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

Anurag Jha
(111901010)

Boda Pranav Aditya
(111901018)

BTP presentation,
under the guidance of prof. Srimanta Bhattacharya
18th May, 2023

IIT PALAKKAD

Table of Content

IIT PALAKKAD

Introduction

The main aim of the project is to **use Intel IPP crypto library to optimize analysis of permutation polynomials**

Previous work includes:

1. **Studying and exploring**: Finite Field Arithmetic, applications of field characteristics, permutation polynomials(PP), irreducible polynomials, hermite's criteria for PP.
2. **Coding**: PP fearures. [especially for checking and confirming PP].
3. **Studying VDFs**
4. **Testing different benchmarking methods**.

IIT PALAKKAD

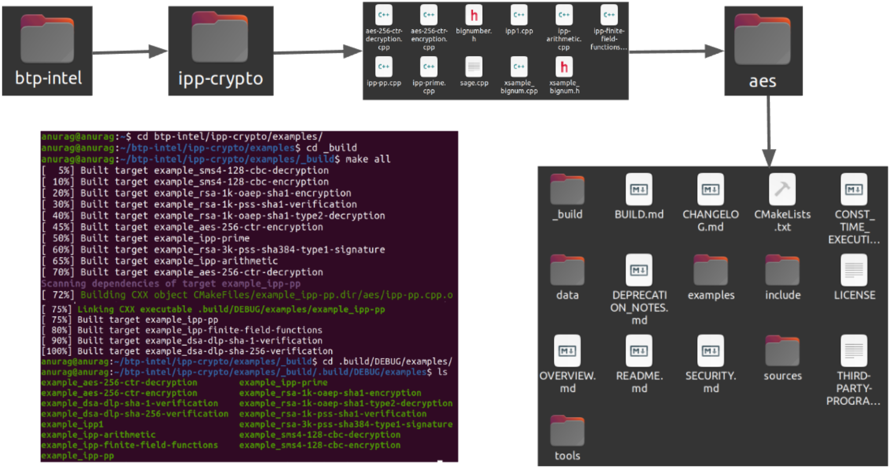
Introduction to ipp-crypto library

Intel Integrated Performance Primitives(Intel IPP) is a software library provided by Intel that offers a wide range of functions for accelerating various computational tasks, including cryptography.
The main features of the library include:

1. Thread safe design
2. Optimized for intel processors
3. Works with a range of operating systems and development environments

IIT PALAKKAD

Running ipp-crypto library




```
anurag@anurag:~$ cd btp-intel/ipp-crypto/examples/
anurag@anurag:~/btp-intel/ipp-crypto/examples$ cd _build
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build$ make all
[  5%] Built target example_sms4-128-cbc-decryption
[ 10%] Built target example_sms4-128-cbc-encryption
[ 20%] Built target example_rsa-1k-oaep-sha1-encryption
[ 30%] Built target example_rsa-1k-pss-sha1-verification
[ 40%] Built target example_rsa-1k-oaep-sha1-type2-decryption
[ 45%] Built target example_aes-256-ctr-encryption
[ 50%] Built target example_ipp-prime
[ 60%] Built target example_rsa-3k-pss-sha384-type1-signature
[ 65%] Built target example_ipp-arithmetic
[ 70%] Built target example_aes-256-ctr-decryption
Scanning dependencies of target example_ipp-pp
[ 72%] Building CXX object CMakeFiles/example_ipp-pp.dir/aes/ipp-pp.cpp.o
[ 75%] Linking CXX executable .build/DEBUG/examples/example_ipp-pp
[ 75%] Built target example_ipp-pp
[ 80%] Built target example_ipp-finite-field-functions
[ 90%] Built target example_dsa-dlp-sha-1-verification
[100%] Built target example_dsa-dlp-sha-256-verification
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build$ cd .build/DEBUG/examples/
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build/.build/DEBUG/examples$ ls
example_aes-256-ctr-decryption          example_ipp-prime
example_aes-256-ctr-encryption          example_rsa-1k-oaep-sha1-encryption
example_dsa-dlp-sha-1-verification      example_rsa-1k-oaep-sha1-type2-decryption
example_dsa-dlp-sha-256-verification    example_rsa-1k-pss-sha1-verification
example_ipp1                            example_rsa-3k-pss-sha384-type1-signature
example_ipp-arithmetic                  example_sms4-128-cbc-decryption
example_ipp-finite-field-functions      example_sms4-128-cbc-encryption
example_ipp-pp
```

IIT PALAKKAD

Big number arithmetic

Big number arithmetic is necessary for handling these large prime numbers and for performing efficient computations within finite fields, which are crucial for many cryptographic operations.
The library provides primitives for performing arithmetic operations with integer big numbers of variable length. The major functions include:

- **Initialisation**: **ippcpInit_BN()** initializes the Big Number arithmetic functionality in the IPP Cryptography library. It creates the internal data structures needed to execute arithmetic operations on big numbers as well as allots memory for the (BN) context.
- **Set**:
  - **ippsSet_BN()**: Sets a big number to a specific value.
  - **SetOctString_BN()**: Converts a string to a big number.

