocates memory for the BN variable.

3.1.3 SetOctString_BN()

ippsSet_BN() is a function that can be used convert a string to a big number. Example code:

```
int size = (sizeof(value)-1+3)/4;
IppsBigNumState* pBN = New_BN(size);
ippsSetOctString_BN(value, sizeof(value)-1, pBN);
cout<<"Big Number is: "<<pBN;
```

3.2 Arithmetic functions

ippcp supports various arithmetic functions of big numbers. The major functions are:

- **cmp_BN** : This function is used to compare 2 big numbers. It returns IS_ZERO, GREATER_THAN_ZERO, LESS_THAN_ZERO based on the input big numbers.
- **CmpZero_BN** : This function returns IS_ZERO, GREATER_THAN_ZERO, LESS_THAN_ZERO based on the input data field.
- **Add_BN** : Returns addition result of 2 big numbers.
- **Sub_BN** : Returns subtraction result of 2 big numbers.
- **Mul_BN** : Multiples 2 big numbers
- **MAC_BN_I** : Accumulates the results of the multiplication of the first two integer big numbers to the third integer big number.
- **Div_BN** : Return a quotient and a reminder when one big number is divided by other.
- **Mod_BN** : The input integer big number's modular reduction is calculated with respect to the provided modulus.

- **Gcd_BN** : Computes GCD of 2 big numbers.
- **ModInv_BN** : With respect to the modulus defined by another positive integer big number, this function calculates the modular inverse of the given big number. i.e. for a given big number e and modulus integer m, it computes a number n such that n * e = 1 mod m.

```
IppStatus ippsCmp_BN(const IppsBigNumState *x, const IppsBigNumState *y, Ipp32u *result);

IppStatus ippsCmpZero_BN(const IppsBigNumState *x, Ipp32u *result);

IppStatus ippsAdd_BN(IppsBigNumState *x, IppsBigNumState *y, IppsBigNumState *result);

IppStatus ippsSub_BN(IppsBigNumState *x, IppsBigNumState *y, IppsBigNumState *result);

IppStatus ippsMul_BN(IppsBigNumState *x, IppsBigNumState *y, IppsBigNumState *result);

IppStatus ippsMAC_BN_I(IppsBigNumState *x, IppsBigNumState *y, IppsBigNumState *result);

IppStatus ippsDiv_BN(IppsBigNumState *x, IppsBigNumState *y,
IppsBigNumState * quotient, IppsBigNumState *remainder);

IppStatus ippsMod_BN(IppsBigNumState *x, IppsBigNumState *y, IppsBigNumState *result);

IppStatus ippsGcd_BN(IppsBigNumState *x, IppsBigNumState *y, IppsBigNumState *result);

IppStatus ippsModInv_BN(IppsBigNumState *n, IppsBigNumState *m, IppsBigNumState * result);
```

Sample code that implements these big number functions can be found here : code

3.3 Big number class

Big number class is also supported by IPPCP. It is a data structure that supports integer arithmetic operations with any level of precision. This class is implemented as part of the multiple precision arithmetic (MPA) functionality in the library.

The big number class in IPP Cryptography is an important component of the library, providing essential functionality for cryptographic algorithms that require large integers, such as RSA and elliptic curve cryptography.

The header file for the BigNumber class can be found here : code

Chapter 4

Prime numbers arithmetic using IPPCP

Primes are important in cryptography because they provide a high degree of security for cryptographic algorithms. Prime numbers are challenging to factor into their component prime components because of their distinctive mathematical characteristics. Many encryption and decryption techniques used in modern cryptography are founded on this characteristic.

4.1 IPPCP functions for prime number generation

- **PrimeGetSize** : The size needed to set up the IppsPrimeState context is returned by this function.
- **PrimeInit** : IppsPrimeState context is initialized based on the user-supplied memory pointer.
- **PrimeGen_BN** : The function generates a random prime number using the BigNum (BN) library. The PrimeGen_BN function generates a random number of the specified bit size and tests it for primality using the Miller-Rabin primality test. The Miller-

Rabin test is a probabilistic primality test that uses random numbers to test whether a number is prime with a certain level of confidence.

- **PrimeTest_BN** : The function is used for primality testing of large integers as BN objects. The PrimeTest_BN function performs a certain number of Miller-Rabin tests on the input integer to ascertain its primality. The input parameter "nTrials" controls how many tests are run. By performing many Miller-Rabin tests, the possibility of a false positive result can be reduced to any desired level.

4.2 Code for prime generation

```
1
2  IppsBigNumState* PrimeGenerationSample(int primeSize){
3
4     int maxBitSize = 256;
5     bool chk=true;
6     while(chk){
7
8        //Prime generator
9        int ctxSize;
10       ippsPrimeGetSize(maxBitSize, &ctxSize);
11       IppsPrimeState* p = (IppsPrimeState*)( new Ipp8u [ctxSize] );
12       ippsPrimeInit(maxBitSize, p);
13
14       //Default random generator
15       ippsPRNGGetSize(&ctxSize);
16       IppsPRNGState* pRand = (IppsPRNGState*)(new Ipp8u [ctxSize] );
17       int seedBits = 160;
18       ippsPRNGInit(seedBits, pRand);
19
20       Ipp32u result;
21       // generate prime of given primeSize
```

```
22        BigNumber primeGenerated(0, primeSize/8);
23        int nTrials = 50;
24
25        while( ippsPrimeGen_BN(primeGenerated, primeSize, nTrials, p,
    ippsPRNGen, pRand) != ippStsNoErr );
26
27        ippsPrimeTest_BN(primeGenerated, nTrials, &result, p, ippsPRNGen,
    pRand);
28
29        if(result != IPP_IS_PRIME) {
30           cout <<"Primality NOT confirmed\n";
31        }
32        else{
33           cout <<"Primality  confirmed\n";
34           chk=false;
35           cout<<"Prime is: "<<primeGenerated<<endl;
36           return primeGenerated;
37        }
38     }
39     return NULL;
40  }
```

Here, we use PrimeGetSize to first set up the PrimeState context. After that, we used PrimeInit to allocate memory and set up the prime state context. The pseudorandom generation functions are then used for the pseudo random generator. We next call the function PrimeGen_BN to generate the probable prime of the provided bitlength, check the status, and continue the process till there is no error. Then we use ippsPrimeTest_BN, which confirms primality using the Miller-Rabin method, to determine the primality of the created number.

