Hardware for Collision Search on Elliptic Curve over $GF(2^m)$

Philippe Bulens[*], Guerric Meurice de Dormale[†]
and Jean-Jacques Quisquater

UCL Crypto Group
Université Catholique de Louvain
Place du Levant, 3
1348 Louvain-La-Neuve — Belgium
{bulens, gmeurice, quisquater}@dice.ucl.ac.be

Abstract

In this last decade, Elliptic Curve Cryptography (ECC) has gain increasing acceptance in the industry and the academic community and has been the subject of several standards. This interest is mainly due to the high level of security with relatively small keys provided by ECC. Indeed, no sub-exponential algorithms are known to solve the underlying hard problem, namely the Elliptic Curve Discrete Logarithm Problem (ECDLP).

The aim of this work is to explore the possibilities of a special purpose hardware implementing the best known algorithm for generic curves: the parallelized Pollard's ρ method. In particular, the computing power of a general purpose processor and a reconfigurable hardware platform will be compared. Hardware is expected to perform faster than software but the improving factor is currently unknown. Such results should help to improve the accuracy of the security level offered by a given key size.

1 Introduction

Since their introduction in cryptography in 1985 by N. Koblitz [15] and V. Miller [20], elliptic curves have raised increasing interest. This rich mathematical tool, already known by number theorists for a long time, was used to set up new asymmetric schemes able to compete with the well established RSA. Such schemes allows many useful functionalities like digital signature, public key encryption, key agreements, ... For those needs, Elliptic Curve Cryptography (ECC) is indeed an attractive solution as the public key scheme provided is currently one of the most secure per bit. This means that it presents highly wanted properties like less processing power, storage, bandwidth and power consumption.

The underlying hard problem of ECC is the intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP). This problem is described as follows. Let $E(\mathbb{F})$ be

[*]Supported by Walloon Region / ALIS - First Europe Program
[†]Supported by the Belgian fund for industrial and agricultural research (FRIA)

an elliptic curve over finite field $\mathbb{F}$ and let P be a point in $E(\mathbb{F})$. For any point $R \in \langle P \rangle$ (the subgroup generated by P), determine the integer k, $0 < k < \#P$, (with $\#P$ the order of P) verifying $kP = R$.

In real life utilizations of public key cryptography, the choice of the key sizes is a major concern. In order to reach good trade-offs between security and computation power, it is essential to know the difficulty of solving ECDLP instances. In particular, accurate estimations are especially needed for long-term security keys.

The aim of this work is to evaluate the complexity of solving ECDLPs by means of special purpose hardware. Until now, the only published attacks were performed on general purpose processor. Hardware platforms are expected to perform faster than software but the improving factor, for given problems, is currently unknown. Such results should help to improve the accuracy of the level of security offered by a given key size. A platform gathering current recommended key sizes with regard to the desired security level can be found on `http://www.keylength.com`.

The framework of this paper is a general attack, independent of the representation of the underlying group. Specific attacks, like MOV, Pohlig Hellman and the exploitation of composite fields GF(2^{mn}) are not handled.

As a reminder, Menezes, Okamoto and Vanstone showed that supersingular curves can be mapped isomorphically into the multiplicative group of an extension of the underlying field [18]. Under this mapping, the problem of the elliptic curve logarithm is reduced to the logarithm in a finite field. Methods with sub-exponential complexity like index-calculus could therefore be applied. As a result, only the non supersingular curves are recommended. This work focuses only on this kind of curves. It is also assumed that the order of the chosen base point P has a large prime number. Otherwise, the ECDLP can be solved with the specific Pohlig Hellman attack [17]. Furthermore, it has been showed that composite fields, GF(2^{mn}), are weak and therefore need less powerful attacks [8, 16].

For general attacks, the methods are the Baby-step Giant-step of Shanks [3] or Pollard's ρ method [10]. The algorithm used in this paper is the fully parallelized ρ method [32] with the observations of Teske [31] and the improvements of Wiener and Zuccherato [34].

In 1997, several challenges were issued by Certicom. The aim was to demonstrate real world examples of how difficult it would be to solve a single instance of an ECC key, even at a low strength [4]. They were split into three categories: Exercises; Level I, comprising 109-bit and 131-bit challenges; and Level II comprising 163-bit, 191-bit and 359-bit challenges. The Exercises and the 109-bit challenges are considered feasible, while the 131-bit challenges would require significantly more resources to solve. All Level II challenges are believed to be computationally infeasible [4].

Until now, the hardest challenges solved were done on 109-bit fields by the team of Chris Monico. The (ECC)p-109 challenge was solved in 2002. It required a massive amount of computing power including 10,000 computers (mostly PCs) running 24 hours a day for 549 days. The (ECC)2-109 was solved in 2004. The effort required 2600 computers and took 17 months. For comparison purposes, the gross CPU time used would be roughly equivalent to that of an Athlon XP 3200+ working nonstop for about 1200 years [4].

In this paper, a complete system able to solve the ECDLP is presented. A Field Programmable Gate Arrays (FPGAs) is used for the computations on the elliptic curve, while a host PC searches the collision between the computed points. The arithmetic operations are the core of the method and determine the overall efficiency of the attack.

For validation purpose, implementation results for the small challenge (ECC)2-79 are first provided. Then, results for a real life challenge (ECC)2-163 are given.

The use of an FPGA as the hardware platform is not a coincidence. First, it allows fast prototyping of a demonstrator, useful for the validation of the method. Second, as it is reconfigurable, the circuits can be fully specialized for a given problem. For instance, the use of a hardwired irreducible polynomial greatly increase the efficiency. Third, the achieved performances are not based on hypothetical circuits.

This paper is structured as follows: section 2 reminds the mathematical background of elliptic curves. Section 3 explains the algorithm and the improvements used for the collision search. The description of the architecture of the whole system stands in section 4. Then, the elliptic curve arithmetic core is described in detail in section 5. The hardware and software results are presented in section 6 and finally, the conclusion and some words about further works are given in section 7.

2 Mathematical Background

2.1 Elliptic Curves

The Weierstraß equation of an elliptic curve using affine coordinates is:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

An elliptic curve E is then defined as the set of solutions of this equation in the affine plane, together with the point at infinity $\mathcal{O}$.

For a non supersingular curve equation of characteristic 2 (defined over a field $\mathrm{GF}(2^m)$) and using several *admissible* changes of variables, the reduced Weierstraß equation has the form:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{1}$$

where $a, b \in \mathrm{GF}(2^m)$, $b \neq 0$, together with the point at infinity.

For a thorough description of the topic, the reader is referred to the literature [2].

2.2 Group Law

When E is the curve defined in equation 1, the inverse of point $P = (x_1, y_1)$ is $-P = (x_1, x_1 + y_1)$.

