Constructing Cluster of Simple FPGA boards for Cryptologic Computations

by

YARKIN DORÖZ

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

Fall, 2011

CONSTRUCTING CLUSTER OF SIMPLE FPGA BOARDS FOR CRYPTOLOGIC COMPUTATIONS

Yarkın DORÖZ

Computer Science and Engineering, Master's Thesis, 2011

Thesis Supervisor: Assoc. Prof. Dr. Erkay Savaş

Keywords: Reconfigurable Computing for Cryptography, Cryptanalytic Attacks, Cryptographic Acceleration, Hardware Implementations of Cryptographic Algorithms

Abstract

In this thesis, we propose an FPGA cluster infrastructure which can be utilized in implementing cryptanalytic attacks and accelerating cryptographic operations. The cluster can be formed using simple and inexpensive, off-the-shelf FPGA boards featuring an FPGA device, local storage, CPLD, and network connection. Forming the cluster is simple and no effort for the hardware development is needed except for the hardware design for the actual computation. Using a softcore processor on FPGA, we are able to configure FPGA devices dynamically and change their configuration on the fly from a remote computer. The softcore on FPGA can execute relatively complicated programs for mundane tasks unworthy of FPGA resources. Finally, we propose and implement a fast and efficient dynamic *configuration* *switch technique* that is shown to be useful especially in cryptanalytic applications. Our infrastructure provides a cost-effective alternative for formerly proposed cryptanalytic engines based on FPGA devices.

KRİPTOLOJİK HESAPLAMALAR İÇİN, BASİT SPKD ÇEVRİM KARTLARINDAN OLUŞMUŞ KÜMELERİN GERÇEKLENMESİ

Yarkın DORÖZ

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2011

Tez Danışmanı: Doç. Dr. Erkay Savaş

Anahtar Kelimeler: Kriptografi İçin Yeniden Yapılandırabilinen Hesaplama, Kripto-analitik Saldırılar, Kriptografik Hızlandırma, Kriptografik Algoritmaların Donanımsal Gerçeklenmesi

Özet

Bu tez ile, kripto-analitik saldırıların gerçeklenmesi ve kriptografik operasyonların hızlandırılması için tasarlanmış, SPKD'lerden (Sahada Programlanabilir Kapı Dizileri) oluşan bir küme altyapısı sunuyoruz. Bahsi geçen küme altyapısı, SPKD cihazı, yerel depolama, KPMC (Karmaşık Programlanabilir Mantıksal Cihaz) ve ağ bağlantısı içeren ucuz ve kullanıma hazır SPKD çevrim kartlarından oluşmaktadır. Küme oluşturma işleminin basit olmasının yanısıra hesaplamalar için gerekli olan donanım tasarımı hariç herhangi bir donanım geliştirme gerektirmemektedir. SPKDlerde gerçeklenebilen bir işlemci çekirdeği sayesinde, SPKD cihazlarını dinamik olarak yapılandırmak ve hatta yapılandırma ayarlarını, işlem sırasında bile, uzaktaki bir bilgisayar üzerinden değiştirmek mümkündür. Aynı zamanda bu işlemci çekirdeği, karmaşık programları SPKDnin kaynaklarını kullanmaksızın yürütülebilmektedir. Ek olarak, bu tez ile dinamik yapılandırma değişim tekniği de öneriyoruz. Uygulamasını gerçeklediğimiz bu teknik özellikle kripto-analitik saldırılarda hızlı ve verimli bir şekilde kullanılabildiğinden, SPKD tabanlı geleneksel kripto-analitik makinalara göre daha uygun maliyetli bir alternatif oluşturmaktadır.

to my beloved family

Acknowledgements

I wish to express my sincere gratitude to Assoc. Prof. Erkay Savaş, for his continuous support and worthwhile guidance throughout my masters studies. He always believed in my skills and he was eager to help me with problems regarding my thesis. Also I am thankful to my thesis defense committee members: Assoc. Prof. Dr. Albert Levi, Assoc. Prof. Dr. İbrahim Tekin, Assoc. Prof. Dr. Yücel Saygın, Assist. Prof. Dr. Cemal Yılmaz for their support and presence.

I would also like to thank to my friends Duygu Karaoğlan, Barış Altop, Burcu Özçelik, Erdal Mutlu, Cengiz Örencik, Erman Pattuk, Leyli Javid Khayati for their help and support in my academic life. Yonca Betül Karadeniz deserve special thanks for her precious and eternal support.

Finally, I want to thank my family for their valuable support and being there when I needed them. I am very grateful for their presence and their respect on my decisions for academic life.

Contents

1 Introduction **1**

2 Background Information **5**

2.1 Modular Arithmetic 5

2.1.1 Modular Multiplication 5

2.1.2 Modular Inversion 6

2.1.3 Modular Addition/Subtraction 8

2.1.4 Montgomery Form 10

2.2 Asymmetrical Cryptography 11

2.3 Elliptic Curve Cryptography 12

2.3.1 Curve Property 12

2.3.2 Point Property 13

2.3.3 Point Addition 13

2.3.4 Point Doubling 14

2.3.5 Scalar Multiplication 16

2.3.6 Order of an Elliptic Curve Group 16

2.3.7 Elliptic Curve Discrete Logarithm Problem (ECDLP) . . . 18

2.3.8 Elliptic Curve Diffie-Hellman 18

2.4 Huff Curves 19

2.4.1 Huff Curve Point Addition/Doubling 20

2.5 Pollard Rho Attack 21

2.5.1 Distinguished Point Property 25

2.5.2 Random Function 26

3 Programmable Logic Devices **27**

3.1 FPGA 28

3.1.1 Microblaze (Softcore) 28

3.1.2 Executable and Linkable Format for Microblaze 31

3.2 CPLD 33

3.3 Related Works 34

4 General Overview of the Proposed Scheme **38**

4.1 Proposed Architecture for FPGA cluster 38

4.2 Our Scheme and Its Operational Steps 41

4.3 Using FPGA Cluster for Acceleration of Cryptographic Computations 44

4.4 Using FPGA Cluster for Cryptanalysis 45

5 Implementation and Experimental Results **48**

5.1 Software Implementations 48

5.1.1 Bootloader 49

5.1.2 Cluster-Head 51

5.1.3 Proxy Code 58

5.2 Hardware Implementations 62

5.2.1 Inversion 63

5.2.2 Point Addition 68

5.2.3 Pollard Rho 73

5.3 Experiments 78

5.3.1 First Experiment 79

5.3.2 Second Experiment 80

5.3.3 Third Experiment 81

5.3.4 Fourth Experiment 82

5.3.5 Fifth Experiment 84

6 Conclusion and Future Works 88

List of Figures

1 Point Addition 14

2 Point Doubling 15

3 Point Behaviour 22

4 Parallel Point Collusion Calculation 23

5 General overview of the FPGA cluster 38

6 Components of the FPGA board 40

7 Memory map of the parallel flash and placement of configuration and data 42

8 The Protocol of Read Operation 55

9 The Protocol of Write Operation 56

10 U-V Architecture 66

11 R-S Architecture 67

12 Point Addition Architecture 72

13 Calculation Module 76

14 Pollard Rho Attack Architecture 77

15 Experimental Setup of FPGA cluster 85

16 Timing statistics (in seconds) for Pollard Rho's alg. on different number of FPGA boards 86

List of Tables

1 Timing overhead (in seconds) for different data and packet sizes . 80

2 Experimental results for exhaustive key search 83

3 Critical resource usage of the distinguished points-generating engine on different FPGA devices 87

1 Introduction

Cryptographic operations usually contain high degree of parallelism, which favors repetitive instantiation of the same basic block for the cryptographic primitives. Thus, hardware-based cryptographic accelerators, harnessing the aforementioned parallelism, have become the focus of both industrial and academical interests especially in the last two decades.

Cryptanalytic studies aim to discover the strength of cryptographic algorithms against certain attack techniques, efficiency of which is determined by, to a large extent, amount of computational power available at affordable costs. As it is possible to make relatively accurate predictions (at least so far) for the increase in computational power and decrease in their associate costs in future (e.g. Moore's Law), we can provide some predictions for the future strength of certain cryptographic algorithms and their key lengths. Moreover, since increase in raw computational power does not necessarily lead to the same level of increase in our capacity for breaking ciphers, it is important to work on new architectures that will make an efficient use of the new *computing capabilities*. As in the case of cryptographic operations, even more so, cryptanalytic applications are characterized by a vast degree of inherent parallelism that can be utilized best by hardware architectures.

Recent developments in Field Programmable Gate Array (FPGA) technology, especially in terms of increased resources and declining costs, emphasize the configurable logic devices as the economic alternative for both cryptographic acceleration and cryptanalytic computations. Grasping this great potential, previous works in literature propose FPGA-based designs and architectures for both cryptographic acceleration [26, 20] and cryptanalytic purposes [19, 7, 8]. Combining the power of hardware, especially by taking advantage of the parallelism, with the flexibility of software based design through rapid prototyping, FPGAs are yet to offer their great potential in cryptologic applications.

Nowadays, many FPGAs can be configured to implement microprocessor cores that can handle mundane tasks, which are not performance bound and unworthy of valuable FPGA resources. Implementing a communication protocol such as TCP/IP stack or interacting with peripherals are examples of such tasks. MicroBlaze, which is a soft processor core (referred as *softcore* henceforth) by Xilinx and can be implemented even on the most inexpensive FPGAs using the general-purpose logic available on all FPGAs [37], can be utilized in this context.

In addition, FPGAs can be *dynamically* configured to implement multiple hardware designs. Relatively fast dynamical switching between configurations provides agility as well as flexibility to meet computational diversity of cryptologic applications. Moreover, the configuration files for multiple designs can be sent over a network (if the device is connected). If different configuration files are stored in a local memory, the configuration switch between hardware designs can be very fast.

Last, but not the least, simple FPGA boards featuring low-end FPGA devices such as Spartan-3E [36] offer network interface and local storage, and come at a very low cost. One such board, Spartan-3E Starter Kit [38], proves that it is possible to have the best of both software and hardware worlds in a very cost-effective manner.

In this thesis, we propose an *FPGA cluster* for cryptologic purposes using Spartan-3E family FPGAs. Our approach differs from similar and the closest works in [19, 7, 8] in the sense that a super computer does from a server cluster. Our FPGA cluster can be formed using a host-PC acting as the *cluster head* and any off-the-shelf FPGA board featuring an FPGA, a network interface, local storage, and a simple Complex Programmable Logic Device (CPLD). A router that provides fast connection is beneficial, but not necessary.

The cluster head is not only responsible for coordinating the computational activities, but also for configuring the FPGA devices in the cluster. In every FPGA, there is one permanent configuration for the softcore stored in the flash memory. The softcore runs on an FPGA in idle times, and performs essential tasks in loading new hardware configurations besides other non-specialized tasks. The cluster head can also send code segments for the softcore to run software applications contributing to the agility of the overall system.

The proposed cluster can be efficiently used for cryptanalytic purposes such as exhaustive search. For certain cases, it can also be used as an accelerator to speedup the cryptographic applications. By supporting a fast, dynamic configuration switch, each FPGA board can combine the versatility of general-purpose computer with the parallel computing capability of hardware designs, even for FPGAs in the low-end of the cost spectrum. The software components we develop and denote as proxies running both in the cluster head and the softcore enables a transparent programing experience similar to the one provided only by middleware for parallel programing and remote procedure call.

Outline of the thesis is as follows: Section 2 gives background information on efficient modular arithmetic, followed by elliptic curve cryptography and Pollard Rho Attack. Section 3 gives information about the Programmable Logic Devices that are used in the process and their features in detail. Section 4 provides the details of the proposed architecture for FPGA cluster, its operational steps, usage of the proposed cluster for cryptographic acceleration and cryptanalysis. Implementation details and experimental results are provided in Section 5. Finally, Section 6 concludes the thesis by summarizing the achievements.

2 Background Information

In this section, we start by giving background information on asymmetrical cryptography and give mathematical background on the modular arithmetic algorithms. Later, we give details on elliptic curves and one of its forms known as Huff curves. Afterwards, we describe the usage of Pollard Rho attack algorithm and give features of the reconfigurable logic devices and softcores on FPGAs. Lastly, we give detailed information on related works.

2.1 Modular Arithmetic

In this section we give detailed information on the algorithms for big integer modular arithmetics, such as modular multiplication, modular inversion and modular addition/subtraction. Efficient algorithms are introduced in the following sections for each of the modular arithmetic operations in order to perform faster arithmetics on computers and electronic devices. Also, these algorithms require form conversions for their operands which is described in Section 2.1.4.

2.1.1 Modular Multiplication

Modular multiplication is a costly operation, especially the cryptographic operations are taken into account. For instance, in ECRYPT II [12] the suggested key length for legacy RSA systems is $N \geqslant 1024$ bits and for new systems is $N \geqslant 2432$ bits. The multiplication of such big numbers have huge costs both in software and in hardware.

There are different methods to make modular multiplication efficient and one of the fastest algorithms is Montgomery multiplication [23]. The algorithm simply utilizes addition, subtraction and shifting operations to perform modular multiplication. Montgomery multiplication exchanges the division operation of the classical methods, which is used for reduction, with simple shift operations, so that the algorithm works faster. On the down side of the algorithm, the multiplier and the multiplicand should be converted into another form which is called Montgomery form. The conversion operation along with the multiplication operation is costlier than classical modular multiplications. However, for operations require multiple multiplications, such as modular exponentiation, only two conversion operations are required and therefore extra costs due to conversions are negligible. Therefore, Montgomery multiplication is beneficial when many multiplication operations in a cryptographic algorithm is required. The steps of Montgomery multiplication given in Algorithm 1 is defined in equation (1) (T and S are two numbers that will be multiplied and are smaller than the modulus N):

$$Modular\ Multiplication(T, S, N) = 2^{-k}\ T\ S \mod N \tag{1}$$

2.1.2 Modular Inversion

Modular inversion operation is frequently used in cryptographic applications. Since it is usually the costliest arithmetic operation when implemented both in hardware and software, an efficient inversion algorithm is needed in order to increase the performance of cryptosystems. The Montgomery inversion [15] is one of the

Algorithm 1 Modular Multiplication

```
Input: $X = Multiplicand,\ (X_{k-1}, X_{k-2}, X_{k-3}, \ldots X_2, X_1, X_0),\quad X < N$
       $Y = Multiplier,\quad Y < N$
       $N = Modulus$
       $k = Bit\ Size\ of\ N$
Output: $R = X\ Y\ 2^{-k} \mod N$
  $R \leftarrow 0$
  for $i = 0 \rightarrow k - 1$ do
    if $X_i == 1$ then
      $R \leftarrow R + Y$
    end if
    if $R_0 == 1$ then
      $R \leftarrow R + N$
    end if
    $R \leftarrow R/2$
  end for
  if $R > N$ then
    $R \leftarrow R - N$
  end if
  return $R$
```

fastest algorithm to perform inversion of big numbers.

Similar to the modular multiplication, the division operation is replaced by simple shift operations. The operand also needs to be converted into Montgomery form in order to be used in the Montgomery inversion algorithm. Conversion of an operand to the Montgomery form and from the Montgomery form are costly operations, but Montgomery inversion can be used in conjunction with Montgomery multiplication in cryptographic applications, which will eventually have a higher performance over classical modular multiplication and inversion methods. The Montgomery inversion algorithm is composed of two phases and the output of the first algorithmic step is used as an input by the second algorithmic step. These phases are given in Algorithm 2 and Algorithm 3 which computes the following:

$$Mod\ Inv(R) = 2^{-2k}\ R \mod N \tag{2}$$

where $R = 2^m,\ m = \lceil log_2 N \rceil$.

2.1.3 Modular Addition/Subtraction

Modular addition and subtraction are the simplest modular arithmetic operations. They can be implemented efficiently both in hardware and software. Even for large bit sizes, addition and subtraction operation can be implemented in a few clock cycles in hardware. The modular addition and subtraction operations are divided into blocks and every block is processed one at a time to have a higher clock frequency and lower the area size in hardware.

