4

IMPLEMENTATION OF WORK

4.1 Introduction

One of the main focuses of this research work is to study if it is suitable to implement the elliptic curve cryptography for embedded platform using a high level object oriented language. It is also studied whether the large data size required for cryptographic based schemes can be made flexible using a user defined structure so that the data size or more specifically key size can be increased or changed during runtime and as and when required as per the needs of security or as required by user application. This means user can select any key size without worrying about if the software is able to support that key size or not.
The objective is to develop a unified software architecture that can perform ECC related operations and it should provide following key features: algorithm agility, resource utilization and compatibility.

**Algorithm Agility**

Algorithm Agility allows the user to switch cryptographic algorithms during runtime. Therefore, if current cryptosystems become obsolete due to new security needs, several back-up cryptosystems can be used as a replacement without requiring redesign.

This is important feature because, the security of most cryptographic algorithms is not proven, but merely presumed to be intractable with currently available computing power. Moore’s law therefore plays an important role in estimating key lengths for long term security. Also, one can never be entirely sure that better methods for cryptanalysis, than those currently known, do not exist. Quite recently a theoretical attack on RSA, based on an improved scheme for factoring integers, has been proposed which, if practical, could render RSA keys of less than 1500 bits insecure.

**Resource Utilization**

Resource Utilization uses the same hardware to perform the majority of the arithmetic used for the different supported algorithms. Therefore, it reduces power and area consumption, which is critical for resource constrained applications. Alternatively, the architecture can be designed so that each algorithm allocates a separate area, which is not ideal.

Instead of implementing a complete cryptographic algorithm in hardware, it is often better to simply build universal arithmetic units. Several of such units for certain complex operations commonly found in cryptographic algorithms, can
be integrated into a microcontroller or microprocessor. The programmability of
the processor provides the flexibility of this approach, the specialization of the
arithmetic unit provides the performance, and the universality of the arithmetic
primitives enhances the reusability. This concept is extended in the current research
to an Object Oriented aspect by defining data types to be of variable length and
implementing related functions for basic arithmetic and logical operations. As
basic operations are defined as programming modules, this achieves the target of
algorithmic reusability in a more flexible way.

**Compatibility**

Compatibility allows interoperability of various applications through a
multitude of cryptographic algorithms. The problem stems from the fact that high
end platforms may utilize cryptosystems such as RSA, whereas, for smart cards or
other embedded applications ECC may be more reasonable choice.

A number of different public key algorithms are in use today. To ensure
compatibility with the rest of the world, cryptographic applications have to support
a large portion of those algorithms. While software implementations are often
easy to upgrade and to adapt to new algorithms or larger key sizes, the same is not
necessarily true for hardware implementations.

Cryptographic mechanisms based on elliptic curves depend on arithmetic
involving the points of the curve. As noted in previous chapters, curve arithmetic
is defined in terms of underlying field operations, the efficiency of which is
essential. Efficient curve operations are likewise crucial to performance. Figure
4.1 illustrates module framework required for a protocol such as the Elliptic Curve
Digital Signature Algorithm (ECDSA). The curve arithmetic not only is built on
field operations, but in this case also relies on big number and modular arithmetic
(binary fields of type $GF(2^m)$).
4.2 Choice of Platform

While considering an embedded platform, 8051 is an obvious choice. But, cryptographic applications have enhanced speed and storage requirements that give a chance to explore other possibilities available in the field of microcontroller. During this research, it is observed that AVR based microcontroller is the most preferred option in the field of elliptic curve cryptography. Papers like [1],[2] consider AVR for implementing ECC. Below is a fair comparison of 8051 and AVR microcontroller.

8051 - 8 bit microcontroller based on CISC architecture (Complex Instruction Set Computer). It has 250 instructions which take 1 to 4 machine cycles to execute. 8051 is able to execute 1 million instructions per second. 8051 has
very powerful instruction set, it has commands which do more complex calculations, it also has strong arithmetic logic unit which makes computation simple. It has instructions to multiply and divide, MUL and DIV respectively. 8051 are manufactured by over 50 companies. For software support platforms, It do support C very well, but when it comes to C++, It is observed that very few open source packages provide genuine support and user friendly environment for C++ programming.

**AVR -** 8 bit micro based on RISC architecture (Reduced Instruction Set Computer). AVRs have 140 instructions which are mostly 1 cycle based. In general, AVR is able to execute 12 million instructions per second. It does not have multiply and divide instruction, but then, for cryptography, special multiplication and division algorithms are used that basically use add/subtract and shift operations which are easily available in machine language. AVR has only single manufacturer that is Atmega. AVR Studio, the most sophisticated tool for AVR based C/C++ IDE is available free. For C++, some extra care has to be taken, but, it has negligible impact on overall performance. Although AVR is costly, it comes with many on chip peripherals and specially the SRAM which provide extra registers for runtime data storage.

### 4.3 Brief AVR Architecture Overview

The Atmel AVR ATmega128 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega128 achieves throughputs approaching 1MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Fig. 4.2 shows pin diagram for Atmega128 [3]
The architecture of the various AVR devices like Atmega 8, Atmega 128 is not very different from each other. There is not much difference between ATMEGA8, ATMEGA16, and ATMEGA32, ATMEGA128.

Fig. 4.2 Atmega128 pinout
Following features of Atmega128 that make it more advanced than 8051, and suitable for faster tasks like the one under discussion.

- **Up to 16 MIPS**: The ATMEGA128, being an AVR core, can execute up to 16 Million Instruction per Second. This is due to the fact that most AVR instructions are executed a single clock cycle. So with 16 Mhz clock, you can perform 16 MIPS. Comparatively, an 8051 clocked at 8 Mhz, would give a throughput of only 666Khz, because 8051 need 12 clock cycles per instruction, also there are instruction requiring more than one machine cycle.

- **EEPROM Memory**: Most AVRs, including the ATMEGA128, come with an on chip EEPROM memory, with ready to use instruction to access this memory. ATmega128 contains 4K bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles.

- **ISP**: In System Programming is becoming a standard in today’s microcontroller, and AVRs fully benefit for this technological advance, making it much simpler for developer to test and debug their chips ‘in system’. The Onchip ISP Flash allows the program memory to be reprogrammed in-system through an SPI serial interface, by a conventional nonvolatile memory programmer, or by an On-chip Boot program running on the AVR core.

- **Very Powerful and versatile TIMERS/COUNTERs**: Most AVR timers/counters have Prescallers, allowing them to be adapted to wide range of applications, and to dramatically reduce the processor
overhead. They also have a high sampling rate, enabling them to count very fast external events.

There is also a set of built-in devices that importantly reduce the number of components in any project:

- Two 8-bit PWM channels
- 8 channel 10 bit ADC
- USART/TWI/SPI interfaces
- On-Chip Analog comparator
- Internal RC oscillator:

This critical feature makes the ATMEGA128 a microcontroller that can run with only single supply (4.5 to 5.5V and GND rails, no any other component or connection is needed to be made to make it functional. Generally, any ATMEGA microcontroller is shipped with the internal oscillator turned ON and tuned to 1 MHz, making it ready to be used with adding any external components like crystal resonator or capacitors.

Another major characteristic of the AVR is that some of their internal registers are READ-ONLY, and some are WRITE-ONLY ensuring reliable operation.