The finite field arithmetic procedures employing IPP cryptography will heavily rely on the prime number arithmetic and use it as the main input.

Chapter 5

Finite Field Arithmetic using IPPCP

The finite field arithmetic functions uses ***IppsGFpState*** to store data of the finite field and ***IppsGFpElement*** to store data of finite elements .

5.1 Finite Field Initialization

Prime ,the prime number used to initialize the finite field is generated using the function ***PrimeGenerationSample*** which takes ***primeBitSize*** (user defined) as input .

Next step is to initialise the context of a prime finite field GF(q). There are several ways to initialize a context of a finite field.

5.1.1 GFpInitFixed

The function initializes pGF associated with the IppsGFPState and sets up the value of the GF(q) modulus to the chosen method.

Syntax: *IppStatus ippsGFpInitFixed(int primeBitSize, const IppsGFpMethod* method, IppsGFpState* pGF);*

Code:

```
int primeSize;
int ctxSize = primeSize;
IppsGFpState *pGF = (IppsGFpState *)(new Ipp8u[ctxSize]);
if (NULL == pGF)
{
    printf("ERROR: Cannot allocate memory (%d bytes) for pGF context\n", ctxSize);
    return NULL;
}
const IppsGFpMethod* method = ippsGFpMethod_p192r1();
IppStatus stat = ippsGFpInitFixed(primeSize, method,pGF);
if (stat == ippStsNoErr)
    printf("No Error \n");
if (stat == ippStsNullPtrErr)
    printf("Error if any of specified pointers is NULL \n");
if (stat == ippStsBadArgErr)
    printf("Error if method is not a pointer to an implementation of
    prime finite field or does not correspond to size of modulus q\n");
```

5.1.2 GFpInitArbitary

The function initializes pGF associated with the IppsGFPState and sets the GF(q) modulus to the value specified by Prime. This function uses ippsGFpMethod_pArb() or other pre-defined method to get an implementation of the finite field arithmetic.

Syntax: *IppStatus ippsGFpInitArbitrary(const IppsBigNumState* pPrime, int primeBitSize, IppsGFpState* pGF);*

Code:

```
int primeSize=60;
IppsBigNumState *Prime = PrimeGenerationSample(primeSize);
int ctxSize = primeSize;
IppsGFpState *pGF = (IppsGFpState *)(new Ipp8u[ctxSize]);
if (NULL == pGF)
{
    printf("ERROR: Cannot allocate memory (%d bytes) for pGF context\n", ctxSize);
    return NULL;
}
IppStatus stat = ippsGFpInitArbitrary(Prime, primeSize, pGF);
cout<<ippcpGetStatusString(stat)<<"\n";
```

5.1.3 GFpInit

The function initializes the pGF context parameter with the values of the input parameters Prime, primeBitSize and method. The three parameters have to be compatible with each other.

Syntax: *IppStatus ippsGFpInit(const IppsBigNumState* pPrime, int primeBitSize, const*

IppsGFpMethod method, IppsGFpState* pGF);*

Code:

```
int primeSize=60;
IppsBigNumState *Prime = PrimeGenerationsample(primeSize);
int ctxSize = primeSize;
IppsGFpState *pGF = (IppsGFpState *)(new Ipp8u[ctxSize]);
if (NULL == pGF)
{
    printf("ERROR: Cannot allocate memory (%d bytes) for pGF context\n", ctxSize);
    return NULL;
}
const IppsGFpMethod* method = ippsGFpMethod_p192r1();
IppStatus stat = ippsGFpInit(Prime, primeSize,method, pGF);
cout<<ippcpGetStatusString(stat)<<"\n";
```

NOTE

- If **method == NULL**, then the behavior of GFpInit() is similar to that of **GFpInitArbitrary()**.
- If **Prime == NULL**, then the behavior of GFpInit() is similar to that of **GFpInitFixed()**. (method must be an output from one of the GFpMethod functions with predefined modulus q, method and primeBitSize must be compatible with each other.)
- Other functions helpful in initialising context of a prime finite field include:
  - **GFpMethod**: This function is useful in the GFpInit and GFpInitFixed functions. Pointer to Implementation of arithmetic operations over the finite field is returned. It encapsulates the properties and operations of the field, and allows for efficient and secure computations on field elements. It supports multiple functions which implements arithmetic operation for specific prime including a function for arbitrary prime.
  - **GFpGetSize**: The function takes as input the degree of the finite field n, and optionally a pointer to an integer variable pSize that will receive the size of the prime modulus in bytes. It returns the size in bytes of the memory buffer required to hold a GFp context structure for a finite field of the specified degree.

Chapter 6

Element of finite field

IppsGFpElement is a data structure used in Intel's IPP library for performing operations on elements of a Galois Field over a prime modulus (GF(p)).

The IppsGFpElement functions provide a set of operations that can be performed on these GF(p) elements. Some of the common operations supported by these functions include:

- **Initialization**: Functions such as ippsGFpElementInit and ippsGFpElementInitFixed can be used to initialize GF(p) elements with either random or fixed values.
- **Arithmetic operations**: Functions such as ippsGFpElementAdd, ippsGFpElementSub, ippsGFpElementMul, and ippsGFpElementDiv can be used to perform arithmetic operations on GF(p) elements such as addition, subtraction, multiplication, and division.
- **Comparison operations**: Functions such as ippsGFpElementCmp can be used to compare GF(p) elements to determine whether they are equal or not.
- **Other operations**: Functions such as ippsGFpElementPower and ippsGFpElementIsZero can be used to perform other operations on GF(p) elements such as exponentiation and checking whether an element is zero.