Algorithm 2 Modular Inversion / Phase - I

```
Input: $X = Operand$
       $N = Modulus$
       $m = Bit\ Size\ of\ N$
Output: $R = N - r, r = X^{-1}\ 2^k, m \leq k \leq 2\ m$
  $u \leftarrow N; v \leftarrow X; r \leftarrow 0; s \leftarrow 1; k \leftarrow 0$
  while $v > 0$ do
    if $u_0 == 0$ then
      $u \leftarrow u / 2; s \leftarrow s \times 2$
    else if $v_0 == 0$ then
      $v \leftarrow v / 2; r \leftarrow r \times 2$
    else if $u > v$ then
      $u \leftarrow (u - v)/2 / 2; r \leftarrow r + s; s \leftarrow s \times 2$
    else
      $v \leftarrow (v - u)/2 / 2; s \leftarrow r + s; r \leftarrow r \times 2$
    end if
    $k \leftarrow k + 1$
  end while
  if $r \geqslant N$ then
    $r \leftarrow r - N$
  end if
  return $N - r, k$
```

Algorithm 3 Modular Inversion / Phase II

```
Input: $k = k\ value\ fromAlgorithm\ 2$
       $t = N - r\ value\ from\ Algorithm\ 2$
       $N = Modulus$
Output: $w = X^{-1}\ 2^m \mod N$
  $w \leftarrow t$
  for $i = 1 \rightarrow (2\ m - k)$ do
    $w \leftarrow w/2$
    if $w \geqslant N$ then
      $w \leftarrow w - N$
    end if
  end for
  return $w$
```

2.1.4 Montgomery Form

Efficient modular multiplication and inversion algorithms are based on Montgomery arithmetic. The operands used by the algorithms should be converted into Montgomery form in order to be used by the algorithms. The conversion operations are costly operations, so their usage for only one multiplication or inversion will not be efficient. However, the extra cost of conversion will be negligible when many multiplication operations are executed as in the case of cryptography.

The conversion of the operands can be achieved by multiplying the operands with 2^m. An operand, say T, should be taken the form of $2^m\ T \mod N$, where m is the bit size of N. This conversion can be performed by using only Montgomery Multiplication Algorithm, so that no extra logic or function needed to be implemented. If operands need to be converted, then the Modular Multiplication is used as following for the conversion:

$$Modular\ Multiplication(T, 2^{2m}, N) = 2^m\ T = R \mod N \tag{3}$$

When the operands are in Montgomery form, the arithmetic operations will output the results in Montgomery form as well. Therefore, result of a multiplication operation can be used as an operand in another multiplication or inversion operation. As long as the calculations continue, no conversions are needed to turn them back to their original form. After the calculations the results can be converted back to the original forms by using Montgomery multiplication again:

$$Modular\ Multiplication(R, 1, N) = 2^{-m}\ R = T \mod N \tag{4}$$

Modular addition and subtraction is also compatible with Montgomery multiplication and inversion. The operands from the Montgomery form can be used in the addition and subtraction algorithms and being in a Montgomery form or not will not affect the calculations. For instance, for the operands that are in the Montgomery forms addition and subtraction operations will also output in the Montgomery form: $2^m\ A \pm 2^m\ B = 2^m\ (A \pm B) \mod N$. Therefore, Montgomery multiplication and inversion can be used with modular addition and subtraction in an algorithm without the need of any form conversions between the operations.

2.2 Asymmetrical Cryptography

In Public Key Cryptosystems [PKCs], each communicating parties poses a public key and a private key. The public key of a party is open to the public and it is used by other parties to encrypt messages and send them to the public key owner. The private key is used to decrypt these messages. These public and private keys are different from each other, while they are linked in a mathematical way. This mathematical relation is computationally infeasible to solve by modern day computers for adequate key lengths.

PKCs are widely used to solve modern day security problems for daily computer users. Many PKCs are proposed to solve different security problems, such as anonymity, integrity and authentication. Some of the main PKCs are RSA [32], Diffie-Hellman [3], Elliptic Curve Cryptography [18], El-Gamal [5], Digital Signature Standard [25] and Paillier Cryptosystem [29].

2.3 Elliptic Curve Cryptography

Elliptic Curve Cryptography [ECC] is a public key cryptography system based on the elliptic curve algebra using finite field arithmetic, which is proposed independently by Victor S. Miller [22] and Neal Koblitz [18]. The security of the ECC is based on the discrete logarithm problem for the points of an elliptic curve. ECC systems provide the same level of security against RSA based systems using much smaller bit sizes. This is especially beneficial in small systems that are limited in energy and memory, like RFIDs and embedded systems.

2.3.1 Curve Property

Most of the ECC systems use the Weierstrass equation [35] for their curve structure with different parameters:

$$y^2 = x^3 + a\,x + b \mod p \tag{5}$$

The a and b are two coefficients that are $0 \leq a, b < p$ and p is a prime number. Different coefficients, a and b, define different curves over F_p. Selection of bigger primes for p will increase the number of points defined on the curve and their group order which will eventually increase its security level.

2.3.2 Point Property

In ECC, points that are solutions to the equation (5) form an Abelian group [16] together with elliptic curve group operation and with an extra point called point of infinity, which behave as the identity element of the group and denoted as ∞. The elliptic curve group operations of two group elements will result in an element that is in the same group, because of the closure property. Therefore, it is possible to create points in the same group using elliptic curve operations such as point addition, point doubling and scalar multiplication.

2.3.3 Point Addition

The point addition is the basic operation that takes two points on the curve and outputs a point on the curve. The addition operation can be applied to any two points on the curve which will result in a new point on the curve. The calculation of the addition between two points, say P and Q, can be done as follows:

1. A line is drawn that passes through P and Q is added to the graph.
2. All the intersection points of the curve and the line are determined.
3. If there is a point Z in an intersection which is not P and Q, then result is the projection of point Z with respect to x axis.
4. If there is not a point rather than P and Q, then result of the addition is ∞.

An example of the addition operation is illustrated in Figure 1. There are three intersection points and the addition of these points will be $P + Q + Z = \infty$.

Therefore, $P + Q = -Z$.

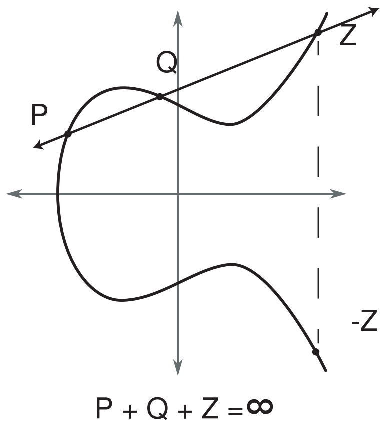

Figure 1: Point Addition

Algebraically, calculating the addition operation of the points P and Q can be performed using the formulae:

1. The slope of the line that passes through P and Q is calculated as, $t = \frac{(y_p - y_q)}{(x_p - x_q)}$.
2. Then, $x_z = t^2 - x_p - x_q$.
3. Then, $y_z = -y_p + t\,(x_p - x_z)$.

2.3.4 Point Doubling

The doubling operation of a point is basically addition of a point by itself. Calculation of point doubling is similar for the geometric approach. As for the algebraic approach the doubling operation for a point, say P, can be done as follows:

1. A tangent line is drawn to the curve that passes through the point P.
2. All the intersection points of the curve and the line are determined.
3. If there is a point Z in an intersection which is not P, then result is the inverse of point Z.
4. If there is not a point rather than P, then result of the addition is ∞.

An example of the doubling operation is illustrated in Figure 2. There are two intersecting points and the addition of these points will be $2\ P + Z = \infty$. Therefore, $2\ P = -Z$.

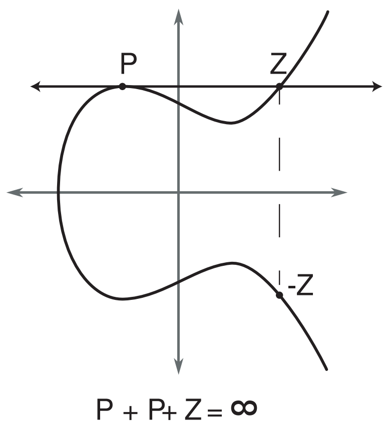

Figure 2: Point Doubling

Algebraically, calculating the doubling operation cannot be done by using the formula for the point addition operation. If the point addition formula is used, while calculating $t = \frac{(y_p - y_q)}{(x_p - x_q)}$, the divisor will result in zero. Thus, for doubling operations the formula is as the following:

1. By taking the differential of the both sides in the equation $y_p^2 = x_p^3 + ax_p + b$, we find $t = \frac{(3\,x_p^2 - a)}{2y_p}$.
2. Then, $x_z = t^2 - 2x_p$.
3. Then, $y_z = y_p + t\,(x_z - x_p)$.

2.3.5 Scalar Multiplication

In ECC, scalar point multiplication is the main cryptographic operation. Scalar multiplication is basically to multiply a point P with a scalar k to calculate a point Q. One way of calculating the scalar multiplication is through addition of P by itself k times. However, this approach is not feasible in a cryptographic case, because scalars are too big in order to satisfy the security needs. Therefore, it is computationally infeasible to calculate a scalar multiplication only using point addition. In order to calculate the scalar multiplication in a feasible time Double-and-Add Algorithm is used:

Algorithm 4 reduces the time complexity of the scalar point multiplication from $\mathrm{O}\big(k\big)$ to $\mathrm{O}\big(log_2\ k\big)$. This time reduction makes scalar multiplication feasible, which makes usage of elliptic curves in cryptography possible.

2.3.6 Order of an Elliptic Curve Group

A point in an elliptic curve group can generate all or a subgroup of the elliptic curve group points by scalar multiplication. The smallest scalar value in a scalar

Algorithm 4 Scalar Multiplication

```
Input: $P = Point\ to\ be\ multiplied$
       $k = Scalar\ value\ to\ multiply\ point$
Output: $Q = kP,\ output\ of\ the\ scalar\ multiplication$

  $Q \leftarrow 0$
  for $i = m \rightarrow 0$ do
    $Q \leftarrow 2\ Q$ (point doubling)
    if $k_i == 1$ then
      $Q \leftarrow Q + P$ (point addition)
    end if
  end for
  return $Q$
```

multiplication that results in point of infinity is the order of that point. The order of a point is cryptographically important, since it gives the number of points that a point can generate in the group. For cryptographic purposes, the order of a point must be large enough.

The order of a point is defined by the prime factors of number of points that satisfies the curve equation. If total number of points satisfies the equation is a prime number then the order of the point is that prime number. However, if the total number of points on a curve is a composite number, then order of a point is one of the prime factors or any multiplicative combination of those prime numbers. Therefore, *oder* of a point should be prime and a large number, so that the system can be cryptographically secure.

2.3.7 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The security of the Elliptic Curve Cryptography depends on the hardness of solving discrete logarithm problem in elliptic curve group. The discrete logarithm problem can be defined on an elliptic curve as follows:

1. Let E be an elliptic curve defined over finite field F_q.
2. Find a point P element of E.
3. Take a scalar number l and compute $Q = l\ P$.
4. Elliptic Curve Discrete Logarithm Problem is finding l when P and Q are given.

The order of the scalar l should be large enough in bit sizes so that it should be infeasible to compute l. Therefore, in cryptographic applications a curve with a large prime order is chosen so that the points' orders are also large primes. Many well known protocols are implemented on elliptic curves using Discrete Logarithm Problem. One of the many algorithms is Elliptic Curve Diffie-Hellman [EDCH] Protocol which will be descried in detail in the section 2.3.8.

2.3.8 Elliptic Curve Diffie-Hellman

EDCH is a protocol to set a shared secret between two parties over an insecure channel using elliptic curves. The Elliptic Curve Diffie-Hellman protocol realized as follows:

1. Alice and Bob uses an elliptic curve E and a point P on E.
2. Alice selects a private scalar k_a, and calculates k_aP and sends it to Bob.
3. Bob selects a private scalar k_b, and calculates k_bP and sends it to Alice.
4. Bob takes k_aP and calculates $k_b(k_aP) = Q_b$.
5. Alice takes k_bP and calculates $k_a(k_bP) = Q_a$.
6. $k_b(k_aP) = k_a(k_bP) \to Q_b = Q_a$, so Alice and Bob shares a secret.
7. Eve only learns k_aP and k_bP by eavesdropping.

Alice and Bob can use the x coordinate of the shared secret as a key or it can be used to create a key for symmetric encryption. Although Eve can eavesdrop on values of E, P, k_aP and k_bP, she cannot calculate k_a and k_b using k_aP, k_bP and P, since doing so necessitates solving ECDL in a feasible amount of time.

2.4 Huff Curves

Huff curves are introduced by Huff [11] in 1948 as a new form of elliptic curves. Every elliptic curve, which contains points that are order of 2 and 4 ($Z/4Z \times Z/2Z$), can be mapped to a Huff Curve. In other words, an elliptic curve, which holds points with the order of 2 and 4 is isomorphic [17] to a Huff Curve. The form of a Huff curve is as follows:

$$ax(y^2 - 1) = by(x^2 - 1) \text{ where } a^2 \neq b^2 \text{ and } a, b \neq 0 \tag{6}$$

Huff curves have both advantageous and disadvantageous properties over elliptic curves. Usage of Huff curves is beneficial since, Joye et al. [13] proposed an efficient unified addition formula that can also be used for doubling operations. The design of a new hardware is not needed for the doubling operations and thus hardware area is smaller. Another advantage of the Huff Curve is that the addition formula does not depend on the curve parameters. Every point addition operation for any curve equation can be calculated through the same addition formula without changing any parameters. Therefore, parameters of the curve equation can be changed easily without the need of setting new parameters in the hardware. Huff curves are disadvantageous since, they need more arithmetic operations to calculate point addition and doubling than elliptic curves do. Like in most cases, the usage of the Huff curve should be determined by the time-memory trade-off.

2.4.1 Huff Curve Point Addition/Doubling

Point addition for points in a Huff curve is calculated as follows:

1. P_1, P_2 are two points that are elements of the curve.
2. Lets, $y = lx + m$ be line passing through points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ that intersects with a third point $-P_3(-x_3, -y_3)$.
3. We can calculate the slope $l = \frac{y_1 - y_2}{x_1 - x_2}$ and $m = y_1 - mx_1$
4. $ax(y^2 - 1) = by(x^2 - 1) \rightarrow ax((lx + m)^2 - 1) = b(lx + m)\ (x^2 - 1)$ by using $y = lx + m$

5. We can obtain:

   (a) $x_3 = x_1 + x_2 + \frac{m(2al-b)}{l(al-b)}$

   (b) $y_3 = lx_3 - m$

6. If we expand and simplify the formula, when $x_1x_2 \neq 1$ and $y_1y_2 \neq \pm 1$:

   (a) $x_3 = \frac{(x_1+x_2)\,(1+y_1y_2)}{(1+x_1x_2)\,(1-y_1y_2)}$

   (b) $y_3 = \frac{(y_1+y_2)\,(1+x_1x_2)}{(1-x_1x_2)\,(1+y_1y_2)}$

Unlike in an elliptic curve the cases where $x_1 = x_2$ or $y_1 = y_2$ will not cause the divider to be zero. Therefore, the addition formula can also be used for point doubling by adding the point to itself.

2.5 Pollard Rho Attack

Pollard Rho Algorithm [30] is basically an integer factorization algorithm invented by John Pollard in 1975. However, its method can also be used in the calculation of discrete logarithm problem for elliptic curves. The algorithm is based on finding a collusion using a random-walk in a cyclic group. The algorithm selects a random starting point and calculates the next point with a random function. Calculations of points create a tail first and as the calculations continue they will form a cycle which will result in a collusion that will help to solve ECDLP. Figure 3 shows progress of the calculation on a random-walk.

The algorithm needs fewer number of point calculations than its order to find collusion because of the birthday paradox. Collusion can be found in $(\pi n/2)^{\frac{1}{2}} + c$

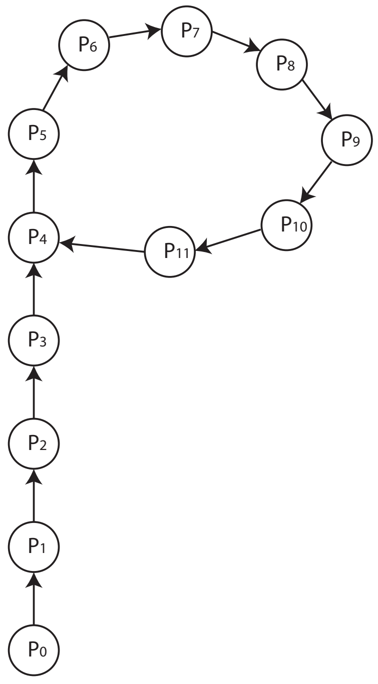

Figure 3: Point Behaviour

calculations on average, where n is the order of the point on the elliptic curve and c is a constant for calculation of the points on the tail.

The original algorithm takes into account only single thread of calculation without parallelism. However, Oorschot and Wiener [27] proposed a parallel version of the algorithm. Even though collusion still can be calculated with same amount of point calculations, the calculations can be divided into multiple processes with a linear decrease in the calculation time with each number of processes. Using the parallel algorithm, collusion can be found in $\frac{(\pi n/2)^{\frac{1}{2}}}{W} + c$ calculations by each process, where W is the number of processes. The Figure 4 depicts the calculation of the points and finding a collusion on a parallel Pollard

Rho attack.