**Input/ Output ports**

Unlike 8051, but like most microcontrollers, it is needed to specify the direction of the PINs of a port. Nevertheless, in AVRs, when a pin is configured as an INPUT pin, you can choose whether it provides High impedance or it is connected to an internal pull up resistor. Enabling the internal pull up resistor can allow you to reduce furthermore the external components counts. For example when connecting a simple push button to ground.
Figure 4.3 shows a simplified diagram of a general PORT of the ATMEGA128. The letter X can be replaced by the name of the port A through G. A general port is controlled by three registers: PORTx, DDRx, and PINx (again, where x is the name of the port).

- DDRx: lets you configure the direction of the pins of the port ‘x’. Each bit of the 8 bits of the DDRx register, represents one PIN of the concerned port. A 0 bit sets the corresponding PIN as Input, a 1 sets it as output.

For example, if:

DDRA = 0xF0; means that PINs 0,1,2 and 3 of PORT A are set as Inputs, while the four others are set as outputs.
DDRx is a READ/WRITE register.

- PORTx lets you define the logic level of the pins, in case they are configured as ‘output’. But for pins that are configured as ‘input’, PORTxlet’s you define whether it is a high impedance input by writing 0 to PORTx, or pulled up input (internal pull up resistor enabled) by writing 1 to PORTx.

The following example sets the PIN 0,1,2,3 as inputs, and the four others as output. Then, we enable the PULL-UP resistor for the first 2 inputs (PIN 0 and 1), and output '00112 on the four output pins:

\[
\text{DDRA} = 0xF0; \\
\text{PORTA} = 0x33; \\
\]

PORTx is a READ/WRITE register.

- PINx lets you read the actual logic levels of the pins of a PORT. This register is the only way to read the state a PIN. In other words, the PINx register is used for INPUT operation.

The following example uses the same pin configuration for PORT A above, but also inputs the state of the pins into a variables named ‘pin_value’:

\[
\text{DDRA} = 0xF0; \\
\text{PORTA} = 0x33; \\
\text{pin_value} = \text{PINA}; \\
\]

If the port has some pin configured as outputs, and some as input, then PINx register will read both inputs and outputs. It is the programmer’s responsibility to filter the unwanted data from the PINx register.

PINx is a READ-ONLY register; you may not write data into it, and most compilers will warn you if you attempt to do so.
Interrupts Vectors and Timers

The syntax used in the GCC compiler to assign an interrupt vector to a function is the following:

```c
SIGNAL(SIG_INT_VECTOR) {
...function’s body...
}
```

Where SIGNAL is a normalized name (you cannot exchange it with any other name like my_function) and SIG_INT_VECTOR is the interrupt vector, and is to be replaced with a valid interrupt vector definition. You can find all the definitions of the interrupt vectors of your microcontroller in it’s header file.

Other features include:

Operating Voltages
- 2.7 - 5.5V for ATmega128L
- 4.5 - 5.5V for ATmega128

Speed Grades
- 0 - 8 MHz for ATmega128L
- 0 - 16 MHz for ATmega128

Whether or not to consider C++ for implementation of this algorithm is a main point of discussion as entire issue of speed versus algorithmic flexibility relies on it. Especially for embedded systems, there has been much debate within the embedded software community about whether C++ is worth the performance penalty. It is generally agreed that C++ programs produce larger executable that run more slowly than programs written entirely in C. However, C++ has many benefits for the programmer, not everything introduced in C++ is expensive. Many older C++ compilers incorporate a technology called C-front that turns C++
programs into C and feeds the result into a standard C compiler. The mere fact that this is possible should suggest that the syntactical differences between the languages have little or no runtime cost associated with them. It is only the newest C++ features, like templates, that cannot be handled in this manner.

For example, the definition of a class is completely benign. The list of public and private member data and functions are not much different than a ‘struct’ and a list of function prototypes. However, the C++ compiler is able to use the public and private keywords to determine which method calls and data accesses are allowed and disallowed. Because this determination is made at compile time, there is no penalty paid at runtime. The addition of classes alone does not affect either the code size or efficiency of your programs.

Default parameter values are also penalty-free. The compiler simply inserts code to pass the default value whenever the function is called without an argument in that position. Similarly, function name overloading is a compile-time modification. Functions with the same names but different parameters are each assigned unique names during the compilation process. The compiler alters the function name each time it appears in your program, and the linker matches them up appropriately. This feature of C++ is rarely used in the proposed functional modules, but it could have been done without affecting performance.

For current implementation AVR Studio4.17 is used which requires support of winAVR. WinAVR is a suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. It includes the GNU GCC compiler for C and C++.

These software development tools include:

- Compilers
- Assembler
Each of the tools included in WinAVR is Open Source and/or Free Software. Each tool has its own project, usually hosted on SourceForge or Savannah, with their own project maintainers and developers who all volunteer to create these tools. Many of these programs come from the Unix and Linux platforms. These programs have been ported to the Windows platform but generally behave for a Unix-like environment. The compiler in WinAVR is the GNU Compiler Collection, or GCC. This compiler is incredibly flexible and can be hosted on many platforms; it can target many different processors/operating systems (back-ends), and can be configured for multiple different languages (front-ends).

The GCC included in WinAVR is targeted for the AVR processor, is built to execute on the Windows platform, and is configured to compile C, or C++. But it has issues while using C++. Some of them, as mentioned in WinAVR FAQ section are:

- None of the C++ related standard functions, classes, and template classes are available.
The operators new and delete are not implemented; attempting to use them will cause the linker to complain about undefined external references. (This has been fixed.)

Some of the supplied include files are not C++ safe, i.e., they need to be wrapped into

```
extern "C" {
...
}
```

Exceptions are not supported. Since exceptions are enabled by default in the C++ frontend, they explicitly need to be turned off using `-fno-exceptions` in the compiler options. Failing this, the linker will complain about an undefined external reference to `__gxx_personality_sj0`.

Constructors and destructors are supported though, including global ones.

One of the major hurdles while programming in C++ on AVR Studio 4.x is that it does not support new and delete operators. Following piece of code is compulsory to make it support C++.

```c
#include <stdlib.h>
void * operator new(size_t size);
void operator delete(void * ptr);
void * operator new(size_t size)
{
  return malloc(size);
}
void operator delete(void * ptr)
{
  free(ptr);
}
```
Following piece of code is also necessary if templates and virtual inheritance has to be used.

```
__extension__ typedef int __guard __attribute__((mode (__DI__))));
extern "C" int __cxa_guard_acquire(__guard *);
extern "C" void __cxa_guard_release (__guard *);
extern "C" void __cxa_guard_abort (__guard *);
```

### 4.4 The Data Structure Used

A hash table data structure is used in this implementation. It can be represented graphically as follows:

![Fig.4.4 The hash table data structure](image)

This table has two fields. A “Data” field which stores the key bits in the form of 64 bits per entry. It is implemented as eight byte array. Second field is a hash value to locate next 64 bits of the key. A simple array pointer is used for this purpose. A “NULL” value to the next address specifies end of the list. We can add as many 8 byte arrays as we need for required key size. This structure is used as a data type for the C++ class longint along with a 16 bit data type size to specify total number of bits in the key. The longint class contains functions for basic operations on this hash table. The functions to convert a given key into this hash
table format, to copy one hash table into another, to shift right bitwise the entire values and to reverse the hash table entries are some of the examples. The design of data structure provides a unique feature that arithmetic operations can be performed on two different numbers of varying lengths. Below is the explanation of some of the important member functions of the class longint.