The sum $P + Q$ of points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ (assuming that $P, Q \neq \mathcal{O}$ and $P \neq \pm Q$) is point $R = (x_3, y_3)$ where:

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \frac{(y_2 + y_1)}{(x_2 + x_1)} \end{aligned}$$

If $P = Q$,

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \\ \lambda &= \frac{y_1}{x_1} + x_1 \end{aligned}$$

As it will be explained in section 3, the point doubling law will not be used. Therefore, the focus is put only on the point addition operation. The required modular arithmetic operations are: one division, one multiplication, one squaring and several additions. Of course, an inversion followed by a multiplication can be used instead of a division.

For our application, the cost of the xor gates used for the addition is negligible. As the problem to solve is assumed to be based on low-weight binary irreducible polynomials (as recommended in [22, 25]), the modular reduction is cheap. This is especially true while using reconfigurable logic. Indeed, the modular reduction can be hardwired and therefore exhibits a low complexity. As a result, a squaring is inexpensive. For the modular multiplication, the price to pay for the multiplication itself is not negligible. Nevertheless, sub-quadratic techniques can be used to achieve efficient circuits. For high speed applications, an inversion is much more expensive than a multiplication, even when specific techniques are applied. This topic will be described in detail in section 5.3.

2.3 Choice of Coordinates

To avoid the costly inversion, other coordinate systems could be used. For example, the use of homogeneous projective coordinates maps the representation of the point P from (x, y) to (X, Y, Z) with $x = X/Z$ and $y = Y/Z$. More multiplications and squarings are required to perform the addition and doubling of points, but the inversion is deferred until the end of a whole computation.

Unfortunately, it is not possible to profit from this kind of representation. As it will be explained in section 3, a hash is computed over the x-coordinate after each point addition. Two points can have different X and Y coordinates while having the same x and y coordinates. As a result, if no canonical representation is available, two equivalent points are unlikely to produce the same hash. If a collision occurs, the chains will not merge and the collision will not be detected by the distinguished point of the two chains. For the hash value, an invariant will be sufficient. Unfortunately, to the best of our knowledge, it will always imply a modular division. Those facts will be explained in detail in the next section.

3 Collision Search

When talking about solving general Discrete Logarithm Problems (DLP), two major algorithms come up: the Baby-step Giant-step by Shanks [3] and the ρ method due to Pollard [10]. Shanks'algorithm computes two lists, the baby and the giant steps, and looks for a place where both footprints can be found at the same time. In such a case this collision allows solving the discrete logarithm instance. This method has a space and time complexity in the square root of the group order $(O(\sqrt{n}))$. Pollard improved the method by dramatically reducing the space complexity, while keeping the same time complexity. The parallelized version of this method, with some improvements, is the best known algorithm to solve a generic ECDLP instance. It has been chosen for the hardware architecture presented in this paper.

3.1 Pollard's ρ Algorithm

The basic idea to solve DLP is to walk in the group as randomly as possible until a collision is found, i.e. once a group element is reached twice, coming from different ways. In his paper [10], Pollard uses the concept of random mapping of a finite set and shows how such a theory can be applied to a given sequence to compute discrete logarithms.

As he suggested, the sequence given in his article can vary in many ways, the aim being that the resulting sequence is complicated enough to be considered as a random mapping. Therefore, the algorithm shown here is a general adaptation of his method to the particular case of the ECDLP.

Say the ECDLP is given by an elliptic curve E and two points P and $Q \in \langle P \rangle$ with the known relation $P = k \cdot Q$, holding for some $k \in \{1, \ldots, \#\langle P \rangle - 1\}$.

A random chain is initialized by a point $R_0 = c_0 \cdot P + d_0 \cdot Q$ for which the following function is iteratively applied :

$$R_{i+1} = \begin{cases} R_i + M_u & \text{when } R_i \in T_u \\ 2 \cdot R_i & \text{when } R_i \in T_v \end{cases}$$

The T_u refers to the set of u partitions of the group of points for which the rule is an addition with some random point $M_u = e_u \cdot P + f_u \cdot Q$. Respectively, T_v is the set of v partitions which requires a doubling step. The point at infinity can be reached during this random walk. In that very unlikely case, a new chain has to be initialized.

In order to solve the discrete logarithm, c and d have to be updated according to the path followed by R points, explicitly:

$$c_{i+1} = \begin{cases} c_i + e_u & \text{when } R_i \in T_u \\ 2 \cdot c_i & \text{when } R_i \in T_v \end{cases} \quad \text{and} \quad d_{i+1} = \begin{cases} d_i + f_u & \text{when } R_i \in T_u \\ 2 \cdot d_i & \text{when } R_i \in T_v \end{cases}$$

As time goes by, it is more and more probable to find $R_i = R_j$, yielding the value of the scalar multiplier k :

$$\begin{aligned} c_iP + d_iQ &= c_jP + d_jQ \\ c_iP + d_ikP &= c_jP + d_jkP \\ k &= \tfrac{c_j - c_i}{d_i - d_j} \bmod \#\langle P \rangle \end{aligned}$$

The path followed by successively iterating on R can be drawn as in Fig 1, where the name of Pollard's ρ method now becomes clear. The advantage of this method is that the memory requirements are small if a clever cycle finding method is used. A simple approach is to use Floyd's cycle-finding algorithm [6]. This works as follows: start two sequences described as above, one of them applying twice the iterative function per step instead of only once, and compare the output of the sequences after each step. At some stage of the computation, the two sequences will reach the same point somewhere on the cycle. In this case, the collision occurs between points R_i and R_{2i}, yielding the logarithm of Q. The expected number of elliptic curve group operations is about $3 \cdot \sqrt{n}$, where n denotes the group order. Knowing that the first collision is expected to happen after roughly $\sqrt{\frac{\pi \cdot n}{2}}$ steps, it is clear that this method requires more than twice the necessary work to find a solution. It can be considered as a huge drawback, but do not forget that the memory requirement has been reduced to "peanuts".

Now, an optimal choice for the number of u and v partitions should be determined. This was done in Teske's work [31]. Her experimental results suggest that the best performance is obtained for a ratio $\frac{v}{u}$ between $\frac{1}{4}$ and $\frac{1}{2}$. However, she also noted that, apart from the case $u = 3$, the introduction of doubling steps does not lead to better performance. Moreover the experiments show that the increase in efficiency becomes extremely slow once u gets greater than 20. This is particularly interesting for our design: it allows the point addition to be solely implemented and suggests an easy rule to determine in which partition lies the point. In our case, a hash of the x-coordinate has been chosen. This hash simply returns the 5 least significant bits and obviously yields 32 partitions of the search space.

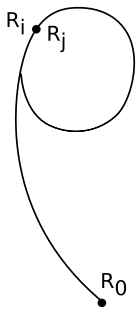

Figure 1: ρ method

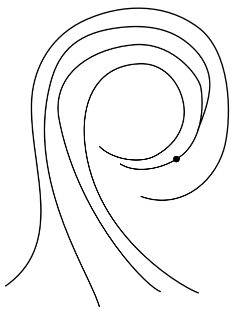

Figure 2: Fully parallelized ρ method

3.2 Improvements

3.2.1 Parallelization

The first occurring improvement is to parallelize the collision search. With M independent machines, the direct parallelization (i.e. each machine following a path on its own side) can already reduce the complexity by a factor $\sqrt{M}$. However, van Oorschot and Wiener [32] showed that it was even more interesting to make the computer share their knowledge, each of them taking part in the global search. The collision search can therefore be viewed as a search between several paths (see Fig 2). The complexity is now cut down by the full factor M.