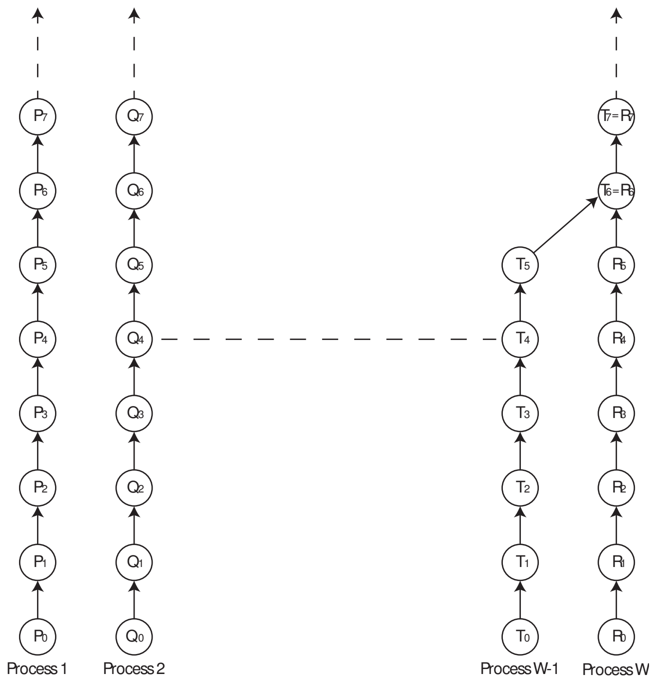

Figure 4: Parallel Point Collusion Calculation

The generic algorithm for multiprocess Pollard Rho attack is shown in Algorithm 5. In the algorithm random function for the random walk can be set to any random function which can be decided according to the specifications or limitations of the platforms. The importance of the selection of the random function and distinguished point property, which is used in the Pollard Rho Algorithm, is described in the sections 2.5.1 and 2.5.2.

Algorithm 5 Pollard Rho Attack Algorithm

```
Inputs: $P \in E(F_p)$
        $l = Scalar\ to\ multiply\ point\ P$
        $n = ord(P)$
        $Q = l\ P$
        $W = Number\ of\ total\ processors$
Outputs: $l,\ which\ is\ the\ discrete\ logarithm\ problem\ l = \log_p Q$

  $Select\ a\ random\ function\ G\ that\ takes\ a\ point$
  $as\ input\ and\ creates\ a\ random\ point$
  for $i = 0 \rightarrow W - 1$ do
    $c_i = rand() \mod n$
    $d_i = rand() \mod n$
  end for
  for $each\ processor$ do
    $Compute\ a\ random\ point\ R_i \leftarrow c_i\ P + d_i\ Q$
    while $No\ Point\ Collusion\ in\ the\ database$ do
      if $R_i\ is\ a\ distinguished\ point$ then
        $Store\ (R_i, c_i, d_i)\ in\ the\ database$
      end if
      $R_i \leftarrow G(R_i); c_i \leftarrow g(c_i); d_i \leftarrow g(d_i)$
    end while
  end for
  if $Any\ Point\ collusion\ found,\ where\ R_j == R_k$ and $c_k \neq c_k$ then
    $l \leftarrow \frac{(c_j - c_k)}{(d_k - d_j)} \mod N$
  else
    $Rerun\ the\ algorithm$
  end if
  return $l$
```

2.5.1 Distinguished Point Property

The Pollard Rho Attack Algorithm is basically based on finding a collusion of two points with different coefficients (i.e. $R_j = R_l \rightarrow c_jP + d_jQ = c_lP + d_lQ$). This is achieved by adding all the calculated points into a database and running a search algorithm to find a collusion. However, this approach is impractical since the total number of points to be calculated is $(\pi n/2)^{\frac{1}{2}} + c$ where n is the order. This necessitates huge storage requirements to perform a successful attack. For instance, in the attack c, d, x, y values are stored and they require a space of 80 bytes for 160 bit field. In practical applications, n should be at least 160 bit for a sufficient level of security, where the attack reduces its security level approximately to 80 bit. This will create a storage requirement of $2^{80} \times 80$ bytes to perform the attack, which is practically impossible. Even in a case where the security level is 30 bit, the attack requires 80 gigabytes of space. In addition to the huge space requirement, communication between the database and the processes and collusion search will also create prohibitively huge overhead.

Overcoming the problem of huge space requirement is to add a distinguished point property, which will decrease the throughput of the point calculation. The addition of a distinguished point function will selectively send the points to the database after the calculation of the points. Generally, the distinguished point property is set to check if the most significant k-bit of a point is $zero$ or not. It is simple and efficient because it does not take much resources and comparison is a fast process. Another benefit of this k bit comparison is that the ratio of the reduction can be determined accordingly. The function searches for a k-bit pattern in general case, therefor the points sent to the database will be reduced by a factor of 2^k. In other words, the process will send points to the database approximately after every 2^k points calculation.

2.5.2 Random Function

A random function is needed to determine the coefficients in generating random number (e_i and d_i in $c_iQ + d_iP$). Pseudorandomness can also be used while calculating points, but true randomness can give better results while finding distinguished points. In a pseudorandom case, the points may stay in a loop without any distinguished points being calculated. However, if the function is truly random then, the chance of such a case is very low. Although usage of a true random function is more beneficial, it is complex to design such a function. A pseudorandom function is easier to implement and it is more practical in terms of calculation. There are different ways to implement pseudorandom functions. One of the simplest and most common implementation is to use the two least significant bits of the point R as selection inputs as shown in Algorithm 6.

Another approach, which also creates more random results, is to form a table by selecting multiple, random c and d coefficients and computing multiple R points. As in the Algorithm 6, depending on the size of the table the least significant bits of R can be used as selection inputs for the table and R, c and d can be updated by using the table. However, this approach has more storage requirements, since a table of coefficients and points are stored. The storage requirement may not be much of a problem in a software implementation, but it may be problematic in small FPGAs. Therefore, the random function should be selected depending on the platform features.

Algorithm 6 Random Function

```
Inputs: $R_{in},\ P,\ Q,\ c_{in},\ d_{in},\ N$
        $R_{in} = Input\ Point,\ where\ R_{in} = R_{in_{k-1}}, R_{in_{k-2}}, \ldots, R_{in_1}, R_{in_0}$
        $P = l \times Q,\ l\ is\ a\ scalar\ for\ the\ discrete\ logarithm\ problem$
        $c_{in},\ d_{in} =\ coefficients\ for\ R_{in}$
Outputs: $R_{out},\ where\ it\ is\ a\ calculated\ random\ point$
        $c_{out},\ d_{out} =\ coefficients\ for\ R_{out}$
  if $R_{in_1} == 0$ and $R_{in_0} == 0$ then
    $R_{out} \leftarrow R_{in} + P;\ c_{out} = c_{in} + 1$
  else if $R_{in_1} == 0$ and $R_{in_0} == 1$ then
    $R_{out} \leftarrow R_{in} + Q;\ d_{out} = d_{in} + 1$
  else
    $R_{out} \leftarrow R_{in} + R_{in};\ c_{out} = c_{in} + c_{in};\ d_{out} = d_{in} + d_{in}$
  end if
  return $R_{out},\ c_{out},\ d_{out}$
```

3 Programmable Logic Devices

Programmable Logic Devices (PLDs) are electronic circuits that are used for hardware programming. Their functionalities are not fixed during their manufacturing, users can program and configure them to implement any functionality later during the usage. There are different types of PLDs and their usage area is different from each other, due to their specifications. Among these different types of PLDs, two of the important PLDs are Field-Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Device (CPLDs).

3.1 FPGA

Field-Programmable Gate Array (FPGA) is an integrated circuit that is designed to be configured after its production by using a Hardware Description Language (HDL). Although FPGAs are also integrated circuits, they are different from application-specific integrated circuits (ASICs) for their configurability property. ASICs operate in much higher clock frequencies in contrast with the FPGAs, but FPGAs have reconfiguration option changing the chip. Having a configuration option also discards the manufacturing process of a chip and creates a rapid development, implementation and production. Another benefit is availability of update for bugs or additional features while the device is in the hand of end-users.

FPGAs contain logic blocks to be used for configuration and these blocks can be used for creation of and, or , xor and other simple logic gates as well as complex logical functions. FPGAs are richer in resource and can be used for configuration of highly complex state machines, which is harder or impossible to implement in other PLDs. In a configuration process of an FPGA, Look-Up Tables (LUTs) of logic blocks are programmed. The LUTs are volatile. Therefore, in every power loss reconfiguration of the FPGA is required. In most self-systems it is achieved by dumping the configuration files in to Flash Memories from which auto-configuration is performed on power-up.

3.1.1 Microblaze (Softcore)

Microblaze is a 32 bit soft-core processor that is designed by Xilinx for FPGAs. The processor generally consists of a Local Memory Bus (LMB) for Block-RAM

(BRAM) access and Processor Local Bus (PLB) for communication with the peripherals connected to Microblaze. Since it is a soft-core processor, it has the flexibility of adjusting its specifications according to applications. For instance, a cache can be added with a user defined size, its pipeline depth can be set as 3 or 5 stages by taking into account area-time ratio, memory management modules can be added to the buses to communicate with external storage devices (i.e. external RAMs, flash drives), barrel shifter and floating point arithmetic unit can also be added to improve time performance or any user hardware module can be added through PLB for specific applications. These user hardware modules can be controlled by software that runs on Microblaze. The availability of this option makes Microblaze a practical instrument to be used for hardware-software co-development.

Microblaze's architecture is suitable to be used with two types of RAMs. The first one is the external RAMs, which are larger in capacity and have higher access times. Microblaze can access the RAM using a memory controller module and any code can be dumped inside of the RAM, from which it can be executed. The second one is internal RAM, which is actually BRAM inside the FPGA. The FPGA uses its resources and creates an interface to use the BRAMs of the FPGA in Microblaze as an internal RAM. It still functions as a RAM with having the advantage of low access time. However, the BRAMs on the FPGAs are too limited and the Microblaze's internal RAM option generally limits to few KBytes. Hence, internal RAMs cannot be an option to execute complex and high resource-oriented codes Therefore, external RAMs are required in such cases.

Xilinx have a Software Development Kit (SDK) that enables the development of software applications for Microblaze. SDK offers a C/C++ programming platform with built-in libraries for usage of the general purpose functions and control of the Xilinx hardware modules that can be embedded into Microblaze. SDK uses GNU toolchain for development of the software. Since software designs are more flexible in contrast to hardware designs, it is easier to test, control and operate hardware modules by a software interface. The platform shows two options for software developments for different usages. The first one is a standalone software development for simple software applications, which is advantageous for limited memories. Another option that SDK provides is a small microlinux environment for software development. It can provide more complicated software development options such as threads, sockets and many other options by using a microlinux kernel. However, the microlinux kernel creates an overhead on the codes, therefore it needs more memory space to function, where in most designs this high memory need is provided by external memory resources.

User hardware modules are generally implemented to improve performance when the software implementations provide low level of performance. Although hardware implementations have a higher performance, usage of these hardware modules are harder without a proper interface. Microblaze can provide the interface and can ease the control of the hardware module. The communication between the processor and hardware module is performed by a software program using the PLB. Microblaze provides the software-hardware interface in an effective way. For instance, a hardware RSA engine might be connected to the Microblaze and the inputs might be supplied to the engine by Microblaze. Microblaze basically executes the software and gets the input values from the software and writes them to the internal registers of the hardware module, so that hardware will use those values as inputs. After hardware completes its process, it puts the outputs in the registers and Microblaze takes the results from those registers. Therefore, Microblaze can provide an easy interface to control the hardware to give inputs and receive outputs in order to use the hardwares' functionality.

3.1.2 Executable and Linkable Format for Microblaze

As mentioned in section 3.1.1 Microblaze can run software programs and it is also capable of running a small kernel such as microlinux. It uses Executable and Linkable Format (ELF) for its executable files. ELF format is suitable for Microblaze, since it is flexible and extensible as Microblaze. Any change or additions in the Microblaze can be adapted also to the software level with additions to the ELF specifications. An ELF have a file structure that consist of sections. The basic sections are text, data, bss and rodata. The features of those basic sections are as follows:

Text Section: It is the section where instructions of the executable code is placed. It is used to read the instructions and perform the operations accordingly. It covers a fixed size in the memory and it is a read-only section in most cases.

Data Section: It holds the initialized static and global variables.It uses a fixed size in the memory. Since initialized data can be changed during the execution of the program, both read and write operations can be performed in the section.

BSS Section: This section has a similar usage as data section. It stores the global and static variables which are either initialized to zero or uninitialized.

Rodata Section: This section includes variables that are declared as constants in the program. The rodata section is also a read-only memory with a fixed size.

Besides the basic sections, an ELF file contains other sections to handle dynamic memory operations in a program. These memory sections are called stack and heap. Heap is used to handle dynamic memory that are dynamically allocated during the execution of a program. Stack, on the other hand, is used to store registers, return addresses and in many other situations where a Last-In-First-Out (LIFO) data structure is needed. The heap and stack are consecutive sections in the memory. Heap stores its data starting from higher address memory location and stack stores its data starting from lower address memory location. Since hap and stack are adjoined sections and they store the variables in the opposite direction, an overflow may cause a distortion on the other section. Unlike basic segments in an ELF file, heap and stack sizes are not fixed since they are used to handle dynamic memory operations. Required size of a dynamic memory will change form application to application and their size values should be set accordingly. In Microblaze, using the SDK heap and stack sizes can be set to any value as long as there is available area in the memory.

As an ELF file not always consists of basic segments, many other segments can be added through the linker. These segments can be used to store specific data or any part of the code segments. This flexibility is beneficial in Microblaze, since important and frequently used functions can be stored in a specific segment and those segments can be mapped to BRAMs. As explained in section 3.1.1 BRAMs are too to hold complex programs, but they feature much faster access times compared to external RAMs. When most frequently used functions are executed from BRAM, the time performance of the software program will be increased.

Microblaze uses virtual addresses for memory and register mapping. It assigns a default starting address to BRAMs, external RAMs and flash memories. Those virtual addresses are fixed values and they are used during the compilation of software programs. Therefore, the segments in the ELF file should be mapped to the right addresses. Otherwise, code execution will fail since the pointers will use fixed addresses in memory operations. For instance, if the segments of the ELF file is linked in the external RAM, where virtual address is 0x1C000000, the code should be directly imported into the RAM starting from the address 0x1C000000.

3.2 CPLD

Complex Programmable Logic Device (CPLD) is a PLD that is similar to FPGAs with some differences. They have moderate amount of sea-of-gates to be configured. Unlike FPGAs, they cannot be configured to implement highly complex designs. Although they are much less powerful than FPGAs, they have non-volatile configuration memory. It is beneficial since reconfiguration is not needed in a power-loss. When power is on, it will automatically start its functionality; for example a state machine will start from its initial state and perform its functionality. Reconfiguration of electronic devices is not possible every time they are turned off for end-users. Therefore, CPLDs play important role for in bootstrapping electronic devices.

3.3 Related Works

Huerta et al. [10] describes a parallel computation scheme using Microblazes. They propose three different network topologies for message transmission between multiple Microblazes that are configured in a single FPGA. Their work measures the communication overhead between the processors to calculate the speedup and efficiency of the systems. Furthermore, the work contains an experimental implementation of AES encryption/decryption algorithm to measure the achieved increase in throughput by multi-core implementations. In their scheme, the speedup of the system depends on the software parallelism using multiple Microblazes and does not achieve any hardware parallelism. In other words a Microblaze based Multiprocessor architecture is used for speedup. Although, parallelism is achieved in the software by multi-core, the frequency of the Microblazes are lower compared to modern day CPUs, so the Microblaze based Multiprocessor may result in a lower performance compared to modern day CPUs. Another drawback of the system is that it does not include the usage of multiple FPGAs and their communication interface, so the design is only limited to a single FPGA and the number of Microblazes that can be fit into the single FPGA.