6.1 GFp element functions:

- **ippsGFpElementGetSize**: This function returns the size in bytes required to hold a GFp element of a specified length. The function takes two arguments: the length of the GFp element in bits, and a pointer to a variable that will hold the size in bytes. The advantage of this function is that it allows you to allocate memory for GFp elements dynamically.

- **ippsGFpElementInit**: This function initializes a GFp element of a specified length with the value zero. The function takes 4 arguments: a pointer to the GFp element, and the length of the GFp element in bits, field element context pointer being initialised, finite field context pointer. All operations with this finite field element use the initialised IppsGFpElement context.

- **SetElement functions**: These function sets the value of a GFp element based on the type of function. The advantage of this function is that it allows you to set the value of a GFp element to a specified byte array without having to set each byte manually. The functions include:

  - **ippsGFpSetElement**: Sets the value of a GF(p) element using an array of big-endian bytes. Input consists of p_ele, p_gfp, Pointer to array storing field element and element length. The maximum length of an element of the finite field provided by the context p_gfp should not be greater than the maximum length of an element(p_ele).

  - **GFpSetElementOctString**: Sets the value of a GF(p) element using an octet string. Input consists of p_ele, p_gfp, octet string and its size. The string size should not exceed the element length.

  - **GFpSetElementRandom**: sets the value of a GF(p) element to a random value. Input consists of p_ele, p_gfp, pseudorandom generator and a pointer

to a parameter that is passed to the random number generation function. This function assumes that the random number generation function generates random numbers that are less than the prime p of the finite field.

  - **GFpSetElementHash**: Sets a finite field element to a value computed as a hash of an input message. The input parameters include: Pointer to the input message for which the hash needs to be computed, message length, finite field element pointer, pointer to context of finite field, and the ID of hash algorithm to be used. This function can be used to generate a random-looking value in the finite field from a given input message. The resulting value can be used as a private key, for example, in a public key cryptography algorithm.

- **ippsGFpCpyElement**: This function copies the value of one GFp element to another. The function takes two arguments: a pointer to the source GFp element, and a pointer to the destination GFp element.

- **GetElement**: This function gets the value of a GFp element and stores it in a specified byte array. The variation of this function include:

  - **GFpGetElement**: The function retrieves the value of the finite field element and stores it in the output buffer whose length is specified by 32-bit word. The value is returned in the default internal format used by the finite field arithmetic in the specified context (pGF).

  - **GFpGetElementOctString**: The function retrieves the value of the finite field element and stores it in the output buffer whose length is specified in bytes. The value is returned in the binary octet string format.

Overall, the IppsGFpElement functions provide a comprehensive set of operations for working with elements of a Galois Field over a prime modulus.

A simple cpp code that implements most of these functions:

```
IppsGFpElement *pR = (IppsGFpElement *)malloc(eleSize);
IppsGFpElement *pR2 = (IppsGFpElement *)malloc(eleSize);
IppsGFpElement *pRes = (IppsGFpElement *)malloc(eleSize);

printf("\nInitialising context of element of finite field\n");

printf("\nFor GFpElementInit: \n");
IppStatus eleInitStat = ippsGFpElementInit(pA, 1, pR, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";
if(eleInitStat == ippStsNoErr)
    cout<<"The context of an element of the finite field initialised.\n";

eleInitStat = ippsGFpElementInit(pA, 1, pR2, pGF);
//cout<<ippcpGetStatusString(eleInitStat)<<"\n";
eleInitStat = ippsGFpElementInit(pB, 1, pRes, pGF);
//cout<<ippcpGetStatusString(eleInitStat)<<"\n";

cout << "\nAssiging a Value to the element of finite field:";

printf("\nFor GFpSetElement: \n");
eleInitStat = ippsGFpSetElement(pA, 1, pR, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";
eleInitStat = ippsGFpSetElement(pB, 1, pR2, pGF);
eleInitStat = ippsGFpSetElement(pC, 1, pRes, pGF);

printf("\nFor GFpSetElementOctString: \n");
Ipp8u pStr[1]={'1'};
IppStatus eleSetStatStr = ippsGFpSetElementOctString(pStr, 1, pR, pGF);
cout<<ippcpGetStatusString(eleSetStatStr)<<"\n";

printf("\nFor GFpSetElementHash: \n");
Ipp8u pMsg[1]={'1'};
```

```
IppHashAlgId hashID = ippHashAlg_SHA256;
eleInitStat = ippsGFpSetElementHash(pMsg, 1, pR, pGF, hashID);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";

printf("\nFor GFpCpyElement: \n");
eleInitStat = ippsGFpCpyElement(pR, pR2, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";

printf("\nFor GFpGetElement: \n");
Ipp32u element_got[8];
eleInitStat = ippsGFpGetElement(pR, element_got, 8, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";
cout<<"Element obtained is: ";
//for(int i=0;i<8;i++)
    cout<<element_got[i]<<" ";
cout<<endl;

printf("\nFor GFpGetElementOctString: \n");
Ipp8u ele_string[32];
eleInitStat = ippsGFpGetElementOctString(pR, ele_string, 32, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";
```

6.2 Arithmetic element functions:

GFp element functions that deal with arithmetic operations include:

- **GFpCmpElement**: Returns a pointer that stores the result of compression of 2 finite field element. The results are based on:

```
#define IPP_IS_EQ (0) // elements are equal
#define IPP_IS_GT (1) // the first element is greater than the second one
#define IPP_IS_LT (2) // the first element is less than the second one
#define IPP_IS_NE (3) // elements are not equal
#define IPP_IS_NA (4) // elements are not comparable
```

- **GFpIsZeroElement**: The function returns 1 if the element is equal to zero and 0 otherwise.