It may be tempting to think that using projective coordinates would speed up the collision search process. This is actually not the case. As mentioned in section 2.3, projective coordinates (X, Y, Z) do not give a unique representation. This completely wipes out the advantage gained from this kind of representation (i.e. avoiding the inversion) as the points would first have to be converted to affine coordinates (x, y) before applying the hash of the x-coordinate or checking for a distinguished point. To avoid the inversion, cross modular multiplications can not be used since it only works for checking an equality. To sidestep this problem, temptation is high to defer the comparison at a later stage, say after some thousand steps were computed (the cost of one inversion every thousand steps is negligible). Unfortunately, this method does not work: a collision occurring at step i will vanish at the next step. The reason is that the hash of the X-coordinate of the two collided points is quite unlikely to be the same, and the probability gets even lower after some steps. As a conclusion, we are stuck to affine coordinates.

In his paper [10], Pollard also introduced a λ method where a tame kangaroo sets a trap to catch its wild fellow. It is sometimes unclear to determine which methods are kangaroo ones as a zoom near the collision in Fig 2 also looks like a λ. Fortunately, Teske [30] removes the cloud surrounding this point. The intrinsic difference lies in the distance of the jumps. We speak of the ρ method whenever random walks[1] are used (these are the iterative functions described in section 3.1) and of the kangaroo method whenever walks with small jump distance are used.

3.2.2 Distinguished Points

In addition to the full parallelization of the algorithm and Teske's improvements, another generic trick can be included to improve the overall efficiency of the algorithm. This

[1]Only a fool would sustain that kangaroos can jump that far away !

is the concept of distinguished points (DPs), attributed to Rivest by Denning [5]. A distinguished point criteria, or property, is chosen and each computed point satisfying this criteria is stored. Notice that it slightly increases the memory requirements compared with Floyd's cycle-finding algorithm. The collision search is now limited to a comparison between the distinguished points.

A well chosen DP criteria will ensure that enough DPs are encountered to limit the number of steps, but not too much (as those DP are stored). The overhead of Floyd's cycle finding over the expected number of steps before a collision occurs almost vanishes. More precisely: if θ is the proportion of points in $\langle P \rangle$ having the DP property, the expected number of elliptic curve operations per processor before a collision of DP is observed is $\frac{1}{M} \cdot \sqrt{\frac{\pi \cdot n}{2}} + \frac{1}{\theta}$. In practice, the DP property is often chosen as a third of the group order's bit length.

In the case where a chain falls into a loop which contains no DP, a security mechanism is included in the design. A counter is associated to each chain and automatically resets it when the counter value exceeds the bound[2] $20/\theta$. The proportion of chains whose length exceeds the bound is $(1-\theta)^{20/\theta} \approx e^{-20}$. Each abandoned chain is 20 times longer than average. As a result, the proportion of work that is dropped is about $20 \cdot e^{-20} < 5 \cdot 10^{-8}$.

Furthermore, in the short note [26], Smart pointed out that some DP criteria for curves based on GF(2^m) lead to inefficient attacks. In particular, the criteria 0 should be avoided.

3.2.3 Negation Map

For elliptic curves over both GF(p) and GF(2^m), given a point $P = (x, y)$, its opposite can easily be computed by either $-P = (x, -y)$ or $-P = (x, x + y)$. Wiener and Zuccherato [34] made the clever observation that this can reduce the cost of a collision search. Actually, each computed point during the search gives (almost) immediately another point: its opposite. As a result, each step produces two points. This allows limiting the collision search to one half of the space.

This is practically done by replacing R_i by the point $\pm R_i$ with the smallest y-coordinate, for some notion of small in GF(2^m). Accordingly, the coefficients c_i and d_i are replaced by $\pm c_i$ and $\pm d_i$ to ensure the relation $R_i = c_i \cdot P + d_i \cdot Q$ is maintained. For a discussion about trivial 2-cycles, the reader is referred to [34].

As the running time is in the square root of the group order, this trick reduces the time complexity by a factor $\sqrt{2}$.

3.2.4 Frobenius Map

If the ECDLP has to be solved for the specific case of anomalous binary curves, also called Koblitz curves, another improvement is available [34]. This kind of curves, first proposed for cryptographic use in [14], provides an efficient implementation of the scalar multiplication when the scalar is represented as a complex algebraic number. Those curves are defined by the following equation, with either $a = 0$ or $a = 1$ and coordinates over GF(2^m):

$$y^2 + xy = x^3 + ax^2 + 1$$

[2] As the mean length of a chain is $1/\theta$, an excessive length of $20/\theta$ might well indicate a problem within the chain.

The advantage provided by the Koblitz curves is that point doubling can be easily performed by the Frobenius map operation. The usefulness of this property will appear shortly.

Consider the trace equation $\tau^2 + 2 = \mu\tau$ where $\mu = (-1)^{1-a}$ with a being the curve parameter. The solution of this equation is $\tau = \frac{\mu+\sqrt{-7}}{2}$. If point $P = (x, y)$ is on a Koblitz curve, the application of the Frobenius map $\tau P = (x^2, y^2)$, produces a point on the same curve. As a result, a point doubling only needs 2 squarings instead of 2 multiplications, 2 squarings, 1 inversion and 5 additions (in affine coordinates).

As explained in 3.1, Teske recommends using only point additions. This operation will therefore not be used for the random walks but to efficiently compute all the points equivalent to a given point. Practically, after each point addition, the Frobenius map is applied $m - 1$ times over the current point. The resulting point yielding the smallest x-coordinate is kept, again for some notion of small in GF(2^m). The negation map could then be applied. As a result, m points, or $2m$ points are covered at each point addition for a *relatively* small hardware cost. This allows limiting the collision search in $1/m$ of the space or in $1/2m$ with the joint use of the negation map.

This is practically done by replacing R_i by the point $\pm\tau^j R_i$ with the smallest x-coordinate. Accordingly, the (complex) coefficients c_i and d_i are replaced by $\pm\tau^j c_i$ and $\pm\tau^j d_i$ to ensure the relation $R_i = c_i \cdot P + d_i \cdot Q$ is maintained. The powers of τ^j should be precomputed in order to increase the efficiency of the method.

As the running time is in the square root of the group order, the joint use of the negation map and the Frobenius map reduces the time complexity by a factor $\sqrt{2m}$.