**Storing a given block of data using Hash Table:**

Conventionally for cryptographic applications, keys of N bit size are stored as a block in contiguous memory location. While programming for cryptographic applications, this block is converted into an array of bytes. Similarly for this implementation this block has to be converted into hash table format.

Two different functions, `to_longint()` and `add_ary()` are used for converting the block of data representing key into hash table format. `to_longint()` function takes two parameters the pointer to the character string representing key and an integer representing key size. Then, it breaks down the string into chunks of eight bytes and calls `add_ary()` function to store them into new array.

The `add_ary()` function adds a pointer of newly created array into hash table and fills in values into newly created array. The function is capable of filling the array in forward or backward direction.

Most of the arithmetic operations proceed from LSB to MSB; therefore the hash table data structure presented here provides access to lower significant byte first by storing it into first location of first array. But some of the arithmetic operations like division proceed from MSB to LSB; therefore MSB should be first accessible byte. It contrasts to this implementation where LSB is accessible first.
The function `reverse()` is used for this purpose. It simply reverses the order of arrays. For the sake of code simplicity, array values are not reversed as we can access any array value using array index. Care has been taken to restore the reversed arrays to their original order after performing the arithmetic operation, so that data values remain intact.

Some of the arithmetic calculations may require shifting right the number by certain number of bits. Specifically for Montgomery modular multiplication and division, right shifting is used. This may generate empty arrays at the end of table if number of bits to be shifted are more than or equal to 80. Therefore a function `refresh()` is designed to truncate empty arrays. It also resets the size variable accordingly.

There are also other functions like copying one hash table into another etc.

### 4.5 Field Arithmetic

Fields are abstractions of familiar number systems (such as the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$) and their essential properties. They consist of a set $F$ together with two operations, addition (denoted by $+$) and multiplication (denoted by $\cdot$), that satisfy the usual arithmetic properties:

(i) $(F,+)$ is an abelian group with (additive) identity denoted by 0.

(ii) $(F\{0\}, \cdot)$ is an abelian group with (multiplicative) identity denoted by 1.

(iii) The distributive law holds: $(a+b)\cdot c = a\cdot c + b\cdot c$ for all $a, b, c \in F$.

If the set $F$ is finite, then the field is said to be finite.

We are using binary prime fields, which are specific type of finite field with their own formats for arithmetic operations, a separate class `prime_fld` is designed and arithmetic operations are implemented in this class.
4.5.1 Binary Prime Fields (Mathematical Definition)

An element \( \alpha \in \mathbb{F}_q^k \) can be represented as a polynomial with coefficients in \( \mathbb{F}_q \) modulo an irreducible polynomial \( m(X) \in \mathbb{F}_q[X] \) of degree \( k \). If \( \theta \) is a root of \( m(X) \) then \( \{1, \theta, \theta^2, \ldots, \theta^{k-1}\} \) is a basis of \( \mathbb{F}_q^k \) over \( \mathbb{F}_q \). Such a basis is called a polynomial basis. The internal representation of a polynomial is similar to multiprecision integers. Indeed, let be the word size used by the processor. Then a polynomial \( u(X) \) of degree less than \( d \) will be represented as the \( r \)-word vector \((u_r \ldots u_0)\) and the \( j \)-th bit of the word \( u_i \), that is the coefficient of \( u(X) \) of degree \( il + j \), will be denoted by \( u_i[j] \). Many operations on polynomials are strongly related to integer multiprecision arithmetic. In general, we do not have the equivalent of single precision operations. For example, on computers there is usually no hardware multiplication of polynomials in \( \mathbb{F}_2[X] \) of bounded degree, even if this operation is simpler than integer multiplication, since there is no carry to handle.

There are approximately \( q^k/k \) irreducible monic polynomials of degree \( k \) in \( \mathbb{F}_q[X] \). Addition, subtraction, and multiplication are made modulo \( m(X) \). Usually, inversion is obtained with an extended GCD computation in \( \mathbb{F}_q[X] \). As the field polynomial \( m(X) \) is irreducible and the polynomial \( a(X) \) representing the field element \( \alpha \) is of degree less than \( k \), one can find a polynomials \( u(X) \) and \( v(X) \) of degree less than \( k \) such that \( a(X)u(X) + m(X)v(X) = 1 \). Accordingly, \( u(X) = a(X)^{-1} \mod m(X) \).

Any irreducible polynomial of degree \( k \) can be used to define \( \mathbb{F}_q^k \) but in practice polynomials with special properties are chosen. In case of binary fields we choose the polynomial with \( q=2 \).

Arithmetic in extension fields of \( \mathbb{F}_q \) where \( q \) is some power of 2 relies on elementary computer operations like exclusive disjunction and shifts. Note that in
general $q$ is simply equal to 2. This allows very efficient implementations, especially in hardware, and gives finite fields of characteristic 2 a great importance in cryptography. But this research explores the possibility of its implementation in software and explores the pros and cons of considering a hash table data structure for multiprecision integers.

4.5.2 Formal Definition

As $F_2^d$ is a vector space of dimension $d$ over $F_2$, an element can be viewed as a sequence of $d$ coefficients equal to 0 or 1. Therefore it is internally stored as a sequence of bits and the techniques introduced for multiprecision integers apply with some slight modifications. Two kinds of basis are commonly used. In polynomial representation, it is $(1, X, \ldots, X^{d-1})$, whereas with a normal basis it is $(\alpha, \alpha^2, \ldots, \alpha^{2^{d-1}})$

Irreducible polynomial representation

Let $m(X) \in F_q[X]$ be an irreducible polynomial of degree $d$ and $(m(X))$ the principal ideal generated by $m(X)$. Then $F_q[X]/(m(X))$ is the finite field with $q^d$ elements there exists an irreducible polynomial of degree $d$ for each positive $d$ but the proof is not constructive. To find such a polynomial, we can consider a random polynomial and test its irreducibility. Once $m(X)$ has been found, computations are done modulo this irreducible polynomial and reduction is a key operation. For this we need to divide two polynomials with coefficients in a field.

Every irreducible polynomial of degree $d$ can be used to build $F_q^d$; however, some special polynomials offer better performance, e.g., monic sparse polynomials are proposed by R. Schroeppel and others[4].

Usually, one uses trinomials or pentanomials since binomials and quadrinomials are always divisible by $X + 1$ and so, except for $X + 1$ itself, are
never irreducible in $F_q[X]$. The existence for every $d$ of an irreducible degree $d$ trinomial or pentanomial is still an open question, but this is the case at least for all $d \leq 10000$. A trinomial $X^d + X^k + 1$ is reducible if both $d$ and $k$ are even as then $X^d + X^k + 1 = (X^{d/2} + X^{k/2} + 1)^2$

Swan [5] has proved the following:

The trinomial $X^d + X^k + 1$, where at least one of $d$ and $k$ is odd, has an even number of factors if and only if one of the following holds

- $d$ is even, $k$ is odd, $d \neq 2k$ and $dk/2 \equiv 0 \text{ or } 1 \pmod{4}$
- $d$ is odd, $d \equiv \pm 3 \pmod{8}$, $k$ is even and $k$ does not divide $2d$
- $d$ is odd, $d \equiv \pm 1 \pmod{8}$, $k$ is even and $k$ divides $2d$.