IIT PALAKKAD

Big number arithmetic

- **Arithmetic**:
  - **Modular Arithmetic**: **ippsModulus**[modular reduction] and **ippsModInv** [modular inverse].
  - **Addition and Subtraction**: **ippsAdd_BN** and **ippsSub_BN**.
  - **Multiplication**: **ippsMul_BN** and **ippsMontMul_BN** [Montgomery multiplication].
  - **Division**: **ippsDiv_BN** calculates only the quotient **ippsDivMod_BN** calculates both the quotient and the remainder.
- **Comparison and Bit Manipulation**

The big number class in IPP Cryptography is crucial for cryptographic algorithms that utilize large integers, providing essential functionality for efficient computations.

IIT PALAKKAD

Prime numbers

The finite field arithmetic procedures employing the library rely on the prime number arithmetic. Primes define the finite field over which cryptographic operations are performed

The generation and utilization of prime numbers in finite field arithmetic are crucial for ensuring the security of cryptographic algorithms.

IPPCP functions for prime number generation include:

1. **PrimeGetSize** : The size needed to set up the IppsPrimeState context is returned by this function.
2. **PrimeInit** : IppsPrimeState context is initialized based on the user-supplied memory pointer.
3. **PrimeGen_BN** : The function generates a random prime number using the BigNum (BN) library. .
4. **PrimeTest_BN** : The function is used for primality testing of large integers as BN objects.

IIT PALAKKAD

Field initialisation

The finite field arithmetic functions uses ***IppsGFpState*** to store data of the finite field and ***IppsGFpElement*** to store data of finite elements. Prime is generated according to the input bits. The next step is to initialise context pf the finite field. Ways to initialise context of finite field include:

1. GFpInitFixed : Initializes pGF and sets up the value of the GF(q) modulus to the chosen method.
2. GFpInitArbitary : Initializes pGF and sets up the value of the GF(q) modulus to the value specified by Prime.
3. GFpInit : 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.

IIT PALAKKAD

Field initialisation

1. ippsGFpInitFixed(primeBitSize, method, pGF);
2. ippsGFpInitArbitrary(pPrime, primeBitSize, pGF);
3. ippsGFpInit(pPrime, primeBitSize, method, pGF);

.

NOTE

- GFpInit() behaves similar to GFpInitArbitrary() when method $==$ NULL and as GFpInitFixed() when prime $==$ NULL.
- **GFpMethod**(Pointer to Implementation of arithmetic operations over the finite field)
- **GFpGetSize**(returns the size of the memory buffer required to hold a GFp context structure).

IIT PALAKKAD

Elements of finite field

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

Operations supported by these functions include:

- **Initialization**: Functions such as **ippsGFpElementInit** and **ippsGFpElementInitFixed** can be used to initialize GF(p) elements with either random(**GFpSetElementRandom**) or fixed values(**ippsGFpSetElement, GFpSetElementOctString, GFpSetElementHash**).
- **Arithmetic operations**: Functions such as **ippsGFpElementAdd**, **ippsGFpElementSub**, **ippsGFpElementMul**, and **ippsGFpElementDiv** can be used to perform arithmetic operations on GF(p) elements.

IIT PALAKKAD

Elements of finite field

- **Comparison operations**: ippsGFpElementCmp can be used to compare GF(p) elements.
- **Other operations**: ippsGFpElementPower and ippsGFpElementIsZero can be used to perform other operations on GF(p) elements such as exponentiation and checking whether an element is zero. The other opertations supported includes computing: conjucate, inverse, negation, squre-root, square etc.

IIT PALAKKAD

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.

- Select an irreducible polynomial of degree m to initialise an extension field.
- Extension field can be created by specifying the field operations in terms of the binary polynomials that stand in for the field elements.
- Polynomials should be carefully chosen to guarantee that the resulting field has desirable cryptographic qualities.

IIT PALAKKAD

Extension field functions

Initializing the context of an extension field and an individual finite field element comes after initialising the context of a prime finite field.