4 Collision Search Architecture

For the practical attack, an FPGA board embedding a Xilinx Virtex-II was used. The communication with the FPGA, the storage and the sorting of DPs is achieved with a host PC. The use of a software platform is a natural choice since a huge amount of memory is required to store DPs. Moreover, since the throughput of DPs is slow and the sorting can easily be achieved on general purpose processors, such platform is clearly appropriate for this task. Therefore, only the computer-intensive operations are performed by the hardware. More precisely, the FPGA performs the additive random walks on the elliptic curve and the PC gets back the distinguished points (DPs) found. Each time a new DP is acquired by the PC, it is compared with all the previous DPs in order to find a collision.

The FPGA board is a VirtexII BenNuey PCI board from Nallatech [21] with an XC2V6000-4ff1152 user FPGA. The host PC uses a Windows XP pro OS and is based on a P4 1.8 Ghz processor with 1 Go RAM.

4.1 Global Architecture

On the hardware side, the attack is divided into three different steps. The first two steps have been specifically added in order to ease the management. During the first step, all the starting points are sent to the FPGA and stored in a RAM buffer. The data of each starting point is made up of the x- and y-coordinates of the point $c \cdot P + d \cdot Q$ as well as the c and d coefficients. Throughout the second step, the pipelined EC-core is filled with the starting points. After that, the attack itself can begin. This third step is devoted to the pipelined elliptic curve point addition on parallel chains. For each point addition, the two coefficients c and d are updated in order to keep track of the relationship between the current point and the P and Q points.

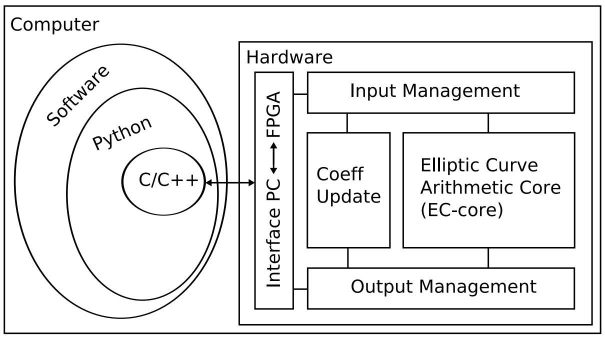

Figure 3: Overview of the architecture.

4.2 Software Implementation

The software is organized as illustrated in Fig 3. Python (v2.4) is used as the high level programming language and wraps the lowest level C/C++ functionalities. Python is an interpreted language but its underlying functions are based on optimized C code.

A C++ module, based on the Victor Shoup's library: NTL (v5.4) [23], is employed for the computation of the starting points. It picks up two random integer in the range $\{1, \cdots, \#\langle P\rangle - 1\}$ and computes the point $c \cdot P + d \cdot Q$. This multiple scalar multiplication is achieved with the joint-NAFs [27] and a simple left-to-right binary method. The performances of this module are not critical but the use of an existing number theory library is convenient.

Another C module, based on the functions provided by the board manufacturer, takes care of the communication with the embedded FPGA. It allows reading and writing data onto the board by opening a DMA channel and transferring a given number of 32-bit blocks.

All these modules were compiled on the host PC with Microsoft Visual Studio (v7.1.3088). They were also wrapped with the swigwin (v1.3.27) [29] tools to make them accessible from the python language. This provides a user-friendly environment where the management of the DP can be easily achieved. In particular, the availability in python of built-in structures like dictionary simplifies the point sorting and collision searching operations. This environment is also convenient for the debugging process.

4.3 Hardware Implementation

The hardware architecture is depicted in Fig 4. The communication bridge between the computer (master) and the FPGA (slave) is made of some logic control and 2 FIFOs. From the FPGA point a view, one is used to read from the computer and the other to write to the computer. An introduction to the hardware interface of the BenNuey board can be found in [7].

The input management module takes care of the incoming 32-bit data blocks and packs them to reconstruct meaningful inputs. For instance, it recovers a starting point and its c and d coefficients thanks to a shift register. After that, this data set is written to the BlockRAMs embedded in the FPGA. Different starting points are collected until there are enough of them to completely fill the EC-core pipeline. When the current points have the DP characteristic, a flag is raised to record the index of the chain corresponding to this event. Indeed, this chain must be reset with a new starting point. If one is ready, the next time the chain enters in the EC-core, the data of the old chain are overwritten by the new starting point. As explained in section 3.2.2, another counter is used to discard the chains exceeding 20 times the expected length. The check for division by zero is currently not implemented since this event is very unlikely.

For the point addition, the decision about the point to add is based on a hash of the x-coordinate. As explained in section 3, the five least significant bits of x are used as a direct map to one of the 32 constant points, associated with the 32 partitions.

In parallel with the computation of the different point additions, the path followed by each chain must be tracked. This operation is completed in the "Coeff Update" module. As explained in section 3.1, this is simply the addition $(\bmod \#\langle P\rangle)$ of the two coefficients of the current point with the coefficients of the constant point to add. More precisely, a pipelined carry-ripple adder is used to avoid long carry propagation delays. Finally, the updated coefficients are written back to the RAMs or replaced by new coefficients when a new starting point is used.

The point addition operation itself is performed by the EC-core. It is a fully pipelined point addition circuit, able to compute a point addition every clock cycle. It is the main module of the hardware part and will be discussed in more detail in the next section.

While the input module packs data, the output module obviously unpacks them. This way, a DP found on any chain is serially shifted out by 32-bit blocks to the output FIFO. The probability to find two DPs in a small time frame is low. Nevertheless, a small buffer, able to store several DPs, has been added in order not to lose them while unpacking the data of the first DP.

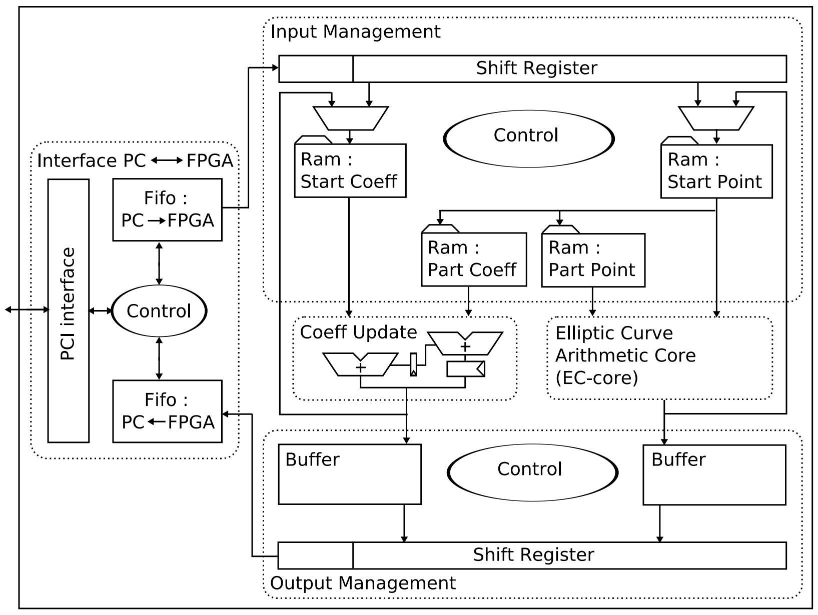

Figure 4: Hardware architecture.