Saldana et al. [1] proposes a TMD-MPI implementation for multiple embedded processors across multiple FPGAs. This work also takes into account the software parallelism using multiple Microblazes by a middle-ware application called TMD-MPI. One of the advantages of their work is that TMD-MPI interface can be used to control multiple FPGAs, so their work does not depend on a single FPGA parallelism. The design still lacks of hardware parallelism to benefit which is implemented in an extended version of their work in [33]. In the extended work, they implemented the TMD-MPI interface to the x86 Intel processors and to the special purpose hardware modules. Their design enables the usage of x86 Intel processors, Microblazes, Power-PC and special purpose hardware engines by a simple software interface. The special purpose hardware modules are controlled by an interface called TMD-MPE which is a hardware implementation. The TMD-MPE is an interface between the spacial purpose hardware and the internal network inside a single FPGA. The internal network of an FPGA is controlled with an on-chip network interface for package distribution. Whenever a package is addressed to the special purpose hardware, the internal network passes the packages to the TMD-MPE. The package contains data and message parameters and MPE performs the desired function using the message parameters. Since TMD-MPI contains large number of functions, TMD-MPE contains a small number of them and it might not be able to perform every functionality that TMD-MPI needs. The external network communication is sustained by an Off-Chip Communication Controller (OCCC) engine. Basically, every FPGA sends and receives the packages via OCCC and forwards them to the on-chip network interface to be distributed. Their scheme does not contain the dynamic configuration option of the special purpose hardware modules. Therefore, the hardware designs should be configured into FPGAs beforehand, in order to be used by the TMD-MPI interface. Another drawback is that the design needs OCCC for every FPGA and a TMD-MPE for every special purpose hardware modules which may be inefficient since, they may cover a large area in the FPGA. Also, FPGAs are connected to a motherboard by Front-Side-Bus using an OCCC. If the total number of FPGAs exceeds a volume than the motherboard can hold, another Host PC should be added to the system which increases the total cost of the system.

Güneysu et al. [7] proposes an FPGA cluster based special purpose hardware to solve discrete logarithm problem for elliptic curves. Their work focus on cryptanalytic attacks by using special purpose architectures. The design is built to attack and measure the security levels of the elliptic curves. The attack is performed by using a special purpose hardware for elliptic curve point addition and a Host PC to manage the control of the hardware designs. Unlike the previous paper in [33], the design does not aim to build a generic architecture for hardware control. However, the work is similar in sense of having a Host PC as the controller and the FPGAs as the work horses. The Host PC is used to send/receive data and to form a database to perform the ECDLP attack. The possibility of hardware acceleration is shown with the work, but it does not contain any protocol for dynamic configuration option for the FPGAs and it does not contain a generic architecture to control multiple FPGAs for massive computations.

In [6] and [8] Güneysu et al. present cryptographic and cryptanalytic works on a special machine called COPACOBANA. COPACOBANA is a machine that is consists of 120 Spartan-3e1000 FPGAs that are connected to a Host-PC by using an FPGA interface. The Host-PC uses software libraries to control and configure the FPGAs which ease the control of the machine. The machine is specifically designed for high performance computing in any field of science where computation power is essential. The papers basically focus on efficient hardware architectures and use the parallelism of the FPGAs to measure the security levels of widely used security algorithms like DES, 3DES, AES, Elliptic Curve Digital Signature, etc. The COPACOBANA machine is moderate in price-performance ratio when compared to computers which have same level of computation power. Although it has a moderate price-performance ratio, it is still an expensive device to obtain.

In this thesis, we try to eliminate the disadvantages of the proposed systems and create an FPGA-cluster with an ease of control mechanism. In Huerta et al. and Saldana et al. the proposed systems either use only Microblazes for parallelism or uses communication interfaces for the hardware engines. We try to eliminate the software parallesim, since hardware parallelism is better, and the interfaces for the communications, because they will utilize area in the FPGAs. We try to take advantage of the software control like TMD-MPI architecture in Saldana et al. with additions of having reconfigurability options on the FPGAs. Güneysu et al. tries to measure the security levels of the widely sued algorithms with a high priced device, but it is not always possible to purchase such devices. Our work aims to use the low level FPGAs to form clusters and estimate the security levels of the algorithms based on the speed of the performed attacks.

4 General Overview of the Proposed Scheme

In this section, we give details on the proposed architecture for FPGA cluster, our scheme and its operational steps, usage of FPGA cluster for acceleration of cryptographic computations and cryptanalysis.

4.1 Proposed Architecture for FPGA cluster

The architectural overview of the FPGA cluster we use in our work is depicted in Figure 5. Since our architecture uses TCP/IP for communication, any FPGA board connected to the Internet can be a part of our cluster and individually accessed from anywhere in the network.

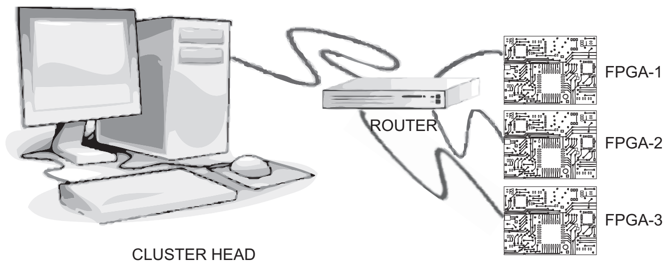

Figure 5: General overview of the FPGA cluster

Our aim is to create a platform consisting of various hardware and software components that can be used for time- and resource-consuming tasks with a specific emphasis on cryptologic applications. The application we target is of the type where direct interprocess communication is not necessary such as exhaustive key search.

As pointed out earlier, our goal is to harness especially the computation power of inexpensive FPGA boards for the use in cryptologic applications. Therefore, we use Spartan-3E Starter Kit board, which is one of the basic equipment used in logic design courses. A Spartan-3E Starter Kit [38] is a board that consists of the following hardware components: i) volatile programmable unit (FPGA - XC3S500E), ii) a nonvolatile programmable unit (CPLD - XC2C64A), iii) 128 Mbit parallel flash memory, iv) 64 MB DDR SDRAM (MT46V32M16), v) Standard Microsystems LAN83C185 10/100 Ethernet physical layer (PHY) interface and vi) a RJ-45 connector. In what follows, we give details of these components and how we utilize them in our cluster.

FPGA: Spartan-3E Starter Kit features an FPGA chip [36] (XC3S500E) of 500K equivalent gates.

For flexibility, we adopt a configuration scheme so that a user not only sends computation tasks to FPGA device, but also configures it remotely via Internet. By configuration, we mean sending small software applications that will run on the softcore as well as configuring hardware resources for the actual computation. These software applications function as proxies, which handle the communication with the cluster head to receive jobs and deal with other management tasks. The softcore (MicroBlaze) is a 32-bit RISC processor which can be implemented on an FPGA device using the general-purpose configurable logic.

CPLD: Unlike FPGA, CPLD configuration is not lost when the power is switched off. Thus, it can be used to implement a state machine that will bootstrap the device and perform simple configuration steps such as loading the softcore to the FPGA during the startup. The CPLD plays an essential role in configuration switch operation as well.

Storage Units - SDRAM and Parallel Flash: Two types of storage devices on the Spartan-3E Starter Kit play an important role in our application. The first one is a 512 Mbit DDR RAM (MT46V32M16), which is a volatile memory (hereafter referred as SDRAM) used mainly for running our proxy codes. The second one is a 128 Mbit (16 MB) Intel Strata Flash parallel NOR Flash PROM (JS28F128-J3D75), which is used for storing configuration files of the softcore and hardware for the computation, proxy code, and finally the computation results in some cases. We will refer this non-volatile memory as the parallel flash. The utilized components on the FPGA board are shown in Figure 6.

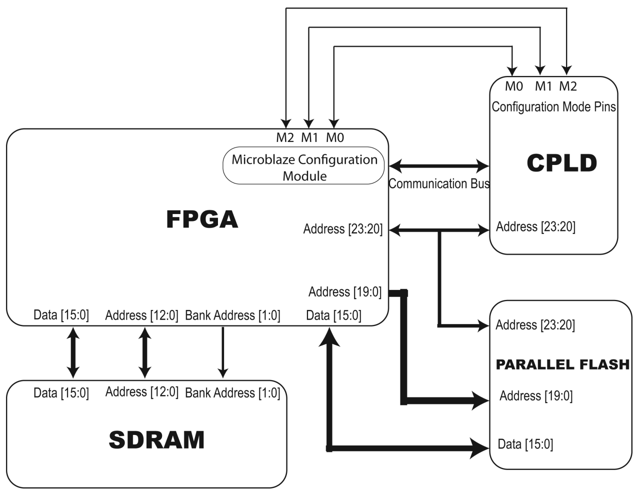

Figure 6: Components of the FPGA board

For flexibity and transparency purposes, the proposed infrastructure is designed as a self-configuring system, which becomes ready for remote configuration once it is connected to the Internet. In addition, *multi-configuration* technology allows to switch between configurations dynamically.

The CPLD controls the most significant four bits of the address lines of the parallel flash as shown in Figure 6 allowing it to store up to 16 different configuration files in the parallel flash. The multi-configuration technology is available in many FPGA devices.

4.2 Our Scheme and Its Operational Steps

Our scheme can be understood better if the following four key steps of its operation are explained in detail as follows:

Softcore Configuration: The FPGA device is expected to start automatically and become ready for remote configuration through the network when the device is turned on. For this, the configuration bit stream of the softcore is stored at address (0x000000)[1] of the parallel flash and the CPLD is configured with a state machine which will help to configure FPGA automatically. The memory map of the parallel flash illustrated in Figure 7 shows the exact location of the configuration file for the softcore.

Following the softcore configuration, a special program called *boot-loader* is executed by the softcore.

Execution of Boot-loader: The boot-loader is a small piece of code, which comes as a part of configuration file of the softcore and stored in internal Block RAM (BRAM) of the FPGA. It is responsible of moving the proxy code (cf.

[1]We use 24-bit address for the parallel flash.

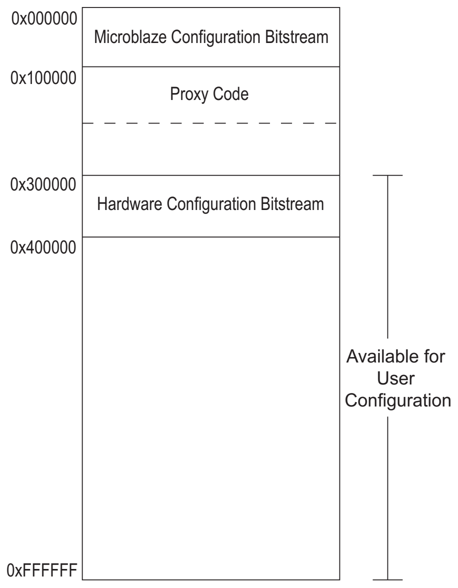
0x000000
Microblaze Configuration Bitstream
0x100000
Proxy Code
0x300000
Hardware Configuration Bitstream
0x400000
Available for User Configuration
0xFFFFFF

Figure 7: Memory map of the parallel flash and placement of configuration and data

0x010000 in Figure 7) from the parallel flash to the SDRAM since the latter is too large to fit in the internal BRAM.

Execution of Proxy for Implementing Client/Server Communication Model: To assign tasks, the cluster head communicates with FPGA boards using reliable TCP/IP protocols which are implemented by the proxy code on the softcore side. In our communication model, the cluster head and the softcore plays the roles of server and clients interchangeably.

Automatic, Remote Configuration of FPGA Device and Configuration Switch: The actual computations for specific tasks are performed by hardware implementations, optimized for FPGA devices. Once the configuration file for a hardware implementation is available, the cluster head can send it through network to the FPGA device. The proxy code running on the softcore is responsible of receiving the hardware configuration file and storing it in the parallel flash (cf. address of 0x300000 of the parallel flash in Figure 7).

The cluster head can send commands to the softcore to perform `write`, `read`, `erase` and `configure` operations. A `write` command can be used to upload a configuration file for a specific hardware to the parallel flash. Then, the `configure` command is used to configure the FPGA with this hardware. Before configuration of the hardware, the softcore can be removed from the FPGA to use its space for the former. Naturally, when the computation by the hardware is finished, the FPGA device should be re-configured back to the softcore. However, the hardware will need to relay the results of the computation to the softcore before its re-configuration. The results are written in a specific location in the parallel flash, whose address is sent to the softcore with the `configure` command.

More precisely, configuration switch necessitates the execution of the the following steps in this order: **i) Communication 1:** the softcore receives input data and stores it in the parallel flash, **ii) Configuration switch 1:** the softcore is removed and the hardware is loaded in the FPGA, **iii) Computation:** The hardware works on the task, and writes the results to the parallel flash **iv) Configuration switch 2:** The hardware is removed and the softcore is loaded in the FPGA, v) **v) Communication 2:** the softcore reads the results from the parallel flash, and sends them to the cluster head.

Alternatively, especially for applications and FPGAs where the area overhead of the softcore is not important, the hardware design and the softcore can run simultaneously in FPGA, which eliminates the need for configuration switch as described above. In subsequent sections, we will demonstrate applications that benefit from the configuration switch.

4.3 Using FPGA Cluster for Acceleration of Cryptographic Computations

A simple and inexpensive FPGA device such as Spartan-3 running at a low clock frequency of 119 MHz can perform an RSA exponentiation operation in about 8 ms using 1553 slices and 10 hardwired multipliers [26][2]. Similarly, the same FPGA device can achieve an encryption rate of 429 Mbps for the AES standard block cipher algorithm using only 103 slices at 161 MHz [9]. Since the FPGA device can realize more than one block of AES encryption engine, it is possible to reach much higher throughput values for encryption operation either using multi-message encryption techniques or a suitable working mode (e.g. counter mode). Therefore, using simple, inexpensive FPGA clusters can be cost-effective alternatives for accelerating cryptographic operations.

When the configuration switch is not used, cryptographic operations can be overlapped with the communication. In other words, as the softcore receives the new requests for cryptographic operation the hardware can perform the calcula-

[2]Note that RSA timings for one signature operation vary between 0.15 ms and 8 ms on a PC depending on the processor (cf. http://bench.cr.yp.to/results-sign.html). In order to obtain acceleration over common PCs, a larger FPGA device that can accommodate more than one instance of crypto unit should be used. Otherwise, many FPGA boards will be needed to outperform PC implementations.

tions for old requests simultaneously. The biggest problem in using inexpensive FPGA boards as cryptographic accelerators is that the softcore consumes a significant portion of FPGA resources leaving only a fraction of them for cryptographic computations. For example, while we can accommodate several AES encryption/decryption blocks [9] on a Spartan-3E along with the softcore, an RSA circuit may not co-exist with the softcore in our FPGA device. Therefore, in RSA case we may need to switch between RSA circuit and the softcore configurations.

During the configuration switch, the SDRAM loses its content since it is essentially a dynamic memory and needs refreshment operations in regular periods due to the fact that data stored as charges decays in time. Therefore, the only way the softcore and the hardware can communicate is through writing to the parallel flash memory. Writing data to the parallel flash, on the other hand, is a very slow operation compared to the execution time of the cryptographic operation for the same data. Consequently, cryptographic acceleration through a simple FPGA board may not be feasible if configuration switching is needed as in the case of RSA circuit. In Section 5, we provide a scenario where cryptographic acceleration may be possible even in the case of configuration switch. But, block cipher acceleration is always possible since the FPGA device can be shared between the softcore and the hardware.

4.4 Using FPGA Cluster for Cryptanalysis

Depending on the cryptographic algorithm, different cryptanalytic attack methods and algorithms are proposed. For recent block cipher algorithms, perhaps the most successful attack method is exhaustive search, which relies on trying out all keys. Exhaustive search or more precisely the effectiveness of exhaustive search made possible through our computational ability also provides a reliable metric for security of many cryptographic algorithms. We tend to quantify the difficulty of breaking cryptographic algorithms in terms of the difficulty of exhaustive search (more precisely the number of basic operations we perform in an exhaustive search) [21, 24, 4]. This is due to the fact that we perform an eventual exhaustive search within a reduced set of secret key candidates in many cryptanalytic attack methods.

Most cryptanalytic algorithms can be adjusted to alleviate the time overhead incurred in inter-process communication between the cluster head and the FPGA boards. Both designs in [7] for exhaustive key search and [8] for solving discrete logarithm problem (DLP) rely on a massively parallel computer of inexpensive FPGA devices as the computational work horses. Exhaustive key search is one of the easiest cryptanalytic problems to parallelize since each FPGA device can run in isolation for a very long period of time that is adjustable with the size of the search interval. Also, as stated in [8], certain computations in the Pollard's Rho method [31, 28] for solving elliptic curve DLP (ECDLP) can be so adjusted to meet any bandwidth restriction between the cluster head and the FPGA boards.

For instance, in an exhaustive search for an AES key using the implementation in [9], one AES block (103 slices) can try approximately 3.3 million key candidates in one second. In a single computation task submitted to an FPGA, which takes about one minute, one additional AES block implemented on the FPGA resources gained by removing the softcore can try out an extra 200 million key candidates. This value commensurates with the number of AES instances that can fit in the space saved through removing the softcore. Since the communication between the cluster head and FPGA device is not intense (in fact only the key interval is needed to be communicated to the FPGA), overlapping communication and computation would not help. Therefore, in such cases it is always beneficial to apply the configuration switch.