- **GFpIsUnityElement**: The function returns 1 if the element is equal to one and 0 otherwise.

- **GFpConj**: The function computes the conjugate of an element of a finite field. The conjugate of an element a in a finite field F is defined as conj(a) = a$\hat{}$(q-1), where q is the order of the finite field. The conjugate of an element a in a finite field of characteristic p can be computed as conj(a) = $a^(p^(k-1))$, where k is the degree of the finite field.

- **GFpNeg**: The function calculates the additive inverse of the input element using the modulus of the finite field. It is used to perform operations such as subtraction or negation in a finite field.

- **GFpInv**: The function computes the inverse of a using the extended Euclidean algorithm, which finds two integers x and y such that $a * x + p * y = 1$. Since a and p are coprime, x is the multiplicative inverse of a (mod p).

- **GFpSqrt**: The function computes the square root of a using the Tonelli-Shanks algorithm, which is a probabilistic algorithm that works efficiently for fields of any size.

- **GFpAdd**: The function performs the addition of two elements in a finite field. In the case of a binary field, the addition is performed using the XOR operation. In the case of a prime field, the addition is performed using modular arithmetic.

- **GFpSub**: The function is used to subtract two elements of a finite field.

- **GFpMul**: The function is used to multiply two elements of a finite field. It uses the Karatsuba multiplication algorithm, which is an efficient multiplication algorithm for

polynomials with coefficients in a finite field. This algorithm is based on the divide-and-conquer technique. The algorithm has a better time complexity than the classic multiplication algorithm.

- **GFpSqr**: The function is used to calculate the square of a given element in a finite field. The calculation is performed using the Karatsuba algorithm. It requires fewer operations and is therefore faster than straightforward multiplication method.

Skeletal code representing working of these functions:

```
IppsGFpElement *pR   = (IppsGFpElement *)malloc(eleSize);
IppsGFpElement *pR2  = (IppsGFpElement *)malloc(eleSize);
IppsGFpElement *pRes = (IppsGFpElement *)malloc(eleSize);

int pResult;
printf("\nFor GFpIsZeroElement: \n");
eleInitStat = ippsGFpIsZeroElement(pR, &pResult, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";
cout<<"Res is: "<<pResult<<endl;

printf("\nFor GFpIsUnityElement: \n");
eleInitStat = ippsGFpIsUnityElement(pR, &pResult, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";
cout<<"Res is: "<<pResult<<endl;

printf("\nFor GFpNeg: \n");
eleInitStat = ippsGFpNeg(pR, pR2, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";

printf("\nFor GFpInv: \n");
eleInitStat = ippsGFpInv(pR, pR2, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";

printf("\nFor GFpSqrt: \n");
eleInitStat = ippsGFpSqrt(pR, pR2, pGF);
cout<<ippcpGetStatusString(eleInitStat)<<"\n";

eleInitStat3 = ippsGFpAdd(pR,pR2,pRes,pGF);
if (eleInitStat3 == ippStsNoErr) printf("\nAddition went fine\n");
cout<<ippcpGetStatusString(eleInitStat3)<<endl;

//similary for other arithmetic functions
```

Chapter 7

Extension fields

Initializing the context of an extension field and an individual finite field element comes after initialising the context of a prime finite field. We normally need to select an irreducible polynomial of degree m to initialise an extension field for use in ipp cryptography. To guarantee that the resulting field has desirable cryptographic qualities, such as a sizable number of elements and a high level of uniformity, this polynomial should be carefully chosen. Following the selection of the polynomial, the extension field can be created by specifying the field operations (addition, multiplication, and inversion) in terms of the binary polynomials that stand in for the field elements.

7.1 Extension field functions

Functions necessary for initialising the context of extension field:

- **GFpxMethod**: GFpxMethod is a structure defined in Intel IPP Cryptography library that represents the arithmetic methods of a finite field extension GF(p^d). It contains function pointers to the basic arithmetic operations for the underlying field GF(p), as well as methods for polynomial arithmetic over extended field. The GFpxMethod structure is used as an argument to the creation of an extended field context. The It provides a flexible way to define arithmetic operations for arbitrary

extension fields GF(p^d), without having to hard-code specific implementations for each field. It also allows for optimizations specific to the underlying field GF(p) to be used in the arithmetic operations, improving performance.

- **GFpxGetSize**: The GFpxGetSize function in Intel IPP is used to calculate the size of the memory that is required for the specified extension field. This function returns the size in bytes of the IppsGFpxState structure, which is required for the extension field. The advantage of using the GFpxGetSize function is that it ensures that the correct amount of memory is allocated for the extension field. This helps to prevent memory errors.

- **GFpxInitBinomial**: GFpxInitBinomial initializes a context for an extension field generated by an irreducible binomial over a prime field. This function computes the irreducible binomial polynomial used to define the extension field and initializes the context for the extension field. The function does not check the binomial's irreducibility. The degree of extension should lie in the range $[2, 7]$.

- **GFpxInit**: GFpxInit initializes a finite field of extension using the supplied field descriptor and element data. This function creates a new finite field object and initializes it with the an array of IppsGFpElement contexts representing coefficients of the field polynomial. The extenal degree should not exceed 7 and the leading coefficient of the polynomial is not required becuase it is a monic polynomial. Hence the numbers of terms in the pointer array $<$ external degree of the field.

- **GFpScratchBufferSize**: Scratch buffers are used to store intermediate results during cryptographic operations. These buffers are allocated dynamically at runtime and can be reused for different operations as needed. The size of the scratch buffer required for a particular operation can vary depending on a variety of factors, such as the size of the input data, the complexity of the algorithm being used, and the architecture of the system running the library.