5 Elliptic Curve Arithmetic Core

This section deals with the main part of the collision search system: the hardware elliptic curve arithmetic core. As the chosen method for solving the ECDLP mainly requires efficient point addition operations, this EC-core will determine the overall efficiency of the attack. Usually, hardware implementations focus on inversion-free coordinates as they are more efficient for high-speed implementations (see [1] for example). Nevertheless, as affine coordinates are mandatory, the approach here is radically different.

As a lot of area is available in the targeted circuit, the best solution for the point addition seems to be a fully pipelined circuit. The main advantages are:

- The circuit can be mainly data-driven. As a result, no control part is needed and the circuit is simpler and faster.
- As it is not necessary to iterate over a generic operator, some algorithms with reduced complexity can easily be used[3]. Interesting economy of scale is achieved.
- It is possible to profit from each feature of the algorithms to specialize the circuit. Compared with an iterated architecture with the same throughput, this solution can save area.

As the EC-core is pipelined, the parallelism of the collision search algorithm is used to fill the pipeline with several chains. The number of simultaneous parallel chains is simply equal to the number of register stages in the design. This leads to a high speed circuit computing a point addition every clock cycle. While keeping a high working frequency constraint, the main concern is then the chip area. This criteria is important since it specifies the number of point addition pipelines that can fit in a given platform.

Reasonable assumptions are made about the ECDLP instance to solve. First, the case of a curve based on fields with an optimal normal basis are not considered. Indeed, the extension degrees are recommended to be prime numbers. For such fields, and for sizes of practical interest, optimal normal basis are scarce. Moreover they are not usable with the recommended curves described in [25, 22]. More detail about this topic can be found in [35].

Second, as sketched out in section 2.2, it is assumed that low-weight binary irreducible polynomials are used. This is reasonable since the polynomials recommended by [25, 22] are trinomials and pentanomials. This choice has a direct impact on the complexity of the modular reduction. When hardwiring such polynomials in the circuit, only few xor gates are needed to compute each bits of the modular reduction. As the squaring is a simple linear operation, a cheap modular reduction leads to an inexpensive modular squaring circuit. As it will be showed in section 5.3.2, it is of prior importance for the inverter circuit.

This section is structured as follows: first, the targeted hardware platform is roughly described. Then, the technique used for the modular multiplication is presented. After that, the problem of the modular inversion is tackled. The two main methods are compared in the context of fully pipelined circuits. Finally, a technique to share the costly inverter unit between different point addition pipelines is explained.

[3]This fact will become clearer when we present the design of the multiplier and the inverter.

5.1 Hardware Platform

The selected platform is assumed to be an FPGA made up of 4-input Look-up Tables (LUTs) and registers. In addition to those basic building blocks, large Block RAMs and integer multipliers are expected to be available. The basic area metric, the slice, is composed primarily of 2 4-input LUTs and 2 registers. While this work focuses on Xilinx Virtex FPGAs, the results can nevertheless be extended to other kinds of modern FPGAs.

Of course, as reconfigurable logic is used for the attack, the irreducible polynomials and each subsequent modular reductions can be hardwired. This is particularly interesting when a lot of repeated modular squarings are needed. Addition with constant values can also be hardwired.

5.2 Multiplication

As the EC-core is fully pipelined, the modular multipliers need to be pipelined too. The modular reduction has a negligible complexity and is simply plugged on the output of the multiplier. Enough register stages will be used in order to exhibit a high working frequency. In particular, the *register balancing* feature of the synthesis tool will be used: it helps to move automatically the registers back and forth in the datapath in order to homogenize the critical path.

Under those constraints, the aim is therefore to build a multiplier with the smallest area requirements. The natural way to reach a sub-quadratic complexity is to use the so-called divide-and-conquer Karatsuba-Ofman Algorithm (KOA) [13]. This method recursively splits the problem into several smaller problems. More precisely, a $2^l \times 2^l$ multiplications is reduced to 3^k $(2^{l-k} \times 2^{l-k})$ multiplications and additions. These small multiplications can then be achieved with simple array multipliers.More detail on parallel implementation of the KOA for GF(2^m) can be found in [24] for example. Nevertheless, a slightly different and simpler approach has been chosen.

A first solution is to take a generic $2^l \times 2^l$ KOA multiplier, with $l = \lceil \log_2(m) \rceil$. The unused inputs are hardwired to 0 and the corresponding outputs are not connected. Thanks to the logic optimization provided by the synthesis tool, all the unnecessary logic is removed. While not strictly optimal, this solution is simple and already exhibits good performances. For the number of recursive calls of the KOA method, practical results show that $2^3 \times 2^3$ array multipliers provide good results.

A second solution is to use a KOA multiplier with a recursive division of the operands into two parts instead of parts being power of two. This is possible since there is no constraints about the size of the array multipliers. For instance, a GF(2^{79}) multiplication is designed to use 10×10 array multipliers. A 80×80 multiplier is therefore build and the unnecessary logic is removed by the implementation tools. If the size of the problem slightly exceeds what can be achieved with such kind of array multipliers, the trivial multiplication method can be used at a higher level. For instance, a GF(2^{163}) multiplication is designed to use 10×10 array multipliers for a 160×160 KOA multiplier. Then, two 160×3 and one 3×3 array multipliers are added in order to achieve the 163×163 multiplication.

This second solution performs slightly better than the first. Moreover, it is easier to handle for the implementation tools. Experimental results show that the area required by the modular multiplication is roughly smaller than $(m/2)^2$ FPGA slices.

5.3 Inversion

The problem of the modular inversion has now to be tackled. For instance, a modular division is required in the point addition algorithm (section 2.2), but it can be replaced by an inversion followed by a multiplication. It is the most critical operation and achieving a high throughput implementation is a real headache. For the purpose of this work, two main approaches were studied: inversion and division based on the extended Euclidean algorithm and inversion based on little Fermat's theorem. As comparisons with modular multipliers will be required, the area complexity of this operation will be assumed to be of the order of $(m/2)^2$ FPGA slices.

5.3.1 Extended Euclidean Algorithm

As a first solution, an inverter based on the extended Euclidean algorithm could be used. Only the binary version is of practical interest and can be found in [28]. As this algorithm is iterative, a fully pipelined circuit would require to fully unroll the $2m - 1$ iterations of the loop, with m being the field extension degree. Unfortunately, even when using efficient implementations for FPGA like in [19], the complexity of such a circuit is quite huge.

In order to accurately analyze this approach, several scenarios are considered. For the size of the different circuits, it is important to remember that the worst case has to be assumed. The *slice* metric is employed and only the area of the operative part is considered since it reflects well the complexity of the different solutions.

The binary extended Euclidean algorithm necessitates 4 variables and $2m-1$ iterations to compute the inversion (or the division). The reader is referred to [19] for the algorithm and detail of implementation. Two variables, U and V are required for the binary GCD transformation over the divisor. These transformations are then mirrored on the R and S variables to compute the result. The area complexity of an iterative version of this algorithm is $2m$ slices.