It follows that irreducible trinomials do not exist when $d \equiv 0 \pmod{8}$ and are rather scarce for $d \equiv 3 \text{ or } 5 \pmod{8}$.

In Table 4.2, irreducible polynomials over $F_2$ of degree less than or equal to 500 are given. More precisely, the coefficients $d, k_1$ in the table stand for the trinomial $X^d + X^k + 1$. In case there is no trinomial of degree $d$, the sequence $d, k_1, k_2, k_3$ is given for the pentanomial $X^d + X^{k_1} + X^{k_2} + X^{k_3} + 1$. For each $d$ the coefficient $k_1$ is chosen to be minimal, then $k_2$ and so on.
**Table 4.1 Irreducible trinomials and pentanomials over GF(2^m)**

<table>
<thead>
<tr>
<th>m</th>
<th>Trinomials</th>
<th>Pentanomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>123, 124, 125</td>
<td>151, 152, 153</td>
</tr>
<tr>
<td>17</td>
<td>193, 194, 195</td>
<td>217, 218, 219</td>
</tr>
<tr>
<td>23</td>
<td>247, 248, 249</td>
<td>271, 272, 273</td>
</tr>
<tr>
<td>31</td>
<td>331, 332, 333</td>
<td>359, 360, 361</td>
</tr>
<tr>
<td>37</td>
<td>389, 390, 391</td>
<td>417, 418, 419</td>
</tr>
<tr>
<td>41</td>
<td>427, 428, 429</td>
<td>447, 448, 449</td>
</tr>
<tr>
<td>43</td>
<td>451, 452, 453</td>
<td>471, 472, 473</td>
</tr>
<tr>
<td>47</td>
<td>487, 488, 489</td>
<td>507, 508, 509</td>
</tr>
</tbody>
</table>

*Implementation of Work*
4.5.3 General Representation

Finite fields of order $2^m$ are called binary Prime fields or characteristic-two finite fields. They are represented as $F_{2^m}$ or GF($2^m$). One way to construct GF($2^m$) is to use a polynomial basis representation. Here, the elements of $F_{2^m}$ are the binary polynomials (polynomials whose coefficients are in the field $F_2 = \{0, 1\}$) of degree at most $m-1$:

$$F_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \ldots + a_2z^2 + a_1z + a_0 : a_i \in \{0, 1\}\}$$

This means all the values of coefficient $a$ must be either 0 or 1. An example of such a polynomial is 1100110 which is a binary polynomial of order 7 such that

$$GF(7^1) = 1z^6 + 1z^5 + 0z^4 + 0z^3 + 1z^2 + 1z + 0z^0$$

An irreducible binary polynomial $f(z)$ of degree $m$ is chosen (such a polynomial exists for any $m$ and can be efficiently found). Irreducibility means that cannot be factored as a product of binary polynomials each of degree less than $m$. Addition of field elements is the usual addition of polynomials, with coefficient arithmetic performed modulo 2. This means, addition has to be performed taking care that coefficient should remain either 0 or 1, and it should be a polynomial addition. Multiplication of field elements is performed modulo the reduction polynomial. For any binary polynomial $a(z)$, $a(z)$ mod shall denote the unique remainder polynomial $r(z)$ of degree less than $m$ obtained upon long division of $a(z)$ by $f(z)$; this operation is called reduction modulo.

Example: The elements of GF($2^4$) are the 16 binary polynomials of degree at most 3. Binary numbers ranging from (0000 to 1111)
Implemention of Work

Table 4.2. A simple representation of GF(2^4) numbers.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.z^3+0.z^2+0.z^1+0.z^0=</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>0.z^3+0.z^2+0.z^1+1.z^0=</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>0.z^3+0.z^2+1.z^1+0.z^0=</td>
<td>z</td>
<td>0010</td>
</tr>
<tr>
<td>0.z^3+0.z^2+1.z^1+1.z^0=</td>
<td>z+1</td>
<td>0011</td>
</tr>
<tr>
<td>0.z^3+1.z^2+0.z^1+0.z^0=</td>
<td>z^2</td>
<td>0100</td>
</tr>
<tr>
<td>0.z^3+1.z^2+0.z^1+1.z^0=</td>
<td>z^2+1</td>
<td>0101</td>
</tr>
<tr>
<td>0.z^3+1.z^2+1.z^1+0.z^0=</td>
<td>z^2+z</td>
<td>0110</td>
</tr>
<tr>
<td>0.z^3+1.z^2+1.z^1+1.z^0=</td>
<td>z^2+z+1</td>
<td>0111</td>
</tr>
<tr>
<td>1.z^3+0.z^2+0.z^1+0.z^0=</td>
<td>z^3</td>
<td>1000</td>
</tr>
<tr>
<td>1.z^3+0.z^2+0.z^1+1.z^0=</td>
<td>z^3+1</td>
<td>1001</td>
</tr>
<tr>
<td>1.z^3+0.z^2+1.z^1+0.z^0=</td>
<td>z^3+z</td>
<td>1010</td>
</tr>
<tr>
<td>1.z^3+0.z^2+1.z^1+1.z^0=</td>
<td>z^3+z+1</td>
<td>1011</td>
</tr>
<tr>
<td>1.z^3+1.z^2+0.z^1+0.z^0=</td>
<td>z^3+z^2</td>
<td>1100</td>
</tr>
<tr>
<td>1.z^3+1.z^2+0.z^1+1.z^0=</td>
<td>z^3+z^2+1</td>
<td>1101</td>
</tr>
<tr>
<td>1.z^3+1.z^2+1.z^1+0.z^0=</td>
<td>z^3+z^2+z</td>
<td>1110</td>
</tr>
<tr>
<td>1.z^3+1.z^2+1.z^1+1.z^0=</td>
<td>z^3+z^2+z+1</td>
<td>1111</td>
</tr>
</tbody>
</table>

The following are some examples of arithmetic operations in $F_2^4$ with reduction polynomial $f(z) = z^4+z+1(10011=19)$.

Addition: $(z^3+z^2+1)+(z^2+z+1) = z^3+2z^2+z+2$. But we will consider only those elements whose coefficients are either 0 or 1. Therefore;

$(z^3+z^2+1) + (z^2+z+1) = z^3+z$
Now, consider its binary representation,

\[(z^3 + z^2 + 1) = 1101\] and

\[(z^2 + z + 1) = 0111.\]

And result

\[z^3 + z = 1010.\]

From the result, we can observe that ex-or operation is performed between the two numbers \(1101 \oplus 0111 = 1010\). Similar will be the case of subtraction as no borrow will be considered.

Subtraction: \((z^3 + z^2 + 1) - (z^2 + z + 1) = z^3 + z.\) (Since \(-1 = 1 \text{ in } F_2\), we have \(-a = a\) for all \(a \in F_2\)).

And in binary form:

\[1101 - 0111 = 1010 \quad (1101 \oplus 0111 = 1010)\]

This means addition and subtraction are equivalent for characteristic 2 finite fields.

Observe that result is calculated independent of , and still result is modulo to \(f(z)\). That is:

\[(13 + 7) \mod 19 = (1101 \oplus 0111) \mod 10011 = (1010) \mod 10011 = 1010\]

In short, when using binary prime fields, no carry will be propagated among the adjacent bytes.