7.2 Skeletal code representing working of these functions:

```
int pStateSizeInBytes;
IppStatus eleInitStatw = ippsGFpxGetSize(pGF1, deg, &pStateSizeInBytes);
//pGF1 is finite field context of GF(p)
if (eleInitStatw == ippStsNoErr) printf("\nippsGFpxGetSize OKK\n");

cout<<"Retreving the size of scratch buffer\n";
stat2 = ippsGFpScratchBufferSize(nExponenets, ExpBitSize, pGF1, &bufSize);
cout<<ippcpGetStatusString(stat2)<<"\n";

Ipp32u pX[pStateSizeInBytes];
IppsGFpElement *pP = (IppsGFpElement*) malloc(maxBitSize);
//initialise element of GF(p) field
eleInitStat = ippsGFpElementInit(pX, pStateSizeInBytes, pP, pGF1);
cout<<"For element initialisation: "<<ippcpGetStatusString(eleInitStat)<<endl;

//call the extension field method for arithmetic operations over extension field
const IppsGFpMethod* methodx = ippsGFpxMethod_com();
IppsGFpState *pGFpx = (IppsGFpState *)(new Ipp8u[pStateSizeInBytes]);
IppStatus extFieldInitStat = ippsGFpxInitBinomial(pGF1, deg, pP, methodx, pGFpx);

IppsGFpElement *px1 = (IppsGFpElement*) malloc(maxBitSize);
IppsGFpElement *px2 = (IppsGFpElement*) malloc(maxBitSize);

eleInitStat = ippsGFpElementInit(pX, 1, px1, pGF1);
cout<<ippcpGetStatusString(eleInitStat)<<endl;
eleInitStat = ippsGFpElementInit(pY, 1, px2, pGF1);
cout<<ippcpGetStatusString(eleInitStat)<<endl;

//pointer for the coeefients of polynomial for extension field initialisation
IppsGFpElement **pP1 = (IppsGFpElement**) malloc(maxBitSize);
pP1[0] = px1;
pP1[1] = px2;
IppStatus eleInitStatq = ippsGFpxInit(pGF1, deg, pP1, 2, methodx, pGFpx);
cout<<ippcpGetStatusString(eleInitStat)<<endl;
```

Chapter 8

Analysis of permutation polynomials using ippcp library

8.1 Mapping an Element in Finite Field w.r.t Polynomial

- A finite field is initialized using IppsGFpInit and the context of each element is initialized using ippsGFElementInit .
- Polynomial is expressed in the form of a string and the input string is passed to the function.
- Context of the element to which mapping need to be done is stored in eleConx1.
- Final result is stored in res .

```
1 //Ipp32u ele[1];
2 //ele[0] = 3;
3
4 Ipp32u poly_map(Ipp32u ele,string binaryString)
5 {
6     IppsGFpElement *eleConx1 = (IppsGFpElement *)malloc(eleSize1);
7     IppStatus eleInitStat1 = ippsGFpElementInit(ele, 1, eleConx1, pGF);
```

```
    IppsGFpElement *eleConx2 = (IppsGFpElement *)malloc(eleSize1);
    eleInitStat1 = ippsGFpElementInit(ele, 1, eleConx2, pGF);

    Ipp32u mul[1];
    mul[0] = 1;
    IppsGFpElement *eleConxMul = (IppsGFpElement *)malloc(eleSize1);
    IppStatus eleInitStat2 = ippsGFpElementInit(mul, 1, eleConxMul, pGF);

    for (int j = 0; j < binaryString.length(); j++)
    {
        if (binaryString[j] != '0')
        {
            eleInitStat1 = ippsGFpAdd(eleConx1, eleConxMul, eleConx1, pGF
   );
        }
        eleInitStat1 = ippsGFpMul(eleConxMul, eleConx2 , eleConxMul, pGF)
   ;
    }


    Ipp32u res[1];
    eleInitStat1 = ippsGFpGetElement(eleConx1, res, eleSize1, pGF1);
    cout << res[0]<<endl;
    return res;
}
```

8.2 Cardinality Check of Permutation Polynomial

Permutation polynomial is a polynomial which maps each element in a finite field to a unique element .

- To verify if a polynomial is permutation , we use the cardinality principle .

- For every element in the finite field , we find mapping for that element w.r.t the polynomial .

- The mapping for each element is then pushed into the set and if the number of elements in the finite field is equal to the number of elements in the set , then the given polynomial is a permutation polynomial .

```

set<int> finalval;
for (int i = 0; i < n; i++)
{
    Ipp32u ele[1];
    ele[0] = i;
    Ipp32u res = poly_map(ele,binaryString); // function defined in 8.1
    finalval.insert(res[0] );
}
if(finalval.size()==n )
 cout<<"Valid Permutation Polynomial "<<binaryString<<endl;
```

8.3 Complete Permutation Polynomial

A polynomial f(x) is a complete permutation polynomial if both f(x) and f(x)+x are permutation polynomials .

- String f(x)+x is calculated and is verified if the found polynomial is permutation polynomial or not.

```
string completeBinary=binaryString;
if(completeBinary.length()==1)
    completeBinary+='1';
else
    completeBinary[1]=(binaryString[1]+1)%n;
```

```
6
7  set<int> finalval,finalval1;
8  for (int i = 0; i < n; i++)
9  {
10     Ipp32u ele[1];
11     ele[0] = i;
12     Ipp32u res = poly_map(ele,binaryString); // function defined in 8.1
13     finalval.insert(res[0] );
14     Ipp32u res1 = poly_map(ele,completeBinary);
15     finalval1.insert(res1[0]);
16 }
17 if(finalval.size()==n && finalval1.size()==n )
18  cout<<"Valid Permutation Polynomial "<<binaryString<<endl;
```

8.4 Permutation Polynomial in extended field

Upon attempting to verify permutation polynomials in a extended field using the aforementioned codes, we encountered issues with the library's functionality. The verification process failed to detect a significant number of valid permutation polynomials. Our investigation led us to conclude that the library only supports functions related to initializing the context of an extension field. It appears to be incapable of handling arithmetic calculations for extended fields necessary for verification purposes. As the library is owned by Intel, limited information is available online regarding its limitations. In response to this issue, we have devised alternative methods for verifying whether a polynomial is a permutation polynomial in an extended field.

