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

The topic of the report is using the Intel Integrated Performance Primitives (Intel IPP) library to investigate finite field arithmetic.

The foundation for this study was established in the previous semester when we investigated permutation polynomials, defined VDF, and provided a strategy for efficiently benchmarking code performance.

We have also looked into the development and workings of the Intel IPP library. Then, we looked at big numbers and the related functions. Then we shifted focus to prime numbers and their functionalities in IPP library. Finally, we looked into the library's provided functions for finite field arithmetic.

In addition, we'll try to delve further into the library to develop appropriate permutation polynomials and evaluate their effectiveness as a VDF.

Contents

Chapter 1

Introduction

1.1 Project aim

VDFs are a cryptographic primitive used for various purposes, such as generating random beacons or producing proofs of elapsed time. They require a specific set of properties, including being computationally difficult to compute, but easy to verify, and taking a specific amount of time to compute. Intel Integrated Performance Primitives Cryptography (Intel IPPCP) is a set of cryptographic functions optimized for Intel processors. The main aim of the project is to use Intel IPPCP to optimize some of the cryptographic operations that are involved in VDFs.

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_ctx;
IppStatus status = ippcpInit_BN(&bn_ctx, 1024);

if (status == ippStsNoErr)
{
    // use bn_ctx for performing arithmetic operations on large integers
    ippcpFree_BN(bn_ctx);
}
else
{
    // error handling
}
```

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:

```
Ipp8u value[] = "\x12\x14\x16\x7a\x9b\xb1\x5e\xf4";

 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.

```
1  IppStatus ippsCmp_BN(const IppsBigNumState *x, const IppsBigNumState *y,
       Ipp32u *result);
2
3  IppStatus ippsCmpZero_BN(const IppsBigNumState *x, Ipp32u *result);
4
5  IppStatus ippsAdd_BN(IppsBigNumState *x, IppsBigNumState *y,
       IppsBigNumState *result);
6
7  IppStatus ippsSub_BN(IppsBigNumState *x, IppsBigNumState *y,
       IppsBigNumState *result);
8
9  IppStatus ippsMul_BN(IppsBigNumState *x, IppsBigNumState *y,
       IppsBigNumState *result);
10
11 IppStatus ippsMAC_BN_I(IppsBigNumState *x, IppsBigNumState *y,
       IppsBigNumState *result);
12
13 IppStatus ippsDiv_BN(IppsBigNumState *x, IppsBigNumState *y,
       IppsBigNumState * quotient, IppsBigNumState *remainder);
14
```

```
15 IppStatus ippsMod_BN(IppsBigNumState *x, IppsBigNumState *y,
      IppsBigNumState *result);
16
17 IppStatus ippsGcd_BN(IppsBigNumState *x, IppsBigNumState *y,
      IppsBigNumState *result);
18
19 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 .

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);
if (stat == ippStsNoErr)
    printf("No Error \n");
if (stat == ippStsNullPtrErr)
    printf("Error if any of the specified pointer is NULL \n");
if (stat == ippStsSizeErr)
    printf("error if primeSize <2 (or) primeSize > 1024 \n");
if (stat == ippStsContextMatchErr)
    printf("Error if Prime does not match the context \n");
if (stat == ippStsBadArgErr)
    printf("Error if q is even (or) bitsize(q) != primeBitSize (or) GF(q)
    modulus q is less than 3 \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);
if (stat == ippStsNoErr)
    printf("No Error \n");
if (stat == ippStsNullPtrErr)
    printf("Error if pGF is NULL (or) Both Prime and method are NULL\n");
if (stat == ippStsSizeErr)
    printf("error if primeSize <2 (or) primeSize > 1024 \n");
if (stat == ippStsContextMatchErr)
    printf("Error if Prime does not match the context \n");
if (stat == ippStsBadArgErr)
    printf("Error if q is even (or) bitsize(q) != primeBitSize (or) GF(q)
    modulus q is less than 3 (or) method is not null and not an output of
    GFpMethod\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.)

Chapter 6

Element of finite field and 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.

6.1 Extension fields

To initialise the context of extension field i.e. p^d field, we can use **GFpxInitBinomial()** or **GFpxInit()** function.

GFpxInit is used to initialize a finite field arithmetic context for prime fields. It requires the specification of the prime modulus of the field and the degree of the irreducible polynomial defining the field.

GFpxInitBinomial, on the other hand, is used to initialize a finite field arithmetic context for binary extension fields. It requires the specification of the degree of the field and the coefficients of the irreducible binary polynomial defining the field, which is typically a binomial.

Functions necessary for initialising the context of extension field:

- **GFpxMethod**: It is one of the techniques used in the IPP library for scalar multiplication on elliptic curves. It either assumes an arbitrary value or returns an implementation of an arithmetic operation over an extension field for a field polynomial.
- **GFpxGetSize**: The function returns the buffer size for the IppsGFpState context, which can be used to store data for the extension 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.

6.2 Element of the 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.

- **Conversion operations**: Functions such as ippsGFpElementExport, ippsGFpElementImport, and ippsGFpElementToOctString can be used to convert GF(p) elements to different representations such as byte arrays, hexadecimal strings, or octal strings.

- **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.

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

Chapter 7

Conclusion and Future Work

The IPP crypto library has been successfully installed and constructed, and the many functions from the domains that are utilised in the fundamental cryptographic processes, such as big numbers, prime numbers, and finite field arithmetic, have been understood and implemented. The context of a finite field has been successfully initialized, and we have explored the many functions needed to initialise the context of each element's finite field and its extension fields. As future work we will be working on implemention of extension fields and implemetaion of permutation polynomials by using the same IPP crypto library.

References

[1] ipp cryptography reference pdf

[2] Permutation polynomials

[3] Intel oneAPI

[4] Finite Field Arithmetic