Multiplication: \((z^3 + z^2 + 1)(z^2 + z + 1) = z^5 + 1\) since

\[(z^3 + z^2 + 1)(z^2 + z + 1) = z^5 + z + 1\]

and

\[(z^5 + z + 1) \mod (z^4 + z + 1) = z^2 + 1\]

Inversion: \((z^3 + z^2 + 1)^{-1} = z^2\) since

\[(z^3 + z^2 + 1)^{-1} \cdot z^2 \mod (z^4 + z + 1) = 1\]
Implementation:

Binary prime fields are implemented as the derived class `prime_fld` from the class `longint`. It uses a single 2 byte integer variable `deg` to represent degree of the binary field. Major functions included here are for binary field base arithmetic like addition, multiplication, division and comparison etc.

Addition is performed simply by x-oring values of two hash tables from LSB to MSB independent of their sizes. The function declaration is as follows.

```c
void add(prime_fld num)
```

This means the function takes only one parameter `num` of the class `prime_fld` and returns no result. The addition is implemented in this unusual way for following two reasons.

1: To save storage and time consumed while defining the return parameter and returning a value from the function respectively. As addition is used repeatedly while doing multiplication and division, it does have an impact on final processing speed and storage requirements.

2: most of the additions while implementing ECC using Montgomery modular multiplication and division are of the form `a=a+b`. This is also true while doing point add and point double of ECC.

In all situations `a` is considered as the object on which function is called and used to store back the result. And `b` is passed as function parameter. Care has been taken that function works properly to handle variable length data. Following situations are considered;
### Table 4.3. Various situations considered to add variable length numbers

<table>
<thead>
<tr>
<th>Condition</th>
<th>Operation(s) performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length($a$) = Length($b$)</td>
<td>Bitwise ex-or</td>
</tr>
<tr>
<td>Length($a$) &gt; Length($b$)</td>
<td>Bitwise ex-or upto Length($b$) only</td>
</tr>
<tr>
<td>Length($a$) &lt; Length($b$)</td>
<td>i. Bitwise ex-or upto Length($a$) &lt;br&gt;ii. Copy remaining bytes of $b$ into $a$</td>
</tr>
<tr>
<td>Length($a$) = 0</td>
<td>Copy $b$ into $a$</td>
</tr>
<tr>
<td>Length($b$) = 0</td>
<td>No operation</td>
</tr>
</tbody>
</table>

#### 4.5.4 Modular Multiplication

It can be defined as:

$$X \div Y \pmod{R}$$

is simply the number $Z$ that satisfies $Z \times Y \equiv X \pmod{R}$.

Let us consider the modular multiplication for decimal numbers for better understanding.

Consider example of $789098 \times 123456 \div 1000000$ to understand modular multiplication. To do this we’ll use the shift-and-add technique, but instead of starting from the left-hand end of 789098 and multiplying by 10 each time, we’ll start from the right-hand end and divide by 10. This is just doing the old multiplication sum backwards, but there is one difference. When we do addition, the carries always propagate from right to left, and in the original method this meant that we couldn’t be certain of the value of any digit of the result until we’d finished the whole calculation. But now we are generating the result from the right-hand end while the carries still flow to the left, and this means that no digit in the result changes once it’s gone past the decimal point. Fig 4.5 describes this technique in detail.
### Implementation of Work

<table>
<thead>
<tr>
<th>Equation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8 \times 123456$</td>
<td>9 8 7 6 4 8</td>
</tr>
<tr>
<td>$\div 10$</td>
<td>9 8 7 6 4 8</td>
</tr>
<tr>
<td>$9 \times 123456$</td>
<td>1 1 1 1 1 0 4</td>
</tr>
<tr>
<td>$9.8 \times 123456$</td>
<td>1 2 0 9 8 6 8</td>
</tr>
<tr>
<td>$\div 10$</td>
<td>1 2 0 9 8 6</td>
</tr>
<tr>
<td>$0 \times 123456$</td>
<td>0</td>
</tr>
<tr>
<td>$0.98 \times 123456$</td>
<td>1 2 0 9 8</td>
</tr>
<tr>
<td>$\div 10$</td>
<td>1 2 0 9 8</td>
</tr>
<tr>
<td>$9 \times 123456$</td>
<td>1 1 1 1 1 0 4</td>
</tr>
<tr>
<td>$9.098 \times 123456$</td>
<td>1 1 2 3 2 0 2</td>
</tr>
<tr>
<td>$\div 10$</td>
<td>1 1 2 3 2</td>
</tr>
<tr>
<td>$8 \times 123456$</td>
<td>9 8 7 6 4</td>
</tr>
<tr>
<td>$8.9098 \times 123456$</td>
<td>1 0 9 9 9 6</td>
</tr>
<tr>
<td>$\div 10$</td>
<td>1 0 9 9 9</td>
</tr>
<tr>
<td>$7 \times 123456$</td>
<td>8 6 4 1 9 2</td>
</tr>
<tr>
<td>$7.89098 \times 123456$</td>
<td>9 7 4 1 8</td>
</tr>
<tr>
<td>$\div 10$</td>
<td>9 7 4 1</td>
</tr>
<tr>
<td>$0.789098 \times 123456$</td>
<td>9 7 4 1 8</td>
</tr>
</tbody>
</table>

**Fig 4.5: Classical Modular Multiplication**
Defining for binary, the objective of an n-bit modular multiplication is to take two n-bit numbers A and B and derive a result $A \cdot B \mod M$ that is at most n-bits as well. This is achieved by subtracting a multiple of the modulus M from $A \cdot B$ such that the result is n-bits wide and also less than M. This differs from conventional multiplication where the product of two n-bit numbers will be at most $2n$ bits wide. For modular multiplication, the product of two n-bit numbers modulo another n-bit number yields a result that is at most n-bits wide. The classical approach for performing modular multiplication involves computing the product $A \cdot B$, and then subtracting a multiple of the modulus M that makes the result to be less than the modulus. An optimization of this approach interleaves the computation of the product and the subtraction of the modulus. The classical algorithm is generally inefficient and very slow. The interleaved classical modular multiplication is presented in Algorithm 4.1.

**Algorithm 4.1: Classical modular multiplication**

Inputs: A, B, M with $0 \leq A, B < M$,

Output: $R = AB \mod M$

$R = 0$;

For $i = n-1$ to 0

Begin

$R = 2 \cdot R + a_i \cdot B$;

$q_i = R \div M$;

$R = R - q_i \cdot M$;

End

4.5.5 The Montgomery Modular Multiplication Algorithm

Another approach to performing modular multiplication is the Montgomery algorithm. The basic idea behind Montgomery multiplication is the fact that one
can add a multiple of the modulus \( M \) to the product \( A \cdot B \) to yield a result that is at most \( 2n+1 \) bits wide. Adding, instead of subtracting, a multiple of the modulus \( M \) does not affect the computation, since the result will be congruent to \( A \cdot B \) modulo \( M \). Two numbers are said to be congruent if their remainder when divided by the modulus is the same. Thus, \( A \cdot B, A \cdot B + M, A \cdot B + 2M \ldots A \cdot B + kM \) are all congruent modulo \( M \). This implies: 

\[
A \cdot B \equiv A \cdot B + M \equiv A \cdot B + 2M \equiv \ldots \ (A \cdot B + kM) \mod M.
\]

In the Montgomery algorithm, the multiple of the modulus \( M \) that is added to \( A \cdot B \) is chosen in such a way that the lower \( n \)-bits of the \( 2n+1 \)-bit result are all zeroes \([6]\). The least significant half of the \( 2n+1 \)-bit result that are all zeroes is then discarded. This way, the result would have been reduced to at most \( n+1 \) bit in width. A single subtraction of the modulus \( M \) can then be performed to further reduce the result to at most \( n \)-bits and make it less than \( M \) if required. It has been shown by Walter \([7]\) that the extra subtraction may not be necessary under certain conditions.