5 Implementation and Experimental Results

In this section, we provide some implementation details and experimental results to evaluate the true potential of the proposed FPGA cluster for cryptologic computations. We start by giving the required resources to implement the softcore in Spartan-3E (XC3S500E) devices, which consumes 4,270 out of 9,312 (45%) 4-input LUTs and occupies 3,526 out of 4,656 (75%) slices. The utilization percentage for such a small FPGA device is relatively high and leaving limited configurable FPGA resources for the hardware unit that will perform the actual computation. While the remaining resources are significant in implementing many instances of block cipher algorithms, the public key algorithms may not be implemented in a low end FPGA device. This is, in fact, one of the primary motivations for the scheme that will allow an efficient configuration switch between the softcore and the hardware unit.

In our experiments, we used a Linux-based PC (cluster head) and ASUS RT-N13U router in addition to multiple Spartan-3E boards. We used Verilog for all hardware designs and C/C++ language for software components on the softcore and the cluster head.

5.1 Software Implementations

In this section, we give detailed information on the software implementations for both the softcore and cluster head.

5.1.1 Bootloader

As mentioned in Section 3.1.1 softcore can run both standalone and μlinux applications that are written in C/C++ language. However, it is not always possible to fit the entire code into BRAM of the softcore, especially μlinux applications, since the BRAM sizes of the softcores are generally too small. Therefore, codes, which requires huge amount of storage, are executed from SDRAM. To this end, a bootloader application is used. The bootloader can be embedded into the BRAM of the softcore and the configuration file of the softcore can be updated with the information in the BRAM. When the softcore is configured using the configuration file, the bootloader application is also imported to the FPGA. Once the configuration is completed, the softcore starts its execution from the BRAM, and the bootloader application starts. The job of the bootloader is to copy the proxy code to SDRAM and start its execution. For the bootloader to perform the operations, the proxy code is put into a file format (referred to as *proxy-file-format* henceforth) which is described in detail in section 5.1.3. The bootloader uses the following steps to start the execution of the proxy code:

1. The bootloader reads the header of the *proxy-file-format* from a specific offset[3] from the flash driver. This header contains necessary information to copy and run the proxy code. The structure of the header is given in table 1.

2. The bootloader copies the proxy code from the flash to the SDRAM using the size of the proxy code to allocate sufficient storage for the proxy.

[3]In our case, offset of the flash is 0x100000

Elements of the Structure	**Vector Memory Addresses in the Softcore**	**Size(In Bytes)**
Size of the Proxy Code	-	4
Reset Vector	0x00000000-0x00000004	4
User Exception Vector	0x00000008-0x0000000C	4
Interrupt Vector	0x00000010-0x00000014	4
Break Point Vector	0x00000018-0x0000001C	4
Hardware Exception Vector	0x00000020-0x00000024	4

3. The bootloader sets the reset, exception and interrupt vector values of the proxy code in the softcore to start its execution. These vectors are important to start the execution flow of the proxy code correctly. The values of the vectors differ in each application and they are also different in the bootloader and the proxy code, since the bootloader uses the virtual addresses of the BRAM and the proxy code uses the virtual addresses of the SDRAM. If the values are not set correctly before launching the proxy application, the execution flow of the proxy code will be damaged in the first exception or interrupt event. The features of these vector values is as in the following: **Reset Vector:** It holds the first instruction to start the execution. When the softcore is started or a reset operation is performed, the instructions starts fetching from the reset vector whose address is 0x00000000. **User-Hardware Exception Vectors:** These vectors are used to handle software exceptions and hardware exceptions. The causes of the software exceptions is to the running program on the softcore. The hardware exceptions are generally caused by situations like illegal instructions, instructions and data bus errors and many other hardware related errors. These exception vectors are used to handle exception problems by storing the memory address of the problem, jumping to necessary functions to handle the exception and per-

forming further operations to continue the execution flow correctly. **Interrupt Vectors:** The interrupt vector is used to handle the interrupts occurred by the hardware peripherals. In an interrupt operation, the return address of the execution is stored in the registers and jumps to the interrupt vector to execute the handling process. Later, the address is restored from the registers to continue execution flow.

4. The bootloader resets the internal registers of the softcore to prevent any disturbance that may occur due to the stored values in the registers.

5. The bootloader resets the Program Counter (PC) of the softcore to start the execution flow from the beginning. Because the bootloader resets to the value of the reset vector, the processor directly jumps to execute the proxy code from the SDRAM.

5.1.2 Cluster-Head

In our work, we want to form an FPGA cluster in which FPGAs can be used in their full potentials with an easy control mechanism. In order to create an easy control mechanism, developing a software library was crucial to handle the communication and manage the FPGAs. Since software programs are easier to develop and more flexible relative to hardware designs, we developed a C/C++ software library for the control mechanism. In the development of the software library, we aimed to create middleware services between the cluster-head and the FPGAs in order to use the cluster-head as the data manager and controller, while FPGAs are used as workhorses. The software library is designed to work on

Linux operating system and it uses TCP/IP socket to handle the communication and Posix-Thread library to handle thread applications. In the following, details of the software library structure will be explained and the protocols between the cluster-head and FPGAs will be given:

Software Library Structure

The software library is formed in a hierarchy of structures to store information on FPGAs and manage the communication. In the low level, the library holds a structure called *FPGA_Structure* that stores information about a single FPGA. The *FPGA_Structure* contains information on network, socket, thread, data and flags for an FPGA. The detailed information on the structures is below:

- **Network Structure:** The network structure holds the information about the IP address of the FPGA and input-output port numbers, which are used in the communication process.
- **Socket Structure:** The cluster-head will run both a server and a client application. The structure holds the socket numbers for both applications.
- **Thread Structure:** The threads are used to parallelize the communications and handle server and client applications simultaneously. For that purpose, applications requires the usage of thread indexes, locks and signals which will be described in detail in $Software\ Library\ Protocols$.
- **Data Structure:** The data structure holds information about the offset and size of the data which will be sent to the FPGA or received from the FPGA. It also contains pointers to sending/receiving data.

- **Flag Structure:** The flags are set to handle usage of server and client applications without a conflict in the cluster-head. The flags contain the following information on the FPGA: whether the FPGA is server or client, connected or not and busy or not.

While the library has a structure *FPGA_Structure* in the low level implementation to store information for an FPGA, a class *FPGAS_Class* is implemented, using the *FPGA_Structure*, in the upper level to create functions to control the FPGAs. This class is used to create applications for distributed computing in the FPGAs which is serving as a middleware application. The *FPGAS_Class* holds the *FPGA_Structure* as a linked list and any number of FPGAs can be added to the list at any time in the application. Any *FPGA_Structure* in the list can be selected by an index to access its information. Usage of indexes helps to set information for every FPGA separately which is useful while assigning different tasks for each FPGA.

Software Library Protocols

As the first step, a server application need to be started in the cluster-head to handle the data exchange in a configuration switch operation. Before any connections are performed by the cluster-head, the IP and port numbers of the FPGAs should be entered. After that, any connection request can be sent to the FPGAs as a client. After a connection is established, any read, write, erase or configuration operations can be requested from the FPGAs. The details of the operations are given in the following:

- **Erase Operation:** In an erase operation, the informations sent to the FPGAs are erase command, offset and size of the flash memory need to be erased. When the job request is completed, the FPGA sends an acknowledgement to the cluster-head, so the cluster-head continues to execute its application.

- **Read Operation:** A read operation may be a time consuming operation in some cases. Therefore, rather than communicating with a single FPGA at a time, a thread application is created in order to receive data from multiple FPGAs in parallel. Before starting the read operation, the size and the offset values of the memory area which will be read should be set in the *FPGA_Structure* structure. Then, the necessary memory should be allocated, using the receive pointer in the *FPGA_Structure* to store the receiving data from the FPGA. When the read operation is executed for a specific FPGA, a thread application is started. This application sends the read command, offset and size information to the FPGA. Then, the cluster-head receives the data in chunks of 0x400 Bytes and to synchronize the communication, for every received chunk an acknowledgement is sent to the FPGA. This process continues until all the data is received from the FPGA and after that the data can be accessed from the allocated memory. The visualization of protocol of the read operation is illustrated in Figure 8. The usage of threads creates an asynchronous communication between the FPGA and cluster-head. In other words, while a read operation is occurring for an FPGA, other operations may also continue on other FPGAs. Although asynchronous communication is useful to fasten the processes and to run the FPGAs in parallel, a problem may occur if the read operation is not finished while the data is required for an operation in the cluster-head. The problem is handled using signals in the thread which will be described in detail in $Waiting\ Operation$.

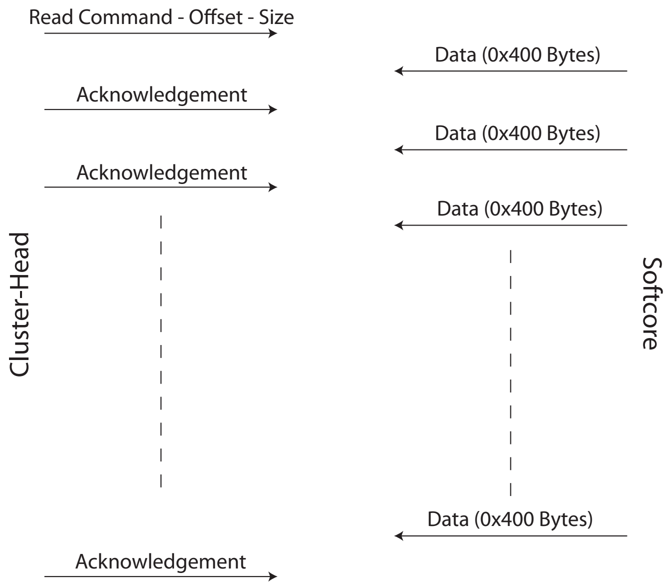

Figure 8: The Protocol of Read Operation

- **Write Operation:** The write operation is similar to the read operation. Before starting the operation, size and offset values of the memory area which will be written should be set in the *FPGA_Structure*. Later, the necessary memory should be allocated, using the send pointer in the *FPGA_Structure*, to store the sending data. After the allocation, the data should be copied into the allocated memory so that a writing operation can be performed. Similar to a read operation, a write operation opens a thread application and the data can be sent in parallel while the cluster-head can continue to perform other operations. In the protocol, the cluster-head sends the data in chunks of

0x400 Bytes and for every chunk, it receives an acknowledgement from the FPGA until all the data is sent to the FPGA. The visualization of the write operation protocol is illustrated in Figure 9. Like in the read operation, the write operation should be handled with a thread-safe method not to cause a race condition between the threads. The problem is handled using signals in the threads which will be described in detail in $Waiting\ Operation$.

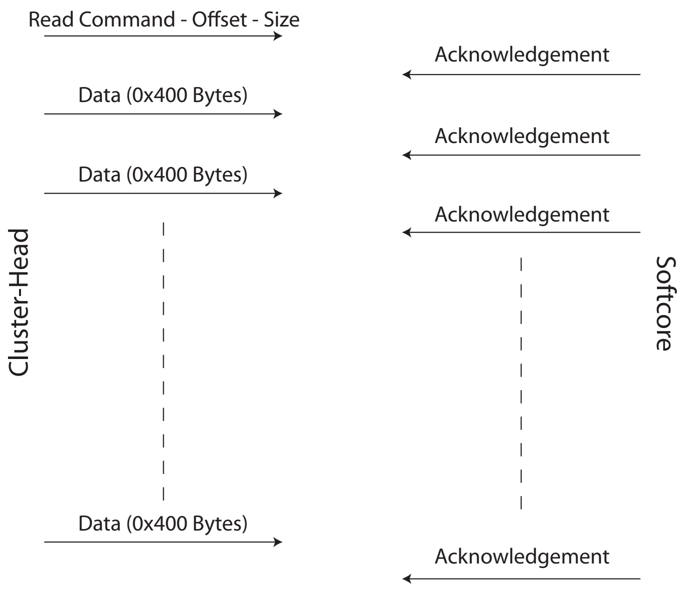

Figure 9: The Protocol of Write Operation

- **Configuration Switch Operation:** As described in Section 4.2, in a configuration switch event, the result of the hardware process, will be sent back to the cluster-head. In order to perform the operation, the server application on the cluster-head should be started in the first place. Then the, offset and size values of the resulting data and configuration command are sent to the FPGA. Later, the operation continues by closing the connection with the FPGA. After the connection is closed, the cluster-head continues to run its other operations, while its server application waits to receive the results. When the FPGA is connected to the server of the cluster-head, which means that hardware operation finished its job and ready to send the results, the cluster-head searches the IP address of the connected device from the linked list and opens a receiving application thread to receive the results. Since it is a thread application, the main part of the application may need to await the results to use them afterwards. This situation is handled like the write and read operation which will be described in detail in $Waiting\ Operation$.

- **Waiting Operation:** Since the library aims to control the FPGA clusters in parallel to perform read, write and configuration switch operations, it needs a thread handling function to prevent race conditions and create thread-safe applications using the library. Therefore, we added a $WAIT$ function to the library to create thread-safe applications. The function is used for waiting an operation that should complete its execution before the other operations can continue. It checks the client/server, busy and connected flags in order to decide whether the FPGA is doing a configuration switch or read/write operation. Once the decision is made, the function waits for the selected operation to complete its process. After the operation is completed, the function finishes its waiting condition and the application continues to execute.

5.1.3 Proxy Code

The softcore contains a large complex software application (the proxy code) to perform the tasks that are sent from the cluster-head. The proxy code is put into a file format for the bootloader to start it. In our setup, all proxy code instructions are executed from the SDRAM and its virtual address is used in the process. Converting the program to a file format is realized with the following steps:

1. The proxy code is compiled and an ELF file is formed.
2. The ELF file is converted into a binary file(referred as *input-binary* henceforth), where the sections of the ELF file are mapped into their virtual address.
3. From the *input-binary* file, size of the proxy code, reset, user exception, interrupt, break point and hardware exception vectors are copied into an output binary file (referred as *output-binary* henceforth).
4. From the *input-binary* file, all the instructions are copied to the *output-binary* file and the file format is ready to be transferred into the flash to be started by the bootloader application.

When the bootloader starts the proxy code it performs some steps to communicate with the cluster-head and waits for tasks to perform. As the first step the proxy code initializes the network peripherals to make available the network interface for communication. Then, it sets the MAC and IP addresses of the softcore which should be set differently for every softcore in the network. After that, the proxy code is ready to run the protocol with the cluster-head. The protocol between the cluster-head and the proxy code is required to perform both server and client applications in both sides. A server/client state bit is stored in the flash memory to keep track of the proxy is acting as a server or client. When the proxy runs a server process, the following process is executed:

1. The proxy code reads the status register from the flash memory location to decide whether to run client application or server application.
2. If the server application is started, which means there is not any job assigned to the softcore, it waits for connection request from the cluster head.
3. When a connection request comes from the cluster-head, it accepts the connection and performs the tasks that are given by the cluster-head. As described in Section 4.2, these tasks can be writing, reading, erasing or and configuration switch operations. A read and write operation in flash memory is a costly operation, so these operations are handled in large arrays. For each operation, the protocol for the proxy code is as follows:
    - **Reading Operation:** The cluster-head sends the offset and size of the data that will be read from the flash memory. The proxy code allocates 0x50000 B of memory in the SDRAM. If the size does not exceeds the allocated space, it stores all the data from the flash memory to the allocated location in SDRAM and sends the data in chunks of 0x400 B to the cluster-head. In case when the size exceeds the allocated memory, it stores the first 0x50000 bytes from the flash memory and sends it to the cluster-head in chunks of 0x400 B. After 0x50000 B

of data is sent, allocated space is used to store the second 0x50000 B of data. This process continues until the proxy-code sends all the necessary data. After the task is finished, the allocated space is freed to be used for later operations.

- **Writing Operation:** The cluster-head sends the offset and size of the data that will be written to the flash memory. In order to write in flash memories, the area should be erased with a erase operation first. After the erasing the necessary memory, the proxy code allocates 0x50000 bytes of memory from the SDRAM. Later, it receives the data in chunks of 0x400 B from the cluster-head and stores it until the allocated space is full. If the size does not exceed the allocated space, the proxy code writes the data into the flash memory. In case, where the size exceeds the allocated space, the proxy code stops receiving data when the allocated space is full and it writes all the data in the allocated space to the flash memory. After the writing process is completed, proxy code continues to receive the data since the allocated memory is available. This process continue until all the data is received and written into the flash memory.
- **Erasing Operation:** In flash memories the erasing operation can be performed in the entire sector. Even to erase a few bytes of the memory in the flash drive, the entire memory sector needs to be erased completely. In our case, Parallel Flash divided into sectors of 0x20000 bytes. In order to perform erase operation, the cluster-head sends the size and offset of the erase operation. The proxy code calculates the number of sectors that need to be erased. Later on, it performs erase

operations for each sector.