8.4.1 Verification using discrete logarithm

In cryptography, the discrete logarithm (DLog) problem is a fundamental problem that is used in many cryptographic schemes. The DLog problem is defined as follows: given a

group G with a generator g and an element h in G, find an integer x such that $g^x = h$. The dlog algorithm checks whether a polynomial is a permutation polynomial by evaluating it at all possible field elements and verifying that each element has a unique image. This verification process can be computationally intensive, particularly for large extension fields and high-degree polynomials.

Skeletal code representing working of this algorithm:

```
int field_size = n^d;
vector<int> dlog(field_size - 1);
bool isPP = true;
for (int i = 1; i < field_size; i++) {
    int elem = i;
    int log = 0;
    while (elem != 1) {
        elem = (inputString[0] * elem + inputString[1]) % n;
        log++;
    }
    dlog[i - 1] = log;
}

sort(dlog.begin(), dlog.end());
auto it = unique(dlog.begin(), dlog.end());
if (it != dlog.end()) {
    isPP = false;
}
```

8.4.2 Verification using Trace method

The trace method can also be used to verify whether a polynomial is a permutation polynomial in an extended field. The basic idea is to compute the trace of the polynomial and check whether it is equal to the trace of a known permutation polynomial.

To use this method, we first need to choose a permutation polynomial of degree d in the extension field. Let's call this polynomial P(x). We can then compute the trace of P(x) as follows:

$\mathrm{Tr}(\mathrm{P}(\mathrm{x})) = \mathrm{P}(\mathrm{x}) + \mathrm{P}(x^p) + \mathrm{P}(x^{p^2}) + \ldots + P(x^{p^{d-1}})$

where p is the characteristic of the field and p^d is the size of the field.

Next, we compute the trace of the polynomial we want to verify, let's call it Q(x):

Tr(Q(x)) = Q(x) + Q(x^p) + Q(x^{p^2}) + ... + $Q(x^{p^{d-1}})$

If Tr(Q(x)) is equal to Tr(P(x)), then Q(x) is a permutation polynomial in the field. Otherwise, it is not.

The advantage of using the trace method is that it is generally faster than the DLOG method, especially for large fields. However, it requires the choice of a permutation polynomial, which can be difficult to find for some fields.

8.4.3 Verification using cardinality check

The idea behind this method is to check whether the polynomial maps every element in the field to a unique element, which can be done by comparing the cardinalities of the domain and the range.

The cardinality of the domain is simply x^p, since GF(x^p) contains x^p elements. To calculate the cardinality of the range, we can construct a table of all the outputs of f(x) over GF(x^p) by evaluating f(x) at each element in GF(x^p). If f(x) is a permutation polynomial, then each output in the table should be distinct.

This method can be computationally expensive for large fields, however, it is a straightforward method that can be easily implemented using standard arithmetic operations over the field.

Skeletal code representing working of this algorithm:

```
// Evaluate the polynomial at all field elements
std::vector<int> values;
for (int i = 0; i < field_order; i++) {
    int x = i;
    int y = 0;
    for (int j = 0; j < coeffs.size(); j++) {
        y += coeffs[j] * pow(x, j);
        y %= p;
    }
    values.push_back(y);
}

// Check if the polynomial is a permutation
std::sort(values.begin(), values.end());
for (int i = 0; i < values.size(); i++) {
    if (values[i] != i % field_order) {
        return false;
    }
}

// Check if the polynomial satisfies the cardinality check
std::vector<int> counts(field_order);
for (int i = 0; i < values.size(); i++) {
    counts[values[i]]++;
}
for (int i = 0; i < counts.size(); i++) {
    if (counts[i] != field_order - 1) {
        return false;
    }
}
return true;
```

Chapter 9

TIME COMPARISION

To compare the effectiveness of the ipp-crypto library, we compared the execution time of verification of permutation polynomial using the library versus execution time of the sage math code which we had worked upon last semester.

9.1 Time taken to verify Permutation Polynomial in SageMath

The following code is used to check polynomial is permutation or not based on Cardinality Principle : code

9.2 Comparision results

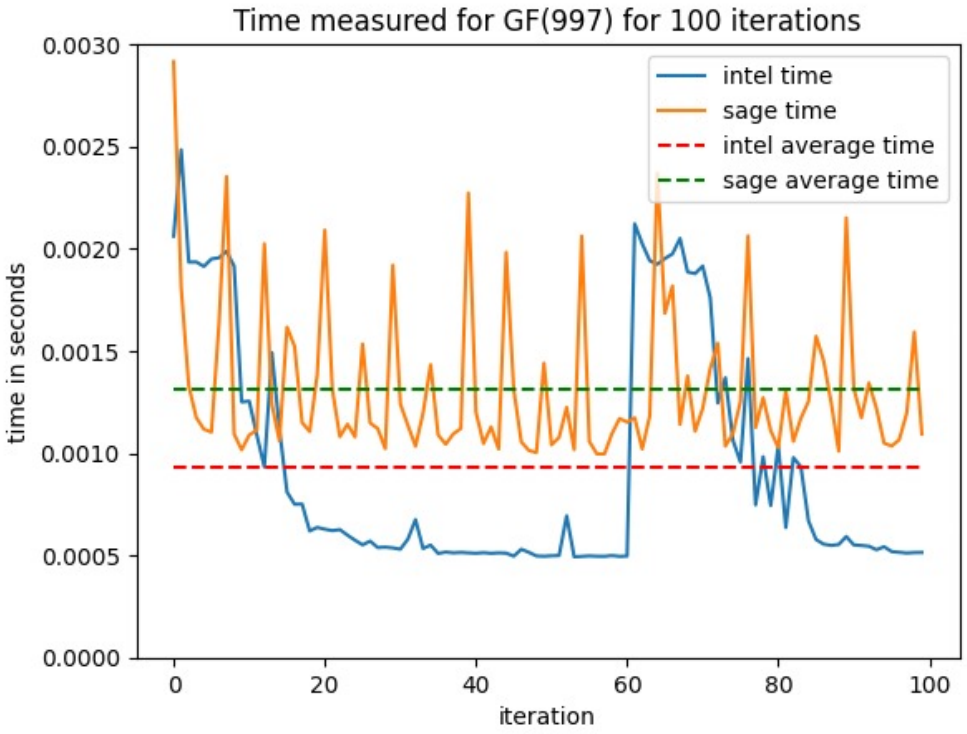
Th permutation polynomial used is $x + 1$.
The system specifications used to run the code include:

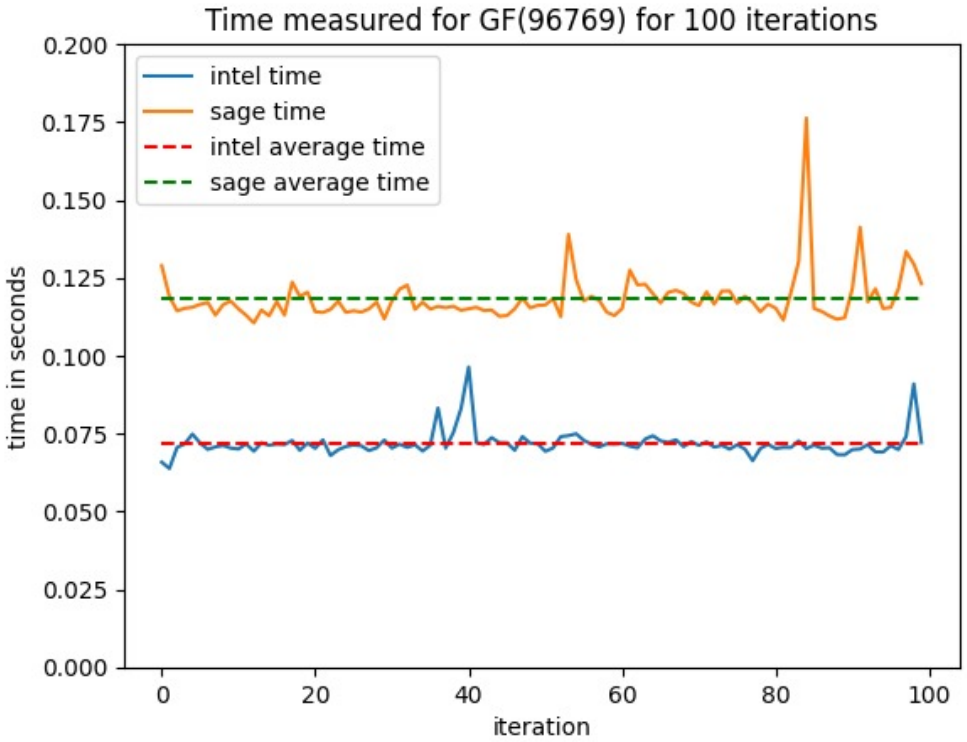
- Memory: 7.6 GB
- Processor: Intel Core i5-8250U CPU @ 1.60GHz × 8
- OS: Ubuntu 20.04.6 LTS, 64 bits

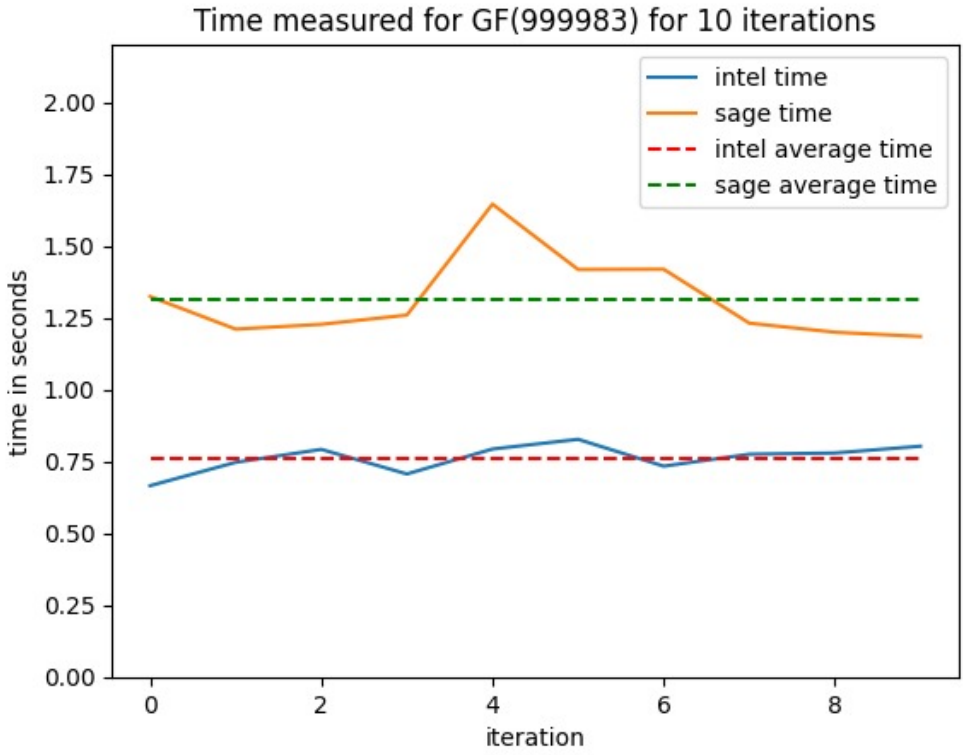
The execution time of both the codes is plotted and tabulated for prime field of different orders namely: 997, 96769, 999983 and 5206837 for more effective comparison.