Before understanding how it is done in binary, first, let us consider an example of MMM for decimal numbers:

Suppose that we want to compute \( 789098 \times 123456 \div 1000000 \mod 876543 \)

Let’s go through the multiplication again.

\[
\begin{array}{c}
8 \times 123456 = 987648 \\
\end{array}
\]

Now we want to divide by 10, but we can’t do it because this number isn’t divisible by 10. So we add a multiple of 876543, which we’re allowed to do because we’re working modulo 876543. The multiple must be such that it should make right most digit of the intermediate result zero so that we can discard it. So for above result 987648 right most digit is 8, we have to add a multiple of modulo 0876543 having 2 as its right most digit.
\[ \begin{array}{c|cccccccc} \text{+} & 3 & 5 & 0 & 6 & 1 & 7 & 2 \\ \hline 8 \times 123456 & = & 4 & 4 & 9 & 3 & 8 & 2 & 0 \\ \hline \div 10 = 0.8 \times 123456 & = & 4 & 4 & 9 & 3 & 8 & 2 \\ \hline 9 \times 123456 & + & 1 & 1 & 1 & 1 & 0 & 4 \\ \hline 9.8 \times 123456 & = & 1 & 5 & 6 & 0 & 4 & 8 & 6 \\ \hline \text{Another multiple of 876543} + & 7 & 0 & 1 & 2 & 3 & 4 & 4 \\ \hline = & 8 & 5 & 7 & 2 & 8 & 3 & 0 \\ \hline \div 10 = 0.98 \times 123456 & = & 8 & 5 & 7 & 2 & 8 & 3 \\ \hline 0 \times 123456 & = & + & 0 \\ \hline 0.98 \times 123456 & = & 8 & 5 & 7 & 2 & 8 & 3 \\ \hline \text{Another multiple of 876543} + & 7 & 8 & 8 & 8 & 8 & 8 & 7 \\ \hline = & 8 & 7 & 4 & 6 & 1 & 7 & 0 \\ \hline \div 10 = 0.098 \times 123456 & = & 8 & 7 & 4 & 6 & 1 & 7 \\ \hline 9 \times 123456 & + & 1 & 1 & 1 & 1 & 0 & 4 \\ \hline 9.098 \times 123456 & = & 1 & 9 & 8 & 5 & 7 & 2 & 1 \\ \hline \text{Another multiple of 876543} + & 2 & 6 & 2 & 9 & 6 & 2 & 9 \\ \hline = & 4 & 6 & 1 & 5 & 3 & 5 & 0 \\ \hline \div 10 = 0.09098 \times 123456 & = & 4 & 6 & 1 & 5 & 3 & 5 \\ \hline 8 \times 123456 & + & 9 & 8 & 7 & 6 & 4 & 8 \\ \hline 8.9098 \times 123456 & = & 1 & 4 & 4 & 9 & 1 & 8 & 3 \\ \hline \text{Another multiple of 876543} + & 7 & 8 & 8 & 8 & 8 & 8 & 7 \\ \hline = & 9 & 3 & 3 & 8 & 0 & 7 & 0 \\ \hline \div 10 = 0.89098 \times 123456 & = & 9 & 3 & 3 & 8 & 0 & 7 \\ \hline 7 \times 123456 & + & 8 & 6 & 4 & 1 & 9 & 2 \\ \hline 7.89098 \times 123456 & = & 1 & 7 & 9 & 7 & 9 & 9 & 9 \\ \hline \text{Another multiple of 876543} + & 6 & 1 & 3 & 5 & 8 & 0 & 1 \\ \hline = & 7 & 9 & 3 & 3 & 8 & 0 & 0 \\ \hline \div 10 = 0.789098 \times 123456 & = & 7 & 9 & 3 & 3 & 8 & 0 \\ \hline \end{array} \]

**Fig 4.6: Montgomery modular multiplication**

164

Department of Computer Science & IT
It’s easier to explain in decimal, but Montgomery multiplication is easier to implement in binary [8]. The place of 1000000 is taken by some suitable power of 2, but the key simplification is that the adding of the multiple of the modulus becomes much easier. The rule is this: if the number you are looking at is odd, add R before you halve it; if it’s even, just halve it.

Montgomery’s approach in essence achieves the objective of modular multiplication which is to take two n-bit numbers, multiply them and derive a result that is at most n-bits wide. The resulting n-bit number is not exactly A·B mod M. It is referred to in the literature as a Montgomery product A·B 2-n mod M. However, most cryptographic schemes make use of repeated modular multiplications such as modular exponentiation – A^e mod M. Montgomery multiplication can then be used in performing the repeated multiplications, and only the final result of the exponentiation is converted back from the Montgomery domain. Thus, the cost of conversion to and from the Montgomery domain is amortized over the repeated modular multiplications. The conversion from the Montgomery domain is just another Montgomery multiplication by 2^{2n}. Just like the classical modular algorithm, Montgomery’s algorithm can also be performed in a fashion whereby the computation of the product and the addition of the modulus are interleaved. The interleaved Montgomery modular multiplication is presented in Algorithm 4.2.

**Algorithm 4.2: Montgomery multiplication**

Inputs: A, B, M with 0 ≤ A, B < M,

Output: R = Montgomery Product (A·B 2^{-n}) mod M

R = 0;

For i = 0 to n-1 do
In each iteration of the loop, the least significant bit of the intermediate result is inspected. If it is ‘1’, i.e. the intermediate result is odd; we add the modulus M to make it even. This is possible since M is guaranteed to be odd in the cryptographic applications of interest. Thus, at each step the intermediate result is made to be even. This even number can be divided by 2 without any remainder. This division by 2 reduces the intermediate result to \( n + 1 \) bits again.

Dividing the intermediate result by 2 is equivalent to discarding the current least significant bit of the intermediate result that is zero. After \( n \) steps these divisions add up to one division by \( 2^n \), or discarding the least significant \( n \)-bits that are zeroes.

Next chapter explains how this function is implemented for binary prime fields.

4.5.6 Modular Division

The Modular division algorithm is based on Extended Euclidean Algorithm for GCD. It is as follows;

Let a and b be integers, not both 0. The greatest common divisor (GCD) of a and b, denoted GCD\((a,b)\), is the largest integer \( d \) that divides both \( a \) and \( b \). Efficient algorithms for computing GCD\((a,b)\) exploit the following simple result.
Theorem 4.1: Let a and b be positive integers.

Then \( \text{GCD}(a,b) = \text{GCD}(b-ca,a) \) for all integers \( c \).