- **Configuration Switch Operation:** As described in Section 4.2, the cluster-head sends the configuration switch command along with the offset and size values of the flash memory location that the hardware process writes its results. The proxy code forms a *status register* format to write it into the flash memory for itself to use it in a configuration switch operation. The first byte of the *status register* holds the client-server status information, which in our case, is set to client state. The following 4 bytes holds the IP address of the cluster-head and the last 8 bytes holds the offset and size information. After the *status register* format is formed, it is copied into a specific part of the flash [4]. Then, the proxy code erases the flash memory starting from the offset by the amount of size to make space in the flash memory for the hardware to write its results. As the last step, the proxy code closes its connection with the cluster-head and sends a bit-stream to the CPLD to start configuration switch to configure the hardware.

After the hardware completes its job, it starts another configuration switch to configure the softcore. This time when the proxy code starts, the client-server status register holds the information to start the client application in the proxy code. In the client case of the proxy code, the communication protocol between the proxy code and cluster-head is explained in the following:

1. The proxy code reads the status register from the flash memory and runs the client application.

[4]Address 0x60000 offset in our application

2. Using the information in the status register, proxy code connects to the server in the cluster-head.

3. The proxy code performs a reading operation and sends the results that is obtained by the hardware to the cluster-head using the offset and size values in the status register.

4. After the completion of the transmission, it erases the status register and starts its server application to be ready for the future job requests.

5. When the server application is ready, it disconnects its client application from the cluster-head.

5.2 Hardware Implementations

In this section, we give detailed information on the hardware implementations. Our implementations use word blocks for storage, therefore arithmetic operations are implemented to handle the data as a form of word blocks. Since our implementations are parametrized, bit size and word block size of the hardware architectures can be set to any value. Therefore, they are flexible designs that are able to be used in any architecture by changing their parameters. Since we are performing an attack on 160-bit Huff curves, we set the bit size to 160 bits and the word block number to 10.

5.2.1 Inversion

In this section the hardware implementation of the Montgomery inversion will be described in detail, which is mentioned in section 2.1.2. The Montgomery inversion Algorithm aims to produce $X^{-1}2^m \mod N$ as an output, for the operand X. The algorithm consists of two phases which are given in Algorithm 2 and 3. In the first phase, required inputs are the X as the inversion operand and N as the modulus of the operation. The output of the first phase is: $X^{-1}2^{k-m} \mod N$. The first phase will run the $while\ loop$ for k iterations. Every iteration in the $while\ loop$ is designed to finish the if-else statements in $W + 1$ clock cycles, where W is the total number of word blocks, in our hardware implementation. The total number of clock cycles to finish the $while\ loop$ is: $k * (W + 1)$. After the $while\ loop$ is finished in the first phase, the if statement at the end will add W clock cycles to the overall timing. Therefore, the total number of clock cycles to finish the phase one is: $k * (W + 1) + W$. When the first phase is completed, it passes the $X^{-1}2^{k-m} \mod N$ and k to the second phase in oder to compute the Montgomery inversion. The second phase is basically calculates the multiplication of the given input by 2^{2m-k}. Using the first phase's outputs, the second phase will calculate: $X^{-1}2^{k-m} * 2^{2m-k} \mod N$, which will result in $X^{-1}2^m \mod N$. In the second phase, the for loop will iterate $2m - k$ times. In every iteration the calculation will take $W + 1$ clock cycles and the total number of clock cycles for the second phase will be $(2m - k) * (W + 1)$. This will result in the total number of clock cycles for the Montgomery inversion algorithm as $(2m - k) * (W + 1) + k * (W + 1) + W \rightarrow 2m * (W + 1) + W$.

The implementation of the Montgomery inversion algorithm is a generic and flexible one. As mentioned before, the implementation can be set to any bit length and word length for the desired usage. The algorithm is implemented by dividing the hardware architecture into two main architectures which are named as u-v architecture and r-s architecture. In the following the details of the architectures are given:

- **U-V Architecture:** The most complex part of the architecture is the $u - v$ part, which is used to update u and v values in the algorithm and the architecture is illustrated in Figure 10. Inside the $while\ loop$, an if-else statement is required to check if u is greater than v or not, when they both are not even. The decision of begin u or v is even only takes a clock cycle by checking the least significant bits of these values. However, in a case where they are both odd, the decision of whether u or v is bigger will require a subtraction operation between u and v. In order to reduce the total clock cycles, two subtracters are used for the case which u and v are both odd numbers. In one of the subtracters, u is subtracted from v and in the other one v is subtracted from u. The decision, to select the bigger one, is made by checking the borrow bits from the results of the subtraction operations. Further clock cycle reduction is done by using the division operation in the same clock cycle with the subtraction operations. Since the division operation is a division by 2, a simple one bit shifter is used to perform the division. Since, the decisions of the if-else statements in the $while\ loop$ can be made after performing the subtraction operations, extra storage is needed in order to store temporary results. Therefore, one extra RAM block is implemented for each of the u and v values. The RAM blocks are named as RAM_u, RAM_v, RAM_u_temp and RAM_v_temp. The selection process of the RAM blocks is decided with a control mechanism. There are two signals, u_change and v_change, which are used as write enable and output selection signals. In the beginning of the process, u_change and v_change are set accordingly; to store u and v values into RAM_u and RAM_v and use the RAMs to read real u and v values. Then, in the $while\ loop$ if any u or v value is changed, its change signal also changes to store temporary calculations to other RAM. For instance, RAM_u holds the real value of the u and for few iterations of the $while\ loop$ (i.e. u did not change). Because u is store in RAM_u, all the temporary calculated u values are stored in RAM_u_temp. When, in an if-else statement the u changes, then RAM_u_temp will be used as the real u source and the temporary calculations will be stored in RAM_u. Since this architecture is used to make the decision of the if-else statement, the decisions are sent to the $r - s$ architecture for r and s calculation.

- **R-S Architecture:** As in the $u - v$ architecture, the $r - s$ architecture can also be implemented with extra RAMs to hold temporary values. However, this approach increase the slice usage of the FPGA. Therefore, we implemented another method to decrease the usage of LUTs without an increase in the total clock cycles for the operations, which is illustrated in Figure 11. In this method, the $r - s$ architecture waits for decisions from the $u - v$ architecture. Since this decisions are made after each iteration in the $u - v$ architecture, the $r - s$ architecture receives the decisions by one iteration behind. For instance, when the $u - v$ architecture is in the third iteration of the $while\ loop$, the $r - s$ architecture performs the second iteration. since

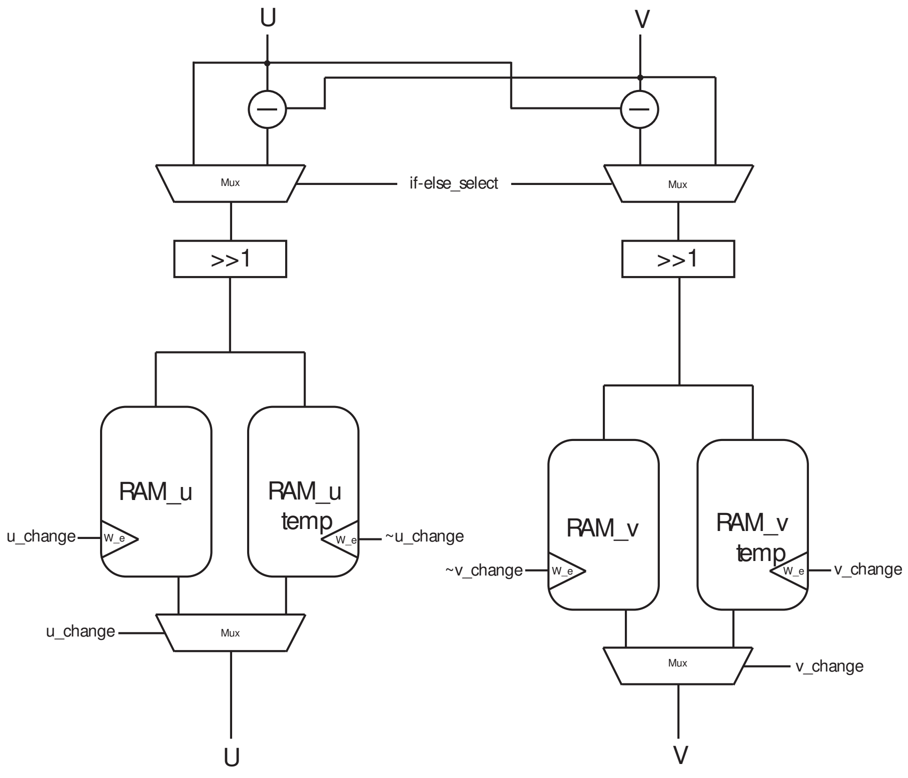

Figure 10: U-V Architecture

the decisions are known beforehand, $r - s$ architecture needs one RAM per r and s values.

The $r - s$ architecture also holds the design to perform the second part of the algorithm for Montgomery inversion. The algorithm 3 is simple to implement by shift and subtraction operations, and to reduce area utilization it was implemented within the $r - s$ architecture. After the calculation of the r and s values, the RAM_r holds the r value and the RAM_s is free for further usage. The second part of the Montgomery inversion algorithm is realized as follows: At start RAM_r is used as the source of r to perform the calculations. In each iteration, r is stored with a shift operation in one RAM and with a shift and a subtraction operation in the other. At the end of each iteration, by looking at the borrow bits, one of the RAM blocks is selected as the updated r for the next iteration.

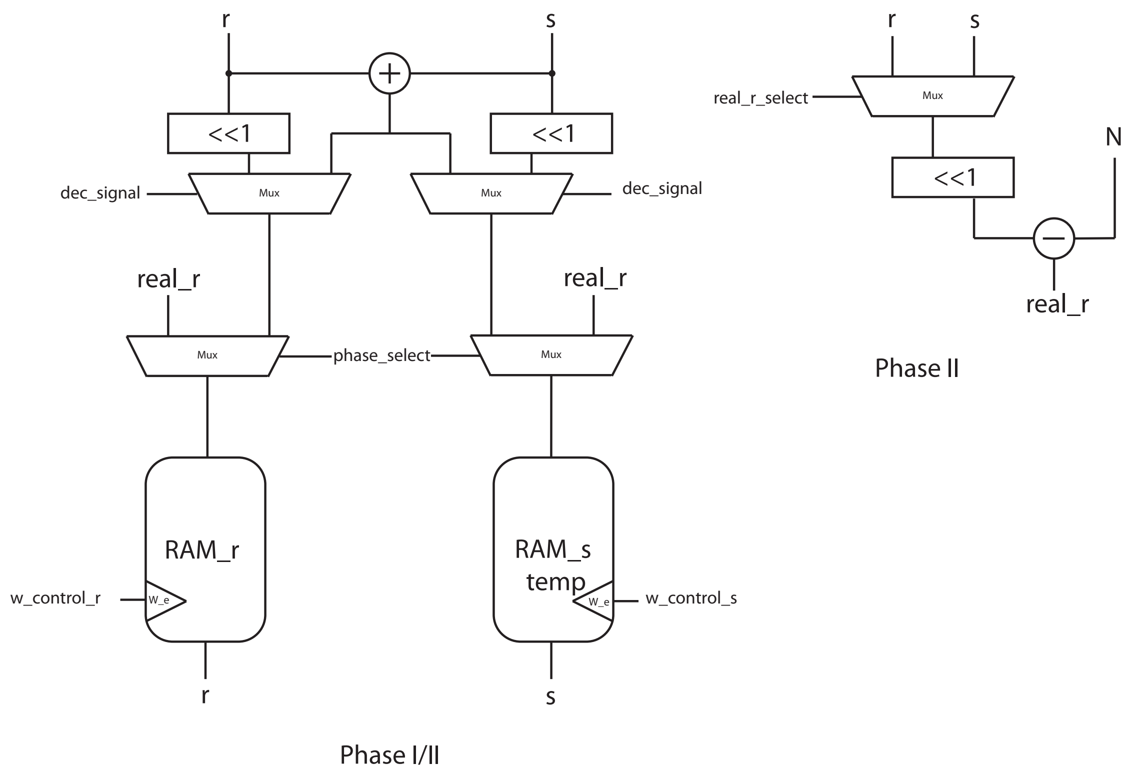

Figure 11: R-S Architecture

5.2.2 Point Addition

The point addition implementation is performed on Huff Curves. Since the design is complex, the point addition is performed by creating a processor that performs modular arithmetic to reduce its complexity. A general overview of the architecture is illustrated in Figure 12. The architecture is formed in three parts which are control logic, ram block and arithmetic unit. The detail information on the architectures is as follows:

Arithmetic Unit: The arithmetic unit consist of four sub-arithmetic operations. The first sub-arithmetic operator is the Montgomery inversion, which was described in detail in Section 5.2.1. The second sub-arithmetic operator is the modular addition/subtraction operation and the same architecture is used for both addition and subtraction with a selection input. Both addition and subtraction are a simple operations and timing for the operations is as follows: W, which is the total number of word blocks, clock cycle to enter the input, two clock cycles to perform the calculation and W clock cycles to output the result. Hence, the total number of clock cycles will be $2 * W + 2$ for all the addition process. The last sub-arithmetic unit is Montgomery multiplication. This arithmetic unit is not implemented rather an efficient, existing design is used from [26].

The arithmetic unit is designed to handle the sub-arithmetic operations with the given inputs. The arithmetic unit takes the operands along with the modulus as inputs and these parameters are transferred in to the sub-arithmetic units. The selection of the sub-arithmetic unit is realized by a opcode. Since there are four arithmetic operations, the opcode takes 2 bit input for operation selection. The input responses of the arithmetic unit for the opcodes is as follows: **2'b00:** The arithmetic unit performs a modular addition. **2'b01:** The arithmetic unit performs a modular subtraction. **2'b10:** The arithmetic unit performs a Montgomery inversion. **2'b11:** The arithmetic unit performs a Montgomery multiplication. Any calculation performed in the sub-arithmetic units, will be given as output with a finished signal. After the completion of the sub-arithmetic operations, the results will take W clock cycles to be outputted from the sub-arithmetic architectures and the results are inserted into the RAM Block by the Control Logic.

RAM Block: The RAM Block holds eleven RAMs for storage. It takes three address buses to select three RAMs that will be used in the calculation process. Two buses are used to select two inputs to feed the Arithmetic Unit for calculations and one bus is used to take the address of the RAM to store the results from the Arithmetic Unit. The Control Logic gives a read signal to the RAM Block so that two selected RAMs will output the values to the Arithmetic Unit to use them in sub-arithmetic operations. After sub-arithmetic operations finish their calculations, the results are written to the selected RAM by activating a write signal by the Arithmetic Unit.

Control Logic: The Control Logic is used to control the arithmetic unit and RAM Block to produce the desired output, which is point addition for the Huff Curve. The calculation algorithm for the point addition of the Huff Curve was given in Section 2.4.1. In our Control Logic, we implemented the point addition operation in twenty steps of arithmetic logic operations. Every step has the following structure to perform the desired operation: Opcode, RAM Address of the operator 1, RAM Address of the operator 2, RAM address of the result. Control Logic manages the execution of the structures in each step by distributing the elements of the structure and performing the steps one at a time.

In order to reduce the total timing in point addition, a trick is performed in the arithmetic of point addition. The costliest operation among the sub-arithmetic units is the Montgomery inversion. The point addition calculation needs to perform this costly operation twice, once to calculate the x coordinate of the point addition and once to calculate the y coordinate of the point addition. In order to reduce the cost, two inversion operations are converted into one inversion and three multiplication operations. The reduction is performed as follows:

1. Lets, a and b are two numbers to be inverted.
2. In two inversion case:
   (a) The a will be inversed by performing one inversion, $a^{-1} \mod N$
   (b) The b will be inversed by performing one inversion, $b^{-1} \mod N$
3. In one inversion and three multiplication case:
   (a) The a will be multiplied by b, which will result in $ab \mod N$
   (b) Then, one inversion operation will be performed on $ab \mod N \rightarrow (ab)^{-1} \mod N$
   (c) Later, to find the inversion of a a multiplication operation will be performed as: $(ab)^{-1} \times b \mod N \rightarrow a^{-1} \mod N$
   (d) Lastly, to find the inverse of b another multiplication will be performed as: $(ab)^{-1} \times a \mod N \rightarrow b^{-1} \mod N$

In the below the twenty steps to calculate the point addition is given for P_1 + P_2 = P_3. The coordinates for the points P_i is shown with the same index i as x_i and y_i. Also, among the eleven RAMs, six of them hold the necessary information that will be used in the calculation process such as x_1, y_1, x_2, y_2, n as modulus and 2^n for Montgomery arithmetic. The other five RAMs are used for temporary storage and their names are R_0, R_1, R_2, R_3, R_4.