- **GFpxMethod**: Represents the arithmetic methods of a finite field extension GF(p^d). Contains function pointers to the basic arithmetic operations over extended field.
- **GFpxGetSize**: Calculate the size of the memory that is required for the specified extension field.
- **GFpxInitBinomial**: Initializes a context for an extension field generated by an irreducible binomial over a prime field.
- **GFpxInit**: Initializes a finite field of extension using the supplied field descriptor and element data.
- **GFpScratchBufferSize**: Scratch buffers are used to store intermediate results during cryptographic operations.

IIT PALAKKAD

Mapping Element in Finite Field

- After defining the finite field and initialising the context of elements , we need to map an element in finite field w.r.t polynomial.
- Polynomial is represented in the form of string .

Pseudo Code:

```
Ipp32u poly_map(element,string)
{
        ele=element
        for i in string
        {
                if(i!="0")
                        ippsGFAdd(ele,mul,ele,pGF)
                ippsGFMul(mul,element,mul,pGF)
        }
        return ele
}
```

IIT PALAKKAD

Permutation Polynomials and Verification

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

.

- Verification of permutation polynomial is done using Cardinality check
  .
- Mapping of every element is found using poly_map function and added to set . If number of elements in finite field and set are equal , then the polynomial is Permutation.

Pseudo Code :

```
set <int > finalval ;
for i in GF(n) {
        Ipp32u res = poly_map(ele , String)
        finalval.insert(res[0])
}
if( finalval.size()==n)
      print(" Valid Permutation Polynomial ", String)
```

IIT PALAKKAD

f(x) is complete permutation polynomial if both f(x) and f(x)+x are permutation polynomials , can be verified using cardinality

- Find the string for polynomial f(x)+x and substitute in poly_map function.

Pseudo Code :

```
set <int > finalval,finalval1 ;
String1 = String ;
if len(String==1): String1 +="1"
else String1[1] = (String1[1]+1)%n
for i in GF(n) {
 Ipp32u res =poly_map(ele,String);finalval.insert(res[0])
 Ipp32u res1 =poly_map(ele,String1);finalval1.insert(res1[0])
}
if( finalval.size()==n && finalval1.size()==n)
     print(" Valid Complete Permutation Polynomial ")
```

IIT PALAKKAD

Time Taken to verify Permutation Polynomial

SageMath: Time module is used to calculate the time taken to run the function to verify permutation polynomial.

```
import time
start=time.time()
// function
end=time.time()
```

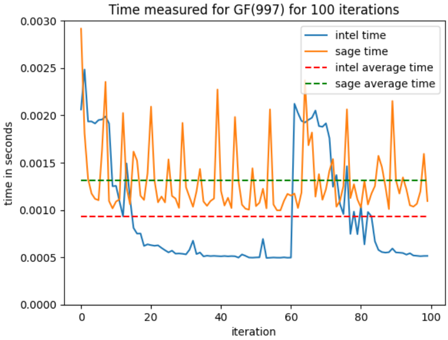
Intel: clock_gettime() function is used with 'CLOCK_MONOTONIC' as specified clock identifier.

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.

IIT PALAKKAD

Primes 997 and 96769

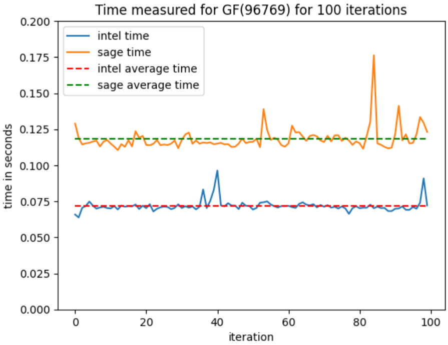
997:



Average Sage Execution Time : 0.001313 s

Average Intel Execution Time : 0.000931 s

96769:

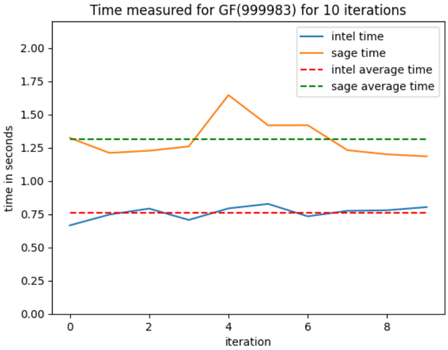


Average Sage Execution Time : 0.118327 s

Average Intel Execution Time : 0.071790 s

IIT PALAKKAD

Prime 999983:



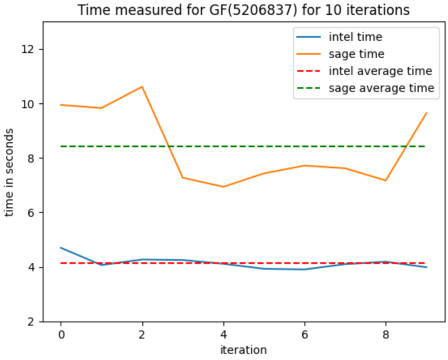
Iteration no.	INTEL Library(s)	SageMath (s)
1	0.665574	1.32403588
2	0.747	1.210637569
3	0.792	1.22729969
4	0.706421	1.259593248
5	0.793246	1.6458301544
6	0.827053	1.418299436
7	0.733639	1.419113397
8	0.775096	1.231425523
9	0.779456	1.1999197006
10	0.803031	1.1850416603
Average Time	**0.7622516**	**1.31211962**

Average Sage Execution Time : 1.312119 s

Average Intel Execution Time : 0.7622516 s

IIT PALAKKAD

Prime 5206837:



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**

Average Sage Execution Time : 8.4145217 s

Average Intel Execution Time : 4.1484549 s

Conclusion and Future works

This project has successfully demonstrated the effectiveness of the ipp-crypto library in analyzing permutation polynomials over finite fields. 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.

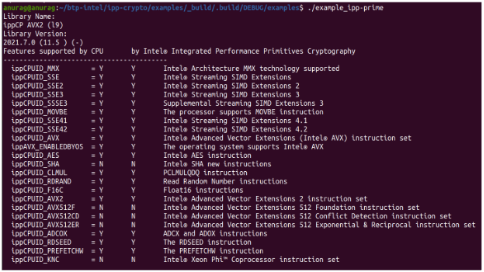
Future works include:

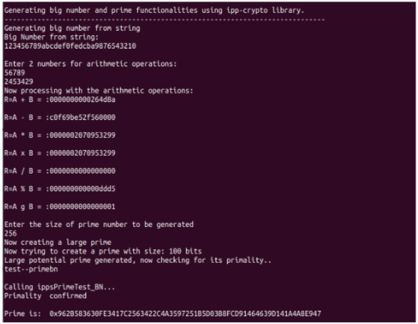
- Increasing the order of the field to verify PP.
- Verifying PP for extension fields using the library functions.
- Utilizing the PP obtained through the library in VDFs and other cryptographic applications.

THANK YOU

IIT PALAKKAD

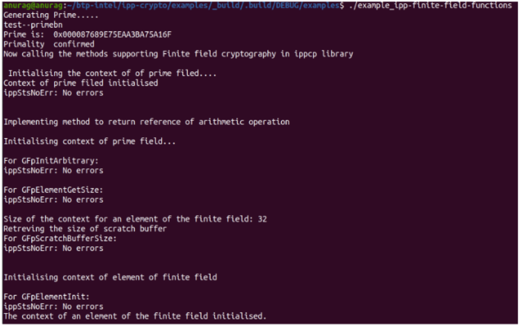
Running big number and prime generation functions



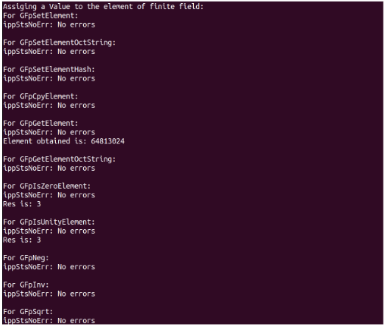


IIT PALAKKAD

Running Finite Field functions



```
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build/.build/DEBUG/examples$ ./example_ipp-finite-field-functions
Generating Prime.....
test--primebn
Prime is:  0x000087689E75EAA3BA75A16F
Primality  confirmed
Now calling the methods supporting Finite field cryptography in ippcp library

 Initialising the context of of prime filed....
Context of prime filed initialised
ippStsNoErr: No errors


Implementing method to return reference of arithmetic operation

Initialising context of prime field...

For GFpInitArbitrary:
ippStsNoErr: No errors

For GFpElementGetSize:
ippStsNoErr: No errors

Size of the context for an element of the finite field: 32
Retreving the size of scratch buffer
For GFpScratchBufferSize:
ippStsNoErr: No errors


Initialising context of element of finite field

For GFpElementInit:
ippStsNoErr: No errors
The context of an element of the finite field initialised.
```



```
Assiging a Value to the element of finite field:
For GFpSetElement:
ippStsNoErr: No errors

For GFpSetElementOctString:
ippStsNoErr: No errors

For GFpSetElementHash:
ippStsNoErr: No errors

For GFpCpyElement:
ippStsNoErr: No errors

For GFpGetElement:
ippStsNoErr: No errors
Element obtained is: 64813024

For GFpGetElementOctString:
ippStsNoErr: No errors

For GFpIsZeroElement:
ippStsNoErr: No errors
Res is: 3

For GFpIsUnityElement:
ippStsNoErr: No errors
Res is: 3

For GFpNeg:
ippStsNoErr: No errors

For GFpInv:
ippStsNoErr: No errors

For GFpSqrt:
ippStsNoErr: No errors
```