In the classical Euclidean algorithm for computing the GCD of positive integers \( a \) and \( b \) where \( b \geq a \), \( b \) is divided by \( a \) to obtain a quotient \( q \) and a remainder \( r \) satisfying \( b = qa + r \) and \( 0 \leq r < a \). By Theorem 4.1, \( \text{GCD}(a,b) = \text{GCD}(r,a) \). Thus, the problem of determining \( \text{GCD}(a,b) \) is reduced to that of computing \( \text{GCD}(r,a) \) where the arguments \( (r,a) \) are smaller than the original arguments \( (a,b) \). This process is repeated until one of the arguments is 0, and the result is then immediately obtained since \( \text{GCD}(0,d) = d \). The algorithm must terminate since the non-negative remainders are strictly decreasing.

Moreover, it is efficient because the number of division steps can be shown to be at most \( 2k \) where \( k \) is the bitlength of \( a \).

The Euclidean algorithm can be extended to find integers \( x \) and \( y \) such that \( ax + by = d \) where \( d = \text{GCD}(a,b) \). Algorithm 4.3 maintains the invariants

\[ ax_1 + by_1 = u, \ ax_2 + by_2 = u, \ u \leq v. \]

The algorithm terminates when \( u = 0 \), in which case \( v = \text{GCD}(a,b) \) and \( x_1 = x_2, \ y_1 = y_2 \) satisfy \( ax + by = d \).

Algorithm 4.3 Extended Euclidean algorithm for integers

INPUT: Positive integers \( a \) and \( b \) with \( a \leq b \).

OUTPUT: \( d = \text{GCD}(a,b) \) and integers \( x, y \) satisfying \( ax + by = d \).

\[ u \leftarrow a, \ v \leftarrow b. \]

\[ x_1 \leftarrow 1, \ y_1 \leftarrow 0, \ x_2 \leftarrow 0, \ y_2 \leftarrow 1. \]

While \( u \neq 0 \) do

\[ q \leftarrow \left\lfloor v/u \right\rfloor, \ r \leftarrow v-qu, \ x \leftarrow x_2-qx_1, \ y \leftarrow y_2-qy_1. \]
\[ v \leftarrow u, \quad u \leftarrow r, \quad x_2 \leftarrow x, \quad x_1 \leftarrow x, \quad y_2 \leftarrow y, \quad y_1 \leftarrow y. \]

\[ d \leftarrow v, \quad x_2 \leftarrow x, \quad y_2 \leftarrow y. \]

Return \((d, x, y)\).

This algorithm can be modified to calculate inverse and division in binary field. Here division can be replaced by right shift operation. It will cut down the cost of division.

One such algorithm for hardware implementation is presented by S. C. Shantz in [9]. It is based on following assumption;

Let \( M(t) \) be the irreducible polynomial of the field \( \text{GF}(2^m) \).

Let \( y(t), x(t) \) be the input polynomials.

We need to compute an output \( r(t) \), the residue, which is defined as

\[ r(t) = \frac{y(t)}{x(t)} \mod M(t), \quad y(t) = r(t)x(t) \mod M(t) \]

Now, initialize four auxiliary registers \( A, B, U, \) and \( V \) with values:

\[ A \leftarrow x(t), B \leftarrow M(t), \quad U \leftarrow y(t), \quad V \leftarrow 0 \]

The routine iteratively reduces the polynomials in \( A \) and \( B \) down to 1 while adjusting the values in \( U \) and \( V \) accordingly in order to maintain the following invariant relationship between the four registers:

\[ A \cdot y(t) \equiv U \cdot x(t) \mod M(t) \quad \text{and} \quad B \cdot y(t) \equiv V \cdot x(t) \mod M(t) \]

The Algorithm is as follows;

**Algorithm 4.4: Modular Division \( \text{GF}(2^m) \)**

Begin

\[ A \leftarrow x(t), B \leftarrow M(t), \quad U \leftarrow y(t), \quad V \leftarrow 0 \]

while ( \( A \) not equal \( B \) ) do

Begin
The algorithm is an iterative process of additions, parity testings, and shifts. The process iterates while steadily reducing $A$ and $B$. In every iteration, either $A$ or $B$ is reduced by one bit, thus, the entire division routine takes no more than $2(m-1)$ iterations. When the routine terminates, register $U$ holds the result of the modular division. The degree of $U$ and $V$ is bounded by $M$ throughout the entire process; therefore, $U$ and $V$ never exceed $m$ bits. The least significant bit of $U$ and $V$ are zeroed prior to a reduction by adding $M$. This can be done because adding an irreducible polynomial of the field is algebraically equivalent to adding zero to the system.
4.6 Elliptic curve Arithmetic and Related Implementation

An elliptic curve $E$ over a field $K$ denoted by $E/K$ as discussed in previous chapters is:

$$E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6 \quad (4.6.1)$$

This equation is called a Weierstrass equation.

Where the coefficients $a_1, a_2, a_3, a_4, a_6 \in K$ are such that for each point $(x_1, y_1)$ with coordinates in $K$ satisfying (4.6.1), the partial derivatives

$$2y_1 + a_1 x_1 + a_3 \text{ and } 3x_1^2 + 2a_2 x_1 + a_4 - a_1 y_1$$

do not vanish simultaneously. The last condition says that an elliptic curve is nonsingular or smooth.

A point on a curve is called singular if both partial derivatives vanish. The $\Delta$ is the discriminant of $E$.

The field $K$ for Weierstrass function is usually taken to be the complex numbers $\mathbb{C}$, reals $\mathbb{R}$, rationals $\mathbb{Q}$, algebraic extensions of $\mathbb{Q}$, $\rho$-adic numbers $\mathbb{Q}_\rho$, or a finite field. The above equation is obtained for any general finite field.

For this implementation finite fields of the form $GF(2^m)$ are considered. The simplified elliptic curve equation for $GF(2^m)$ is given by:

$$E : y^2 + a_1 y = x^3 + ax^2 + b$$

Following is the group law for this form of elliptic curves

1. Identity. $P + \infty = \infty = \infty + P$ for all $P \in GF(2^m)$.

2. Negatives. If $P = (x, y) \in GF(2^m)$, then $(x, y) + (x, x + y) = \infty$. The point $(x, x + y)$ is denoted by $-P$ and is called the negative of $P$; $-P$ is indeed a point in $GF(2^m)$. Also, $-\infty = \infty$.
3. **Point addition:** Let \( P = (x_1, y_1) \in GF(2^m) \) and \( Q = (x_2, y_2) \in GF(2^m) \), where \( P \neq \pm Q \). Then \( P + Q = (x_3, y_3) \), where

\[
x_3 = \lambda^2 + \lambda x_1 + x_2 + a \quad \text{and} \quad y_3 = \lambda (x_1 + x_2) + x_3 + y_1
\]

4. **Point double:** Let \( P = (x_1, y_1) \in GF(2^m) \) where \( P \neq -P \). Then \( 2P = (x_3, y_3) \), where

\[
x_3 = \lambda^2 + \lambda + a = \frac{x_1^2 + b}{x_1^2} \quad \text{And} \quad y_3 = x_1^2 + \lambda x_1 + x_3 \quad \text{with} \quad \lambda = x_1 + \frac{y_1}{x_1}
\]

### 4.6.1 Choice of Coordinates

While implementing ECC, it is also required to select proper coordinate system as it affects a lot on the final speed of various arithmetic operations. Two different coordinate systems affine vs. projective are considered for this implementation. Below is the short description of affine space and projective space.