In the worst case scenario, the maximum degree of both U and V starts to decrease after m iterations. As a result, the area complexity of this block is $m^2 + \frac{m^2}{2}$ slices. For the R and S variables, the full size is immediately reached while computing a division. Therefore, $2m^2$ slices are required for this second block. While computing an inversion, the area complexity of the first iterations will depend on the complexity of the irreducible polynomial. Nevertheless, as a division is required, this solution is less attractive since an additional multiplication will be required.

Another solution would be the use of a Montgomery almost inverse algorithm [12]. With this method, the maximum degree of both R and S linearly increases from the first iteration to the m^{th} iteration. As a result, $m^2/2$ slices can be saved. Unfortunately, an additional multiplication is required to map this almost inverse to the appropriate domain. It can be done with a regular multiplication and "precomputed" powers of 2, or repeated modular shifts. Depending on the selected input and output domains, the repeated modular shift method will require $(m/2)^2$ slices (Montgomery $\rightarrow$ Montgomery) or $(2 \cdot m/2)^2$ slices (Classical $\rightarrow$ Classical). Adding the cost of another multiplication to achieve a division, the overall area complexity of this technique renders it useless.

As a result, a solution based on the Euclidean algorithm seems to be quite expensive. Fortunately, another efficient method based on the Little Fermat's theorem could be used.

5.3.2 Little Fermat's Theorem

Another method for the computation of a modular inverse, using a modular exponentiation, is based on little Fermat's theorem. The following relationship holds for all $\beta \neq 0 \in$ GF(2^m):

$$\beta^{-1} = \beta^{2^n-2} = (\beta^{2^{n-1}-1})^2$$

As the GF(2^m) fields have a special group order, an interesting improvement is available. In order to minimize the number of multiplications (the squarings are much cheaper), the multiplication chain technique of Itoh and Tsujii [9] can be employed. This method is based on the identities:

$$\beta^{2^{m-1}-1} = \begin{cases} \beta^{(2^{\frac{m-1}{2}}-1)(2^{\frac{m-1}{2}}+1)} = (\beta^{2^{\frac{m-1}{2}}-1})^{2^{\frac{m-1}{2}}} \cdot \beta^{2^{\frac{m-1}{2}}-1}, m \text{ odd} \\ \beta \cdot \beta^{2^{m-1}-2} = \beta \cdot (\beta^{2^{m-2}-1})^2, m \text{ even} \end{cases}$$

The number of multiplications required to compute this expression is $\lfloor \log_2(m-1) \rfloor + W(m-1) - 1$, where $W()$ stands for the hamming weight function.

As this algorithm needs a lot of repeated squarings and only some multiplications, it could be worth using both Normal Basis (NB) and polynomial basis. NB could be used to efficiently perform the repeated squaring operations (a simple roll shift) while polynomial basis could be used for cost-effective multiplication. Unfortunately, this hybrid approach does not really help. The cost of a parallel circuit for change-of-basis is really huge for general fields and irreducible polynomials. This cost is only acceptable when an optimal normal basis is available. As discussed in the beginning of the current section, such availability is not assumed. Using NB alone is also ineffective since, one way or another, a change-of-basis is needed in the multiplication process.

The utilization of normal basis seems not the path to follow. Fortunately, the use of trinomials and pentanomials does not yield so inefficient repeated squarers. Results show that the required area is less than half the area of a multiplier. Moreover, they are simple to achieve with a scripting language and the *register balancing* feature of the synthesis tool. First, a script can be written to generate the logical expression of repeated modular squarings. Then, several register stages can be added at input and output of the multi-squaring core. For this particular circuit, in order to correctly exploit the *register balancing* feature, registers must be put especially at the inputs of the core. Designs with large operands require also the pre-synthesis of the different modules. Furthermore, a circuit with n repeated squarings consumes less area than n squaring circuits.

This method has been selected for the implementation of the pipelined point addition circuit.

5.4 Inverter Variations

The inverter consumes much more area than the multiplier. For instance, the ratio roughly equals ten. It could therefore be worth reducing the throughput requirement of the inverter at the expense of more multipliers. For this purpose, the Montgomery trick [3] is investigated both for a single point addition pipeline and for multiple point addition pipelines. This method achieves the following area/time tradeoff: with $3(n-1)$ additional multipliers, the throughput of the inverter can be divided by a factor n. The

Montgomery trick is presented in Alg. 1.

Given n elements $a_1, a_2, \dots, a_n$, Montgomery trick may be used to compute their inverse $b_1, b_2, \dots, b_n$ in the following way [3]:

Algorithm 1 Montgomery trick

```
Require: $a_1, a_2, \dots, a_n$
Ensure: Inverses $b_1, b_2, \dots, b_n$ of $a_1, a_2, \dots, a_n$
  $c_1 \leftarrow a_1$
  for $i = 2$ to $n$ do
    $c_i \leftarrow c_{i-1} \cdot a_i$
  end for
  $u \leftarrow c_n^{-1}$
  for $i = n$ downto 2 do
    $b_i \leftarrow c_{i-1} \cdot u$
    $u \leftarrow u \cdot a_i$
  end for
  $b_1 \leftarrow u$
```

For a single point addition pipeline, dividing the throughput by a factor 2 is not so simple. While the multiplications are generic, each repeated squaring circuits are different. As a result, an inverter architecture needing the computation of two different sets of multi-squarings will lose efficiency. Especially as a circuit with n repeated squarings is more efficient than n squaring circuits. This solution has not yet been deeply investigated but the benefit of this method vanishes when multiple point addition pipelines are used.

The FPGAs considered in this work have a high slice count. As a result, the implementation is likely to embed multiple point addition pipelines to increase the throughput. With only two pipelines, the best performances yielded by a full inverter will be showed.

Let us assume a cost of 10 Multiplications (M) for the inverter. If the Montgomery trick is applied on a single pipeline with $n = 2$, the cost of the inverter is $(5+3)M$. Two additional M are also required to complete point addition. The total cost of point addition is therefore $10M$ instead of $12M$ without the Montgomery trick. If this inverter is shared between two pipelines, the first still needs $10M$ while the second requires $(3+3+2)M$: 3 to share the inverter, 3 to accommodate the throughput reduction and 2 to complete point addition. As a result, $18M$ are necessary instead of $24M$ with two simple independent pipelines. However, if a full inverter is allowed, the cost of each additional pipelines is simply $(3+2)M$. Therefore, with 2 pipelines, the cost is $(12+5)M$.

To summarize, the use of a full throughput inverter allows having a cost of $(12+5(p-1))M$, where p is the number of point addition pipelines. The use of a half throughput inverter permits only a cost of $(10+8(p-1))M$. As a result, the Montgomery trick should be used at the multiple pipelines level instead of the single pipeline level.