IIT PALAKKAD

Running Arithmetic functions

```
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build/.build/DEBUG/examples$ ./example_ipp-arithmetic
Generating Prime of given bitsize
Checking Primality:
Prime is:  0x000087689E75EAA3BA75A16F
Primality  confirmed

Now calling the methods supporting Finite field cryptography in ippcp library

 Initialising the context of of prime filed....
Context of prime filed initialised

Implementing method to return reference of arithmetic operation over GF(q)..

Initialising context of prime field GF(p^d) field...
execution ok1

Elem size is: 32

Initialising context of element of finite field using ippsGFpElementInit
First element initialised
Second element initialised
Result element initialised

setting values to the elements of finite field using ippsGFpSetElement
Values Set completed
ippsGFpGetElement works
Element 1: 1
ippsGFpGetElement works
Element 2: 0

implementing arithmetic operations on the elements of the field

ADD:
Addition went fine

SUB:
Subtraction went fine

MUL:
Multiplication went fine

Square:
Square computation went fine
```

Running PP functions

```
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build/.build/DEBUG/examples$ ./example_ipp-pp
Enter the order of filed
5206837
Enter the Degree of Extension
1
Enter the Coefficeints of polynomial as string
11
1x^0 + 1x^1 + 0

Initialising the context of prime field....
Context of prime filed initialised
execution ok1
For ele init: ippStsNoErr: No errors
Initialising element of finite field:
ippStsNoErr: No errors
Initialising element of finite field:
ippStsNoErr: No errors
FOR BINOMIAL
ippStsBadArgErr: Incorrect arg/param of the function

Size of the Field = 5206837

Now for every elemnt is the field
Finding a mapping of the element using diffrent ipp-crytp library functionalities and stroring it in the set
Now starting the time measure using clock_gettime

The elements of the fields and the image set sizes are the same:
The given string is a Permutation polynomial

VALID
Time measured: 9.675 seconds.
-------------------------
```

IIT PALAKKAD

Running PP functions

```
Enter the order of field
1000
Enter the Degree of Extension
1
Enter the Coefficeints of polynomial as string
11
1x^0 + 1x^1 + 0
Size of the Field = 1000

Now for every elemnt is the field
Finding a mapping of the element using diffrent ipp-crytp library functionalities and stroring it in the set
Now starting the time measure using clock_gettime

The elements of the fields and the image set sizes are not the same:
The given string is not a Permutation polynomial

Time measured: 0.001 seconds.
```

IIT PALAKKAD

Running PP functions

```
anurag@anurag:~/btp-intel/ipp-crypto/examples/_build/.build/DEBUG/examples$ ./example_ipp-pp
Enter the order of field
999983
Enter the Degree of Extension
1
Enter the Coefficeints of polynomial as string
11
1x^0 + 1x^1 + 0

Size of the Field = 999983

Now for every elemnt is the field
Finding a mapping of the element using diffrent ipp-crytp library functionalities and stroring it in the set
Now starting the time measure using clock_gettime

The elements of the fields and the image set sizes are the same:
The given string is a Permutation polynomial

VALID
Time measured: 1.481 seconds.
-------------------------
```

Running PP functions

```
Enter the order of field
97
Enter the Degree of Extension
1
Enter the Coefficeints of polynomial as string
22
2x^0 + 2x^1 + 0
```

```
Size of the Field = 97

Now for every elemnt is the field
Finding a mapping of the element using diffrent ipp-crytp library functionalities and stroring it in the set
Now starting the time measure using clock_gettime

The elements of the fields and the image set sizes are not the same:
The given string is not a Permutation polynomial
```

```
Enter the order of field
997
Enter the Degree of Extension
2
Enter the Coefficeints of polynomial as string
11
1x^0 + 1x^1 + 0
Size of the Field = 994009

Now for every elemnt is the field
Finding a mapping of the element using diffrent ipp-crytp library functionalities and stroring it in the set
Now starting the time measure using clock_gettime

The elements of the fields and the image set sizes are the same:
The given string is a Permutation polynomial
```

IIT PALAKKAD