**Affine Space:**

An affine space can be thought of as any space containing points and vectors together, with following rules:

- Any point plus a vector gives a point
- Difference of two points is a vector
- Points cannot be added
- There is no distinguished point that serves as an origin

Affine coordinates is a coordinate system for the \( \mathbb{R}^n \) that is determined by any basis of \( n \) vectors, which are not necessarily orthonormal. Therefore, the resulting axes are not necessarily mutually perpendicular nor have the same unit measure.
Although points cannot be added in affine space but the uniqueness of elliptic curve group law allows point addition for two points situated on elliptic curve to get a third point.

**Projective space:**

Projective space is a geometric object of one dimensional vector subspace of a given vector space. For constructing a real projective plane, following three formal definitions are considered.

- The set of lines in \( \mathbb{R}^3 \) passing through origin \((0,0,0)\). Every such line meets the sphere of radius one centered in the origin exactly twice, say in \( P=(x,y,z) \) and its antipodal points \((-x,-y,-z)\).

- \( P^2(\mathbb{R}) \) can also be described to be the points on the sphere \( S^2 \), where every point \( P \) and its antipodal point are not distinguished.

- It can also be defined as a set of equivalent classes of \( \mathbb{R}^3(0, 0, 0) \) i.e. 3 spaces without the origin, where two points \( P=(x, y, z) \) and \( P’=(x’, y’, z’) \) are equivalent iff there is a nonzero real number \( \lambda \) such that \( P=\lambda P’ \) (\( x=\lambda x’, y=\lambda y’, z=\lambda z’ \)). Any point of a 3D coordinate system is represented in projective space as \([X:Y:Z]\)

In an informal way, projective planes are used to formalize the statements like “Parallel lines intersect at infinity”. Projective space is related to “perspective”, the way an eye or a camera projects a 3D scene to a 2D image. Following figure gives the example;[10]
However speaking about affine and projective spaces in Elliptic Curve Cryptography, the simple thing to remember is

\[
\text{Affine } \longleftrightarrow (x, y) \quad \text{Projective } \longleftrightarrow [X: Y: Z]
\]

With the relation that a point \(P(X, Y, Z)\) in projective space is equivalent to a point \(P(X/Z, Y/Z)\) in affine space. Projective points are obtained as projections of affine space into plane \(z=1\). Hence each projective points can be considered as affine point with coordinates \([x, y, 1]\).

The Elliptic curve equation for \(\text{GF}(2^m)\) under projective coordinates is represented as

\[
Y^2Z+XYZ=X^3+ax^2Z+bZ^3
\]

The equations for point addition and point doubling are simplified as below

**Point Addition:**

Let \(P(X_1; Y_1; Z_1)\) and \(Q(X_2; Y_2; Z_2)\) such that \(P \neq \pm Q\) then

\[
P+Q=(X_3; Y_3; Z_3)
\]

is given by
\[ A = YZ_2 + ZY_2, \]
\[ B = XZ_2 + ZX_2, \]
\[ C = B^2 \]
\[ D = Z_2, \]
\[ E = (A^2 + AB + bC)D + BC, \]
\[ X_3 = BE, \quad Y_3 = C(AX_1 + Y_1)Z_2 + (A + B)E \quad Z_3 = B^3D \]

**Point Double:**

Let \( P(X_1:Y_1:Z_1) \) then \( 2P = (X_3:Y_3:Z_3) \) is given by

\[ A = X_1^2 \]
\[ B = A + Y_1Z_1 \]
\[ C = X_1Z_1 \]
\[ D = C^2 \]
\[ E = (B^2 + BC + aD) \]
\[ X_3 = CE \quad Y_3 = (B + C)E + A^2C \quad Z_3 = CD \]

In projective coordinates, division is not required. It enhances the speed as division is the most time consuming operation in binary or even finite field arithmetic. The addition needs \( 16M + 2S \) operations where \( M \) stands for multiplication and \( S \) stands for squaring. Squaring algorithms are not considered in this implementation so an addition requires \( 18M \) operations.

Similarly, doubling requires \( 8M + 4S = 12M \) operations. For NIST curves, coefficient \( a = 1 \). It saves one multiplication. If the addition receives one input in
Implementation of Work

affine coordinated, i.e. \((X_2; Y_2; 1)\) the cost reduces to \(12M+2S\). This means we can perform addition in mixed coordinates. Such point addition is given by [11];

\[
A = Y_i + Y_2Z_i^2
\]

\[
B = X_1 + X_2Z_1
\]

\[
C = BZ_1
\]

\[
Z_3 = C^2
\]

\[
D = X_2Z_3
\]

\[
X_3 = A^2 + C(A + B^2 + aC)
\]

\[
Y_3 = (D + X_3)(AC + Z_3) + (Y_2 + X_2)Z_3^2
\]

It requires only \(9M+5S\).

**Point Multiplication:**

Finally we need to calculate \(Q=kP\) where \(k\) is an integer and \(P\) is an elliptic curve point. This operation has the largest impact on ECC schemes. Note that lengths of \(k\) and \(P\) need not be equal. There are various algorithms available for this purpose. Schemes are also needed to generate random numbers from which we select a \(k\) for every new key.

Progress in generating random number sequences has been significant. However, people are still trying to figure out new methods for producing fast, cryptographically secure random bits. Before the first table of random numbers was published in 1927, researchers had to work with very slow and simple random number generators (RNG), like tossing a coin or rolling a dice. Needless to say, these methods were very time consuming. It was not until 1927 when Tippetts published a table of 40,000 numbers derived from the census reports that people had access to a large sequence of random numbers. Then research advanced to...
generate random numbers computationally. It is further branched in two directions, generation of random numbers algorithmically and sampling physical data. Today two types of random numbers generators are used.

*Pseudorandom Number Generator* that uses computers arithmetic operation to create deterministic random sequences.

*True Random Number Generator* that uses physical data like noise generated by a transistor or dual oscillator, capturing keystrokes or other user inputs etc.

There are various algorithms available for point multiplication. One of such algorithm is given below. it is called right to left binary method for point multiplication

**Algorithm 4.5 Right-to-left binary method for point multiplication**

**INPUT:** $k = (k_{t-1}, \ldots, k_1, k_0); \ P \in \mathbb{E}(F_q)$.

**OUTPUT:** $kP$.

$Q \leftarrow \infty$.

For $i$ from 0 to $t-1$ do

- If $k_i = 1$ then $Q \leftarrow Q + P$.
- $P \leftarrow 2P$.

Return($Q$).

Starting from lsb, for each bit $k_i$ of the integer $k$, if $k_i=1$, point addition is performed followed by point doubling. This means if $k_i$ is 0 only point doubling is performed.

**4.7 Conclusion**

Implementation of the research work is described in this chapter. It begins with a justification for AVR as a chosen platform and C++ as the preferred language for embedded systems. It followed by the outline of data structure and methodology...
of its implementation. Then, arithmetic operations related to binary field are discussed and various algorithms selected while implementing binary field are explained. It is followed by details regarding elliptic curve arithmetic operations. Two coordinates systems, affine vs. projective are considered and implementation is done in projective and mixed coordinates. Next chapter will describe the speed and code optimizations of implemented algorithms along with details of their final implementation structure.
Chapter 4

References:


[10]. www.saintoflightanddark.deviantart.com