6 Implementation Results

6.1 Hardware

All the hardware modules described above were written in VHDL. Synthesis and Place & Route were performed using Xilinx ISE 7.1.03i. Table 1 reports the area requirements and working frequency for each module where the targeted device is a Xilinx Virtex2 XC2V6000-4ff1152. It embeds roughly 33000 slices. As explained in section 4, it is the device for the prototype.

Module	Freq. (MHz)	Area (Slices)	BlockRAM
Multiplication	170	2324	0
Inversion	154	15849	0
Point Addition	154	18335	0
Whole Management	124	4361	30
Whole design	100	22236	30

Table 1: Implementation results for GF(2^{79}) after place and route.

All the modules were designed to reach a work frequency of 100 MHz. As the design is fully pipelined, the throughput is one point every clock cycle, that is $100 \cdot 10^6$ point additions per second.

The following table (2) contains the data for GF(2^{163}). The targeted device is a Xilinx Virtex4 XC4VLX200-11ff1512 and embeds roughly 89000 slices. Unfortunately only synthesis results are provided: crashes of the implementation software tools occur while achieving map, place and route. This is triggered by the need of more than 2 Gb of memory; this problem has not yet been solved.

Module	Freq. (MHz)	Area (Slices)	BlockRAM
Multiplication	257	5940	0
Inversion	245	58885	0
Point Addition	242	71469	0
Whole Management	228	8793	60

Table 2: Implementation results for GF(2^{163}) after synthesis.

The Virtex2 can work at a maximum frequency of 200 MHz. Unfortunately, the use of embedded Block RAMs and Multipliers make it difficult to exceed 100 Mhz with a speedgrade of 4. In contrast, the Virtex4 is theoretically able to run at 500 MHz with a speedgrade of 12. The embedded Block Rams and the multiply and accumulate DSP slices of such devices have been designed in order to sustain this maximum frequency. This extension to the Virtex4 is provided to sketch out the performances reachable by state of the art FPGAs.

6.2 Software

To figure out whether hardware implementation can outperform software, a program computing elliptic point addition on several chains was developed using NTL library. The tables below give the timings obtained on a Xeon 3.2 GHz processor with 3 Gb RAM.

The operations analyzed in Table 3 are the multiplication (M) and the inversion (I). Those operations are the most time consuming while performing point additions. The chosen fields are those provided by Certicom [25] for their challenges together with the NIST recommended elliptic curve B-163 [22]. All timings are expressed in seconds.

As discussed in section 5.4, it can be of good practice to share the costly inversion operation when computing point additions on several chains at the same time. Some tests were run in order to choose the accurate number of inputs to combine before performing

m	79	89	97	109	131	163	B-163
1000 M	0.52	0.70	0.97	1.03	2.14	4.06	4.05
1000 I	5.33	6.92	11.63	13.86	28.92	54.02	53.54
I/M	10.33	9.85	12.01	13.43	13.51	13.30	13.23

Table 3: Timings (in seconds) for inversion and multiplication in GF(2^m).

the inversion. A series of experimental results are presented in Table 4 where MT-X denotes the X-input Montgomery trick. Each data set is organized in two rows. The cells of the first row are the timings themselves. In the cells of the second row, those results are weighted in order to take into account the number of parallel chains.

m	79	89	97	109	131	163	B-163
1000 I	5.33	6.92	11.63	13.86	28.92	54.02	53.54
232.79	1240.32	1611.39	2706.21	3226.50	6732.83	12574.29	12463.48
1000 MT-2	6.88	8.88	16.59	17.20	35.52	66.84	65.92
116.40	800.22	1033.02	1931.48	2002.37	4133.93	7780.39	7673.19
1000 MT-4	9.92	13.88	20.97	23.08	48.88	90.30	89.77
58.20	577.44	807.50	1220.36	1343.10	2844.49	5255.18	5224.27
1000 MT-8	15.97	22.36	33.09	35.63	72.63	140.05	139.42
29.10	464.68	650.66	963.00	1036.65	2113.35	4075.30	4057.11
1000 MT-16	28.20	39.61	57.14	60.11	122.44	238.19	236.33
14.55	410.34	576.31	831.37	874.57	1781.43	3465.55	3438.51
1000 MT-17	29.88	40.38	59.89	62.89	128.77	259.27	249.57
13.69	409.10	552.88	820.14	861.22	1763.29	3550.35	3417.48

Table 4: Timings (in seconds) for inversion with MT-X in GF(2^m).

As shown in Table 4, the timing improvement is negligible while exceeding MT-16. Therefore, the 16-input Montgomery trick was used in the point addition program. The last table from this section (Table 5) exhibits the timings for 1000 steps of the algorithm for each of the 16 chains sharing the inversion[4].

The aim of such timing results was of course not to complete an entire attack, but rather to get an idea of the throughput reachable by software. This metric allows making comparison[5] with the throughput of the FPGA.

6.3 Expected Running Time

Although the practical attack was not yet completed, it is still possible to get an idea of achievable performances. From the implementation results and the expected number of

[4]Therefore, these timings should be interpreted as the time required to perform $16 \cdot 1000$ point additions

[5]However, the software timings could be improved since no time was spent to optimize the code with assembly language for example. It essentially relies on the NTL library written exclusively in C/C++ language.

m	79	89	97	109	131	163	B-163
1000 PA	22.39	35.92	26.75	32.78	45.39	66.062	63.78

Table 5: Point addition timings (in seconds).

group operations (ENO) before a collision occurs, a rough expected running time (ERT) can de derived as follow :

$$\text{ERT} = \frac{\text{ENO}}{\text{Throughput}} \quad \text{where ENO} = \frac{1}{M} \cdot \sqrt{\frac{\pi \cdot n}{2}} + \frac{1}{\theta}$$

For the design handling ECC2-79, the various parameters are M[6]$= 1$, the number of processor distributed among a network and $\theta \approx \frac{1}{2^{26}}$, the proportion of distinguished point for a criteria of 26 bits (about the third of the element's bit size). A completely pipelined design as a throughput of one data at each clock cycle, this yields a throughput of $100 \cdot 10^6$ point additions per second. As a result, the FPGA should find a collision after

$$\text{ERT(h,79)} = \frac{\sqrt{\frac{\pi \cdot 2^{79}}{2}} + 2^{26}}{100 \cdot 10^6} \approx 1000 \text{ s} \approx 3 \text{ hours}$$

For comparison purpose, on a single computer, the (non-optimized) software with MT-16 should solve the same ECDLP instance in about

$$\text{ERT(s,79)} = \frac{\sqrt{\frac{\pi \cdot 2^{79}}{2}} + 2^{26}}{\frac{16000}{22.39}} \approx 1000 \text{ years}$$

Nevertheless, as software clients are more common than hardware, the expected number of software-based machines M is likely to be higher than hardware platforms.

The same estimates can be made for the case of ECC2-163. In that case, the size of the DP property is assumed to be 54 bits. The following table summarizes the expected running times for each cases.