1. $R_0 \leftarrow x_1 \times x_2$
2. $R_1 \leftarrow y_1 \times y_2$
3. $R_2 \leftarrow R_0 \times R_1$
4. $R_3 \leftarrow 2^n - R_2$
5. $R_4 \leftarrow R_3 + R_0$
6. $R_4 \leftarrow R_4 - R_1$
7. $R_3 \leftarrow R_3 - R_0$
8. $R_3 \leftarrow R_3 + R_1$
9. $R_2 \leftarrow R_3 \times R_4$
10. $R_2 \leftarrow {R_2}^{-1}$
11. $R_3 \leftarrow R_2 \times R_3$
12. $R_4 \leftarrow R_2 \times R_4$
13. $R_0 \leftarrow 2^n + R_0$

14. $R_1 \leftarrow 2^n + R_1$

15. $R_0 \leftarrow R_0 \times R_3$

16. $R_1 \leftarrow R_1 \times R_4$

17. $R_2 \leftarrow x_1 + x_2$

18. $R_3 \leftarrow y_1 + y_2$

19. $R_0 \leftarrow R_0 \times R_2$ $(output\ 1:\ x_3)$

20. $R_3 \leftarrow R_1 \times R_3$ $(output\ 2:\ y_3)$

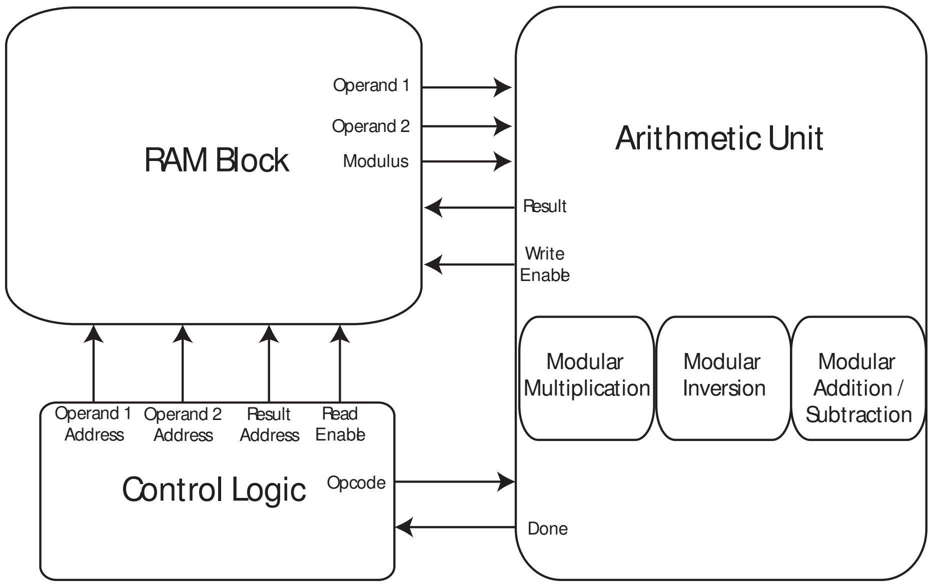

Figure 12: Point Addition Architecture

5.2.3 Pollard Rho

We have implemented the Pollard Rho attack algorithm for Huff curves in our cluster. The computation part of the attack is performed by FPGAs and the data management and FPGA controls are handled by the cluster-head. The attack can be divided into two main parts, as the Server and Huff Curve Attack Architecture:

Server

The cluster-head is our server in the Pollard Rho attack algorithm. It is used to handle the parts that is hard or impossible to handle with hardware, such as data storage, management, control flow of the attack, etc. The server consists of the following modules: **Pollard Rho Point Generator:** The module is used to create a random P, Q and l to create a test attack. It is also used to create random starting points for Huff Curve Attack Architecture. **Database:** The database is used to store distinguished points along with their coefficients. Also it is used in searching distinguished point collusions. **Communication Interface:** It is used to handle the communication between the server and the FPGAs. The realization of the communication is performed via the software library which was described in section 5.1.2. **Modular Arithmetic:** This module is used in case of a collusion to calculate the l_{cal} from the collusion and compare the result with l to check its correctness.

Huff Curve Attack (HCA) Architecture

We have implemented the computational part of the Pollard Rho Attack in hardware to benefit from the parallelization. The HCA architecture consists of several inner architectures to perform the attack. These are configuration switch controller, calculation modules and a control logic. **Configuration Switch Architecture:** Since, the implementation is done by using the configuration switch architecture, a controller is implemented to manage the configuration switch operation when the hardware module completes its operation. **HCA Control Logic:** The HCA Control Logic is used to control the flow of the attack. For instance, it manages the calculation modules to read/write data from/to the flash memory to prevent collusions, since there is only one flash bus. Also it manages order of calculation steps for the calculation modules to operate them in synchronous. **Calculation Module:** The calculation modules are used to calculate the point additions for the attack. Any number of module can be implemented as long as there is available not utilized area in the FPGA. For instance, only two modules can fit into Spartan-3E 500 FPGAs and Spartan-3E 1600 can hold up to six modules. The calculation module consist of the following architecture:

- **Ram Tables:** The ram tables contain the initial and updating values for the calculation. These are Q_x, Q_y, P_x, P_y, R_x, R_y, R_c, R_d and n as the modulus. The points are stored into the RAMs before starting the attack and R_x, R_y, R_c and R_d are the values that are updated in every point calculation.
- **Point Addition Architecture:** The point addition architecture, that is used, was described in section 5.2.2. It takes the inputs from the Ram Tables and calculates the next point and later outputs the results to the RAM Table again.
- **Modular Adder:** The modular adder is used to update the R_c and R_d values. According to the random walk function, only R_c, only R_d or both R_c

and R_d are updated.

- **Flash Module:** The Flash module is used to read/write data from/to the flash memory.
- **Distinguished Point Checker:** The Distinguished Point Checker checks distinguished point property of the calculated output. In our case it means the 15 most significant bits of the point is zero. It is a simple comparator logic and when a point is featured as a distinguished point, a signal is send to the Calculation Module Controller Logic so the point can be written into the flash memory.
- **Calculation Module Controller Logic:** The controller logic controls the flow of the Calculation Module by communicating with the HCA Controller Logic. It manages the flow of point calculations, as well as writing operations of the distinguished points to the flash memory. It waits for a write signal from the HCA Controller Logic to prevent collusions while using Flash Module, since there may exist multiple Calculation Modules in the design.

The visual overview of the Calculation Module and the Pollard Rho Attack Architecture are illustrated in Figures 13 and 14.

These two main parts, the Server and the HCA architecture, are used by creating a protocol between them to perform the Huff Curve Pollard Rho attack on FPGAs. The details of the protocol and the realization of the attack is given as follows:

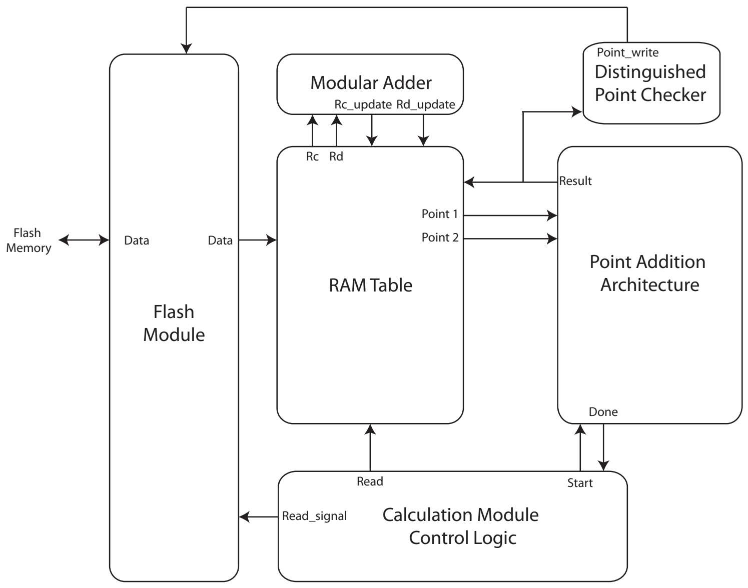

Figure 13: Calculation Module

1. The Server creates a random l for two random points P and Q such as $l \times P = Q$.

2. The Server creates random starting points according to the number of calculation module engines in the FPGA device. Two points per Spartan-3e500 and six points per Spartan-3e1600.

3. The Server converts the points to the Montgomery form. Later, it converts the points into point packages, so that calculation module can use them.

4. The Server connects to all the FPGAs and sends the point data packages to the FPGAs.

5. The Server sends configuration command to the FPGAs and waits for the

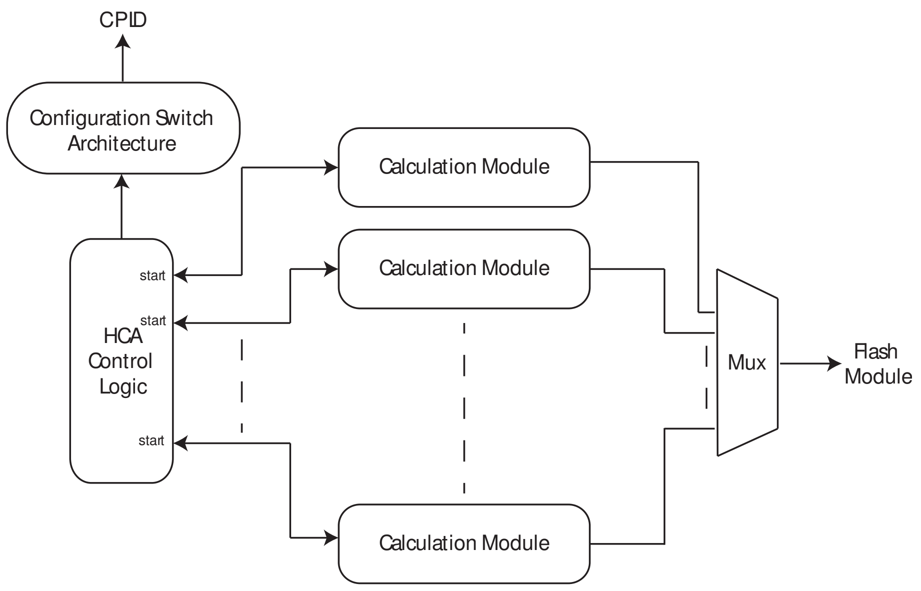

Figure 14: Pollard Rho Attack Architecture

results.

6. The Huff Curve Attack Architecture, distributes the points to the calculation engines and they start to calculate random points.

7. If any point has distinguished point feature, the point is stored with its coefficients in the flash memory.

8. All the calculation module engines need to calculate 25 distinguished points per engine. In other words a total of 50 distinguished points in Spartan-3e500 and 150 points in Spartan-3e1600.

9. When the points are calculated, a configuration switch occurs and the soft-core sends the results to the Server.

10. The Server takes the results and put into a distinguished point list.
11. The distinguished point list is searched for a collusion with the new added points.
12. If a collusion is found, the points are converted back from Montgomery form and the l is calculated. Then, the server waits for other FPGAs to finish their jobs.
13. If there is not a collusion, the last distinguished points[5] of every calculation module engine is converted into point packages.
14. These point packages are send to the FPGA and configuration command is sent so that more distinguished points will be calculated.
15. The Server continues this operation until it finds a collusion and stops the process.

5.3 Experiments

In this section, we give detailed information on the experiments that we performed in our FPGA cluster to measure its performance.

[5]The last distinguished points are chosen, so the calculation module engines can continue to calculate points where they left

5.3.1 First Experiment

The first experiment is intended to find out the efficiency of using the FPGA boards mainly for cryptographic acceleration if the hardware unit and the softcore cannot co-exist in the FPGA board and therefore, configuration switch is necessary. The experiment is performed as follows: the cluster head sends input values (e.g. messages to be encrypted or signed) to the FPGA device which are written in the parallel flash thereafter. To alleviate the timing overhead in communication and writing operations, the data are sent in relatively large chunks since writing large arrays of data to a flash memory gives better timing results. After the transmission is completed, a configuration switch command is sent to the FPGA.

Since we are interested only in the overhead the whole process creates, the hardware unit reads the data from the parallel flash first and then writes it back to it. Without doing anything else, it switches immediately back to the softcore configuration. Following the startup of the softcore, a connection to the cluster head is initiated and the same data in the parallel flash is sent back.

The data exchanged between the cluster head and the FPGA are sent in different packet sizes (i.e. sizes of send/receive buffers in both sides). The timing values obtained through averaging for the first experiment are enumerated in rows 2-4 of Table 1. As can be observed, using large buffer sizes and larger arrays of data helps reducing the overhead. To give an idea what these overhead values actually mean, we study the case where the FPGA board is used for RSA acceleration. Timing overhead for handling 1 MB of data is about 27.31 s, on average. 1 MB of data, for example, means 8192 RSA operation (e.g. signature), where the modulus is 1024 bit. This results in an overhead of 3.33 ms per 1024-bit RSA operation. Considering that the state-of-the-art implementation of RSA for Spartan-3E in [26] executes the same operation in about 8 ms, this roughly increases the effective time per RSA operation by 37% percent, on average. Note that this overhead would be about 0.87 s for 1 KB data size, which is definitely not an acceptable performance for a cryptographic accelerator. In summary, our FPGA cluster may be useful in case the configuration switch is necessary only when we are able to group the input data in large chunks.

Packet Size (B)	Storage Device	*1 KB*	*10 KB*	*100 KB*	*1 MB*
256	Parallel Flash	15.74	17.38	32.70	216.81
512	Parallel Flash	14.40	15.50	21.11	102.90
1024(opt.[3])	Parallel Flash	6.96	7.44	8.79	27.31
1024(opt.[3])	SDRAM	0.27	0.35	1.06	7.81

Table 1: Timing overhead (in seconds) for different data and packet sizes

5.3.2 Second Experiment

In the second experiment, we measured the time to send and receive data of different sizes when configuration switch is not needed for the scenario where the cryptographic unit and the softcore fit in the FPGA device. The cryptographic hardware unit can be directly connected to the softcore peripheral within the FPGA. The SDRAM can be used to store the data since there is no configuration switch that causes the SDRAM to miss refreshment cycles. The timing values for the second experiment are enumerated in the last row of Table 1 only for buffer size of 1024 B. As can be observed from the table, using the SDRAM rather than the parallel flash, decreases the total time by 19.50 s for 1 MB of data.

However, the timing values in the last row of Table 1 should not really be considered as actual overhead for two reasons. Firstly, operations for sending/receiving data and writing/reading to/from the SDRAM can be overlapped with the actual cryptographic computation. Secondly, since the softcore is a microprocessor that can implement any server process reachable from the network, the time spent on the communication and SDRAM access will be similar to the overhead that a PC would incur for the same reason. Therefore, using FPGA without configuration switch does not add significant overhead for many scenarios.

5.3.3 Third Experiment

In the third experiment, we tried to measure the total overhead time when the data transfer is not intense and configuration switch can be used, which is typical mostly for cryptanalytic purposes. The cluster head sends a message of 32 B to the FPGA board, which contains the task description as well as the input parameters. Assuming that hardware unit is a special-purpose design that implements a limited number of interfaces, the task description occupies only a small fraction of the message, leaving the rest for input parameters. Moreover, increasing the message size to a certain extent (double or triple) does not lead to any significant increase in timing spent on communication.

After the task description and input parameters are received, configuration switch occurs and the hardware unit that will accomplish the task takes over the FPGA. On job completion, it writes the results back to the parallel flash and hands over the FPGA back to the softcore via another configuration switch. The softcore reads the results and sends them back to the cluster head. With this, the FPGA becomes available for further tasks. In the experiment, we performed all these steps except for the actual computation time of the task to determine the overhead in time. The timing results for one, two, and three FPGA boards are measured as 7.06 s, 7.09 s, and 7.14 s, respectively.

The interpretation of these timing values can be as follows. When there is one FPGA board available for tasks, 7.06 s after a task is sent to it (excluding the duration of the task itself), the cluster head can send another job. In order to compensate these overhead values, the cluster head should send jobs that will take relatively higher execution times to an FPGA board[6].

The timing values can also be seen as the throughput of the cluster. Assuming one unit of task is sent to the FPGA boards every time, the throughput (number of task units sent in a unit time) increases linearly by the number of FPGAs. For instance, with three FPGA boards, we can initiate approximately three times more task units compared to one FPGA board case.

5.3.4 Fourth Experiment

In the next experiment, we performed an exhaustive key search for PRESENT algorithm [2], which is a lightweight block cipher intended for embedded appli-

[6]Another motivation of sending long lasting jobs to the FPGA boards every time is the fact that there is a constraint on the number of writes that can be performed on the parallel flash memory. Since each task assignment necessitates writings to the flash memory, tasks that will run longer will significantly increase the lifetime of the board.