The code used to plot and to find all time values measured : code

9.3 graphs and tables

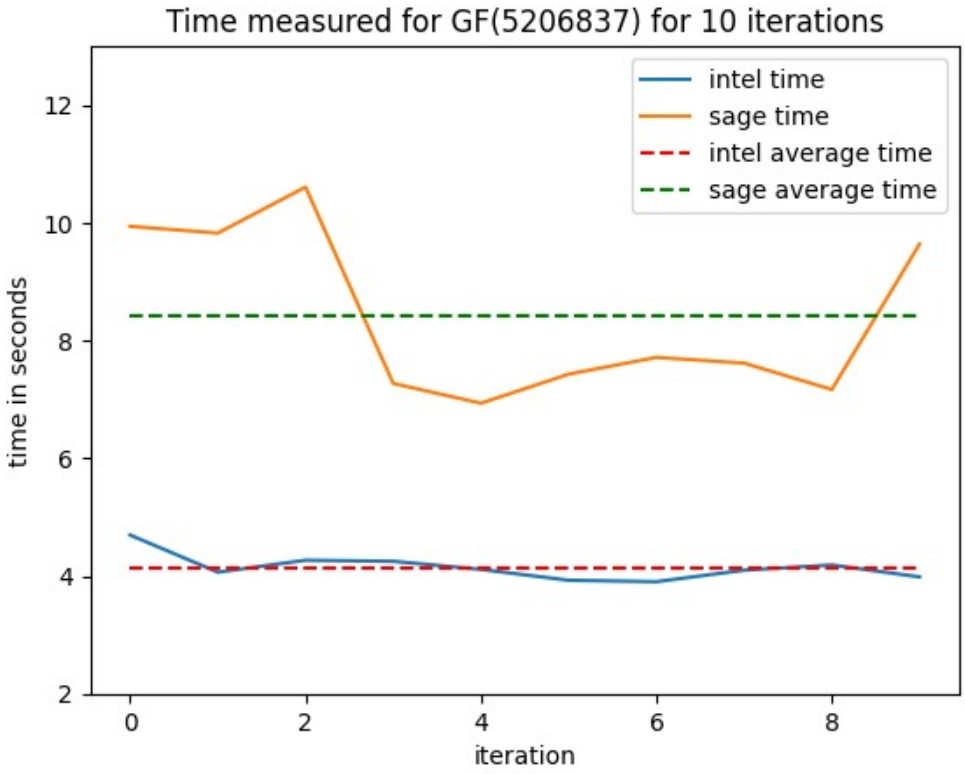

Time measured for GF(997) for 100 iterations
intel time
sage time
intel average time
sage average time
time in seconds
iteration
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
20
40
60
80
100


Time measured for GF(96769) for 100 iterations
intel time
sage time
intel average time
sage average time
time in seconds
iteration
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
20
40
60
80
100



Iteration no.	INTEL Library(s)	SageMath (s)
1	0.665574	1.32403588
2	0.747	1.210637569
3	0.792	1.22729969
4	0.706421	1.259593248
5	0.793246	1.6458301544
6	0.827053	1.418299436
7	0.733639	1.419113397
8	0.775096	1.231425523
9	0.779456	1.1999197006
10	0.803031	1.1850416603
Average Time	**0.7622516**	**1.31211962**

Fig. 9.1 Time measured using Intel and Sage for prime 999983



Iteration No.	INTEL Library(s)	SageMath (s)
1	4.49518	9.940570116
2	4.06419	9.8260138034
3	4.26722	10.60764145811
4	4.24903	7.26995444297
5	4.11124	6.9353291988
6	3.92675	7.42881321907
7	3.90301	7.713739395
8	4.09779	7.6161673069
9	4.18637	7.167483568191
10	3.98377	9.6395049095
Average Time	**4.148454**	**8.4145217418**

Fig. 9.2 Time measured using Intel and Sage for prime 5206837

9.4 Infernce

The experimental results clearly indicate that the ipp-crypto library is more efficient in analyzing permutation polynomials compared to the Sage math code. Specifically, for fields with lower order, the average execution time is comparable to that of the Sage code. However, for fields with higher order, a considerable difference in execution time is observed between the two codes, and the effectiveness of the ipp-crypto library increases significantly. It should be noted that the ipp-crypto library can only handle primes of order up to approximately $5 * 10^7$, after which the code crashes. Nevertheless, for any value below this order, the code is very effective in verifying whether a polynomial is a permutation polynomial or not for the given order.

Chapter 10

Conclusion and Future Work

Overall, this project has successfully demonstrated the effectiveness of the ipp-crypto library in analyzing permutation polynomials over finite fields. By implementing various methods for verifying permutation polynomials and comparing the performance of the ipp-crypto library with Sage code, we have shown that the library is highly efficient and performs well for fields with lower and moderate orders. However, there is a limit to the maximum order that the library can handle effectively.

There is significant potential for further development of the project in the field of cryptography, particularly in regards to Verifiable Delay Functions (VDFs). To continue building upon the findings of this project, the following areas could be explored for future work:

- Investigating methods for increasing the maximum order of the field to verify permutation polynomials.
- Exploring ways to verify permutation polynomials for extension fields using the library functions.
- Examining the feasibility of utilizing the permutation polynomials obtained through the library in VDFs and other cryptographic applications.

References

[1] ipp cryptography reference pdf

[2] Permutation polynomials

[3] Intel oneAPI

[4] Finite Field Arithmetic