	Software	Hardware
GF(2^{79})	1000 years	3 hours
GF(2^{163})	$540 \cdot 10^{12}$ years	$700 \cdot 10^6$ years

Table 6: Expected running time on a single platform.

Despite the fact that the software is clearly not optimized, the hardware seems particularly attractive for cryptanalysis purposes. Nevertheless, even though the hardware performs well, it is (fortunately) unexpected that ECC2-163 will be broken in the near future.

7 Conclusion

In this paper, a hardware architecture dedicated to ECDLP solving has been presented. Generic curves were used and the corresponding best known algorithm was implemented: the parallelized Pollard's ρ method together with several improvements. In order to provide expected timings for the attacks, the design required to solve the small ECDLP instance (ECC)2-79 was developed and constrained for a Xilinx Virtex2 FPGA board. Then, expected results for B-163 NIST curved using state of the art FPGAs were investigated.

[6]The number of chains is equal to the pipeline depth of the whole design.

Solving real-life key sizes is out of reach for this work. To the best of our knowledge, it is nevertheless the first reported hardware attack against certicom's elliptic curve challenge. This should allow the crypto community to have a better idea of the benefits of an hardware-based attack. In particular, such results should help to improve the accuracy of security level offered by a given key size.

8 Further Work

The debugging process of the architecture of the attack against the (ECC2)-79 challenge still need to be completed. The attack is expected to require roughly three hours of computation. Different improvements could then be added like the negation and Frobenius maps. After that, expected results for Certicom's level-I challenges (ECC)2-109 and (ECC)2-131 could be provided. The Gaussian normal basis could also be investigated in order to make comparison with the polynomial basis approach. Finally, an elliptic curve arithmetic core for fields of prime characteristics could be built. This should complete the analysis of ECDLP solving on hardware platforms.

9 Acknowledgment

The authors wish to thank anonymous referees for helpful comments and suggestions.

References

[1] B. Ansari, M. Anwar Hasan, *High Performance Architecture of Elliptic Curve Scalar Multiplication*, CACR Research Report 2006-01, 2006.

[2] I.F. Blake, G. Seroussi, N.P Smart, *Elliptic Curves in Cryptography*, London Mathematical Society, Lecture Notes Series 265, Cambridge University Press, 1999.

[3] H. Cohen, *A course in computational algebraic number theory*, Graduate Text in Mathematics, 138, Springer-Verlag, New York, 1993.

[4] Certicom, `http://www.certicom.com`.

[5] D.E. Denning, *Cryptography and Data Security*, Addison Wesley, 1982.

[6] D. E. Knuth, *The Art of Computer Programming*, vol. 2 : Seminumerical Algorithms , 2nd ed., Addison-Wesley, 1981.

[7] F.-X. Standaert, *PCI to user interface core for BenNUEY*, UCL Crypto Group Technical Report Series, 2003.

[8] P. Gaudry, F. Hess, N. P. Smart, *Constructive and Destructive Facets of Weil Descent on Elliptic Curves*, Journal of Cryptology, vol. 15, no. 1, pp. 19–46, 2002.

[9] T. Itoh and S. Tsujii, *A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases*, Information and Comp., vol. 78, pp. 171–177, 1988.

[10] J.M. Pollard, *Monte Carlo Methods for Index computation (mod p)*, Mathematics of computation, vol. 32, n143, pp. 918–924, July 1978.

[11] J.M. Pollard, *Kangaroos, monopoly and discrete logarithms*, Journal of Cryptology, 13, pp. 437–447, 2000.

[12] B. S. Kaliski Jr., *The Montgomery Inverse and its Applications*, IEEE Transactions on Computers, 44(8), pp. 1064–1065, August 1995.

[13] A. Karatsuba and Yu Ofman, *Multiplication of multidigit numbers on automata*, Soviet Physics Doklady, vol. 7, n 7, pp. 595–596, Jan 1986.

[14] N. Koblitz, *CM curves with good cryptographic properties*, CRYPTO'91, LNCS 576, pp. 279–287, 1991.

[15] N. Koblitz, *Elliptic curve cryptosystems*, Math. of Comp., vol. 48, pp. 203–209, 1987.

[16] A. Menezes, E. Teske, and A. Weng, *Weak fields for ECC*, CT-RSA 2004, LNCS 2964, pp. 366–386, 2004.

[17] A. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwe Academic Publishers, Boston, 1993.

[18] A. Menezes, T. Okamoto, S.A. Vanstone, *Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field*, 22nd ACM Symp. Theory Computing, pp. 80–89, 1991.

[19] G. Meurice de Dormale, J.-J. Quisquater, *Iterative Modular Division over GF(2^m): Novel Algorithm and Implementations on FPGA*, ARC 2006, LNCS, To Appear, 2006.

[20] V. Miller, *Uses of elliptic curves in cryptography*, CRYPTO'85, LNCS 218, pp. 417–426, 1986.

[21] Nallatech, `http://www.nallatech.com`.

[22] U.S. Department of Commerce/National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS)*, FIPS PUB 182-2change1, 2000.

[23] NTL : A Library for doing Number Theory, `http://www.shoup.net/`.

[24] F. Rodríguez-Henríquez, Ç.K. Koç, *On Fully Parallel Karatsuba Multipliers for GF(2^m)*, (394) Computer Science and Technology - 2003, 2003.

[25] Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*, v1.0, 2000.

[26] N. P. Smart, *A note on the x-coordinate of points on an elliptic curve in characteristic two*, Technical Report CSTR-00-019, Department of Computer Science, University of Bristol, December 2000.

[27] J. A. Solinas, *Low-Weight Binary Representations for Pairs of Integers*, CACR Combinatorics and Optimization Research Report 2001-41, 2001.

[28] J. Stein, *Computational problems associated with Racah algebra*, J. Computational Physics, vol. 1, pp. 397–405, 1967.

[29] SWIG : Simplified Wrapper and Interface Generator, `http://www.swig.org/`.

[30] E. Teske, *Computing Discrete Logarithms with the Parallelized Kangaroo Method*, CACR Combinatorics and Optimization Research Report 2001-01, 2001.

[31] E. Teske, *On Random Walks for Pollard's rho method*, Mathematics of computation, vol. 70, n 234, pp. 809–825, 2000.

[32] P. C. van Oorschot and M. J. Wiener, *Parallel Collision Search with Cryptanalytic Applications*, Journal of Cryptology, 12, pp. 1–28, 1999.

[33] P. C. van Oorschot and M. J. Wiener, *Parallel Collision Search with Application to Hash Functions and Discrete Logarithms*, 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, pp. 210–218, November 1994.

[34] M. J. Wiener and R. Zuccherato, *Faster Attacks on Elliptic Curve Cryptosystems*, Selected Areas of Cryptography, Springer, LNCS 1556, pp. 190–200, 1999.

[35] H. Wu, *Low Complexity Bit-Parallel Finite Field Arithmetic Using Polynomial Basis*, CHES 1999, LNCS 1717, pp. 280–291, 1999.

[36] Xilinx Corp. , `http://www.xilinx.com`