Conf.	**Area LUT + Slice**	**Max/Usable Freq. (MHz)**	**no of keys tried in $\approx 60s$**
Single PRESENT	$3\% + 3\%$	187.37/NA	NA
13 PRESENT	$74\% + 99\%$	65.23/50	928,628,190 in 61.76 s
7 PRESENT + Softcore	$83\% + 99\%$	53.709/50	510,656,511 in 60.10 s

Table 2: Experimental results for exhaustive key search

cations. The results for a single FPGA board are listed in Table 2.

The second row gives the implementation results for a single encryption engine of the PRESENT algorithm. As can be seen in the third row, the maximum number of encryption engines that will fit in Spartan-3E is only 13 and there is a significant decrease in the clock frequency. This is natural due to two reasons: i) additional control circuit that enables the parallel execution of 13 encryption engines incurs some overhead, and ii) successful optimization for placement-and-routing steps becomes harder for larger designs. With configuration switch and communication costs included, we are able to test about 928 million keys in 61.76 s. Note that in this part of the experiment, the softcore is able to run at a clock rate of 83 MHz since it executes alone in the entire FPGA chip. This 60% increase in operation frequency allows us to accelerate operations in the softcore mode. The last row enumerates the experimental results when there is no configuration switch and the softcore and seven encryption engines run concurrently at the same frequency value of 50 MHz. This experiment demonstrate the advantage of configuration switch for exhaustive key search applications. Note that speed optimized C implementation of the PRESENT algorithm (cf. http://www.lightweightcrypto.org/present/) on a single-threaded PC implementation with an AMD 3.2 GHz quad-core processor and 4 GB RAM can try roughly 106 million keys in 62 s, which also demonstrates that acceleration of PRESENT algorithm is possible.

5.3.5 Fifth Experiment

Finally, we implemented Pollard's Rho method [31, 28], whose experimental setup can be observed in Figure 15, to compute discrete logarithms in elliptic curves over prime fields of odd characteristics. For elliptic curve arithmetic we used Huff model to take advantage of the fast explicit formulae for point additions on Huff curves [14]. Functional units for modular arithmetic operations (i.e. addition, multiplication, and inversion) are designed and optimized for FPGA implementation. The FPGA boards are used to find the distinguished points, which constitutes the most time-consuming part of the computation in Pollard's Rho method. Since a similar approach to the one in [7] is adopted, we only need to implement elliptic curve addition. The hardware implementation of the circuit to find the distinguished points (i.e. distinguished points-generating engine) consumes 50% of the total LUTs, 19% of slice flip-flops, and 5% of the block RAMs. While the design can be synthesized at maximum clock frequency of 70.4 MHz, we operate it at 50 MHz, which is one of the applicable clock frequency values for the FPGA board. In the experiments, we used an elliptic curve defined over a prime field where the prime is a 160-bit integer. The order of the base point is chosen as a 25-bit integer to demonstrate that the discrete logarithm can be computed within a reasonable amount of time using several FPGA boards. Using three boards, each with one instance of the distinguished point-generating engine, our experiments demonstrate that we can compute one discrete logarithm over the described elliptic curve in about 30 minutes, on average. A single-threaded PC implementation on an AMD 3.2 GHz quad-core processor with 4 GB RAM completes the same task in about 6 minutes, on average, using NTL package [34]. Therefore, in order to outperform the single-threaded PC implementation, at least 16 instances of the distinguished point-generating engine must be implemented in the FPGA cluster.

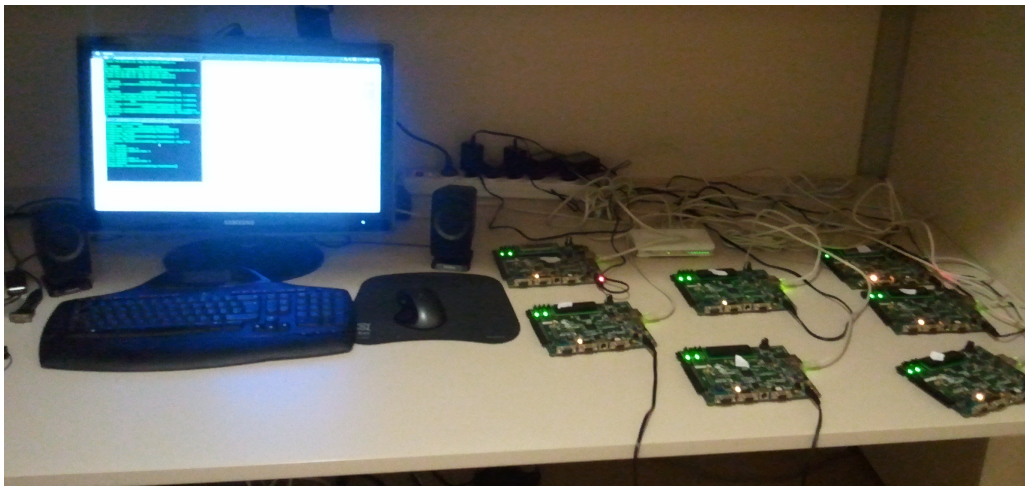

Figure 15: Experimental Setup of FPGA cluster

In order to demonstrate that the time performance of the attack improves linearly with the number of FPGA boards (and the total number of distinguished point-generating engines), we conducted several experiments. Firstly, we optimized the distinguished point-generating engine to fit two instances of it in one Spartan-3E500 and six instances of it in one Spartan-3E1600 devices. Secondly, we employed different number of FPGA boards in our experiments. Using the same curve and the base point mentioned above, we solved different number of elliptic curve discrete logarithm problems and enumerated the timing statistics in Figure 16. As can be observed from the table, we can solve one elliptic curve discrete logarithm problem in about 5.03 minutes using seven FPGA boards (i.e. 22 instances of distinguished point-generating engine), on average[7].

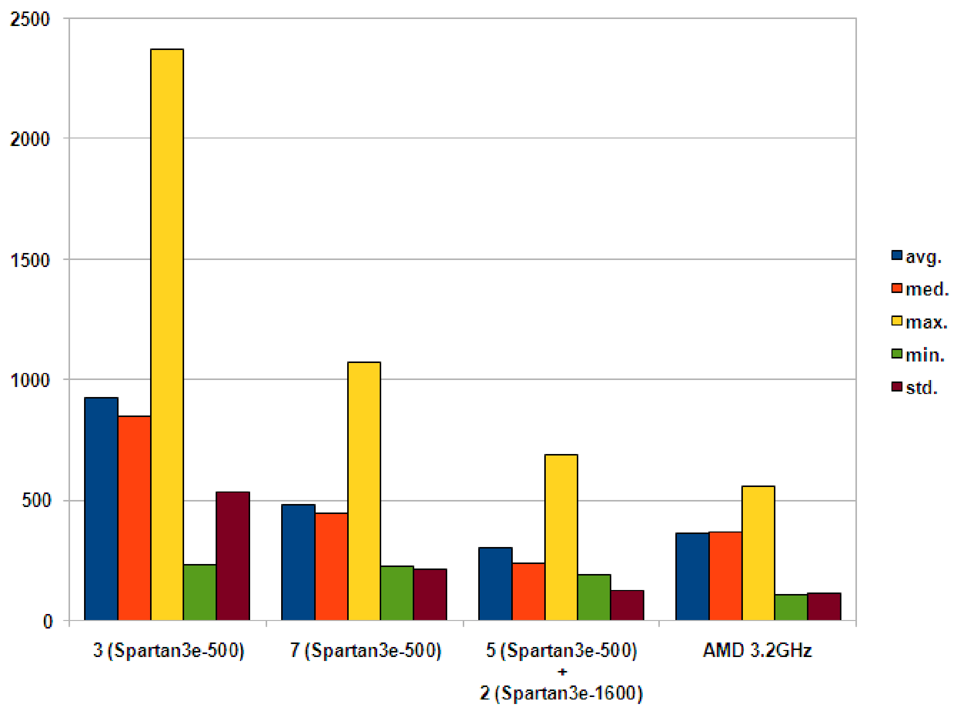

Figure 16: Timing statistics (in seconds) for Pollard Rho's alg. on different number of FPGA boards

In addition, newer versions of the Spartan devices can accommodate more instances of the engine than the FPGA devices used in the experiments. Table 3

[7]We used Asus GIGAX1008B 8 Port 10/100 Layer2 Switch to connect seven FGPA boards.

shows implementation results for the distinguished points-generating engine for different target devices. The FPGA device used in the experiments is very modest in available resources (cf. XC3S500E in the second column of Table 3) while a slightly better and more recent Spartan-3E device (XC3S1600E) is capable of implementing six instances of distinguished points-generating engine resulting in a much higher throughput in a single FPGA device. A new technology in Spartan family, Spartan 6, offers more resources and more than doubles the clock frequency for the design. Much more advanced, yet relatively expensive FPGA devices such as Virtex 6 in Table 3 can offer far more superior performance by accommodating many more instances of distinguished points-generating engine.

Reconfigurable device	**LUT usage**	**Mult./DSP usage**	**Max. Clock Frequency (MHz)**
XC3S500E (Spartan 3E)	4687/9317 (50%)	4/20 (20%)	70.4
XC3S1600E (Spartan 3E)	4660/29504 (15%)	4/36 (11%)	70.4
XC6SLX45T (Spartan 6)	3313/27288 (12%)	4/58 (6%)	179.5
XC6SL150T (Spartan 6)	3313/92152 (3%)	4/180 (2%)	179.5
XC6VLX240T (Virtex 6)	3316/150720 (< 1%)	4/768 (< 1%)	222.5
XC6VLX550T (Virtex 6)	3316/343680 (< 1%)	4/864 (< 1%)	222.5

Table 3: Critical resource usage of the distinguished points-generating engine on different FPGA devices

[8] In these experiments, we performed the attack using three and seven boards of Spartan-3E500, which can fit two instances of Pollard Rho Attack.

[9] In this experiment, we performed the attack using two Spartan-3E1600, which can fit six instances of Pollard Rho Attack, along with five boards of Spartan-3E500.

6 Conclusion and Future Works

FPGAs are powerful candidates to parallelize the cryptographic operations that will be used as to measure the security levels of the cryptographic algorithms against cryptanalytic attacks. On top of it, a cluster formed by a number of FPGAs can increase the computational power linearly. However, there is a trade-off between the utilization of the communication interfaces and the parallelized hardware resources: the more the communication interfaces utilized, the less the parallelized hardware resources. Therefore, it is important to minimize the cost imposed by the communication interfaces.

The experiments demonstrate that the proposed FPGA cluster can be useful for both cryptographic acceleration and implementing cryptanalytic attacks. Dynamic configuration switch between the softcore and the hardware unit, proposed as among the foremost contributions of this work, proves to be useful especially in exhaustive search applications in cryptanalysis, where the need for interprocess communication is very limited (if not absent). Dynamic configuration switch can be useful even for more powerful FPGA devices since FPGA resources salvaged from the softcore can be put into effective use. Moreover, running the softcore alone in the FPGA can be beneficial in increasing the operating frequency.

The proposed FPGA cluster offers advantages (by means of hardware parallelism) over PC-based implementations, when a single FPGA device can accommodate as many instances of the main computation unit as possible. While exhaustive search for simple algorithms, such as PRESENT, can be substantially accelerated, relatively heavy-weight algorithms, such as RSA, does not benefit from the cluster if only one instance of RSA circuit is implemented in one FPGA device. For the acceleration of heavy-weight algorithms, either more advanced FPGA devices or a multitude of simple FPGA devices should be used. There are also the moderate-weight algorithms that lie between the simple- and heavy-weight algorithms, through which an improvement, compared to PCs, can be observed by using sufficient number of FPGAs (i.e. Huff curve attack). Naturally, price performance analysis of the FPGA cluster must be performed on the basis of the specific operation we are trying to accelerate.

In addition to our contributions, the developed software library provides a transparent interface to process data within the hardware modules, yielding even a software programmer to utilize them easily. Besides, our FPGA cluster can benefit from both hardware resources of FPGAs and software resources of the cluster head. For instance, if there were no software resources, i.e. database and control mechanism, in the Huff curve attack, described in Section 5.2.3, it would not be possible to perform the Pollard Rho parallel attack. Nonetheless, the idea of using software resources can be extended to multi-core parallelism of the cryptographic computations in the cluster head, to further decrease the corresponding latency, along with the use of the FPGA cluster.

References

[1] *TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs*, 2006.

[2] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[3] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[4] ECRYPT. Yearly report on algorithms and keysizes. Available from http://www.ecrypt.eu.org/documents/D.SPA.10-1.1.pdf., March 2005.

[5] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[6] T. Güneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.

[7] T. Güneysu, C. Paar, and J. Pelzl. Special-purpose hardware for solving the elliptic curve discrete logarithm problem. *TRETS*, 1(2), 2008.

[8] T. Güneysu, C. Paar, G. Pfeiffer, and M. Schimmler. Enhancing copacobana for advanced applications in cryptography and cryptanalysis. In *FPL*, pages 675–678. IEEE, 2008.

[9] Helion. *High Performance AES (Rijndael) cores for Xilinx FPGA*, 2011. `http://www.heliontech.com/aes.htm`.

[10] P. Huerta, J. Castillo, J. I. Martinez, and V. Lopez. A microblaze based multiprocessor soc. *Wseas Transactions On Systems*, 4:423–430, 2005.

[11] G. B. Huff. *Diophantine problems in geometry and elliptic ternary forms*, volume 15 of *Duke Math.* 1948.

[12] E. II. Yearly Report on Algorithms and Keysizes, 2011.

[13] M. Joye, M. Tibouchi, and D. Vergnaud. Huff's Model for Elliptic Curves. In G. Hanrot, F. Morain, and E. Thomé, editors, *Algorithmic Number Theory, 9th International Symposium, ANTS-IX*, volume 6197 of *Lecture Notes in Computer Science*, pages 234–250, Nancy, France, 2010. Springer. The original publication is available at www.springerlink.com.

[14] M. Joye, M. Tibouchi, and D. Vergnaud. Huff's model for elliptic curves. Cryptology ePrint Archive, Report 2010/383, 2010. `http://eprint.iacr.org/`.

[15] B. S. Kaliski. The montgomery inverse and its applications. *IEEE Trans. Comput.*, 44:1064–1065, August 1995.

[16] I. Kaplansky. *Infinite Abelian groups*. University of Michigan publications in mathematics. University of Michigan Press, 1954.

[17] A. Knapp. *Elliptic curves*. Mathematical notes. Princeton University Press, 1992.

[18] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[19] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Copacobana a cost-optimized special-purpose hardware for code-breaking. In *FCCM*, pages 311–312. IEEE Computer Society, 2006.

[20] A. Le Masle, W. Luk, J. Eldredge, and K. Carver. Parametric encryption hardware design. In P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12133-39.

[21] A. K. Lenstra. Handbook of information security. *IET Computers and Digital Techniques*, 2:617–635, 2005.

[22] V. S. Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO '85: Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer Berlin / Heidelberg, 1986.

[23] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):pp. 519–521, 1985.

[24] NIST. Recommendation for key management part 1: General, NIST special publication 800-57. Available from http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf, August 2005.

[25] C. NIST. The digital signature standard. *Commun. ACM*, 35:36–40, July 1992.

[26] E. Öksüzoglu and E. Savas. Parametric, secure and compact implementation of rsa on fpga. In *Proceedings of the 2008 International Conference on*

Reconfigurable Computing and FPGAs, pages 391–396, Washington, DC, USA, 2008. IEEE Computer Society.

[27] P. C. V. Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12:1–28, 1996.

[28] P. C. V. Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12:1–28, 1996.

[29] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes, 1999.

[30] J. M. Pollard. A monte carlo method for factorization. *BIT*, 15(3):331–334, 1975.

[31] J. M. Pollard. Monte carlo methods for index computation $\pmod p$. *Mathematics of Computation*, 32(143):pp. 918–924, 1978.

[32] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[33] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam. Mpi as a programming model for high-performance reconfigurable computers. *ACM Trans. Reconfigurable Technol. Syst.*, 3:22:1–22:29, November 2010.

[34] V. Shoup. NTL: a library for doing number theory. Online: last accessed, 2011. `http://www.shoup.net/ntl/`.

[35] J. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics Series. Springer, 2010.

[36] Xilinx. *Spartan-3E FPGA Family: Data Sheet*, August 2009. `http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf`.

[37] Xilinx. *MicroBlaze Soft Processor Core*, March 2011. `http://www.xilinx.com/tools/microblaze.htm`.

[38] Xilinx. *Spartan-3E Starter Kit*, March 2011. `http://www.xilinx.com/products/devkits/HW-SPAR3E-SK-US-G.htm`.