Solving the Discrete Logarithm of a 113-bit Koblitz Curve with an FPGA Cluster

Erich Wenger and Paul Wolfger

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, 8010 Graz, Austria
Erich.Wenger@iaik.tugraz.at,
Paul.Wolfger@student.tugraz.at

Abstract. Using FPGAs to compute the discrete logarithms of elliptic curves is a well-known method. However, until to date only CPU clusters succeeded in computing new elliptic curve discrete logarithm records. This work presents a high-speed FPGA implementation that was used to compute the discrete logarithm of a 113-bit Koblitz curve. The core of the design is a fully unrolled, highly pipelined, self-sufficient Pollard's rho iteration function. An 18-core Virtex-6 FPGA cluster computed the discrete logarithm of a 113-bit Koblitz curve in extrapolated 24 days. Until to date, no attack on such a large Koblitz curve succeeded using as little resources or in such a short time frame.

Keywords: elliptic curve cryptography, discrete logarithm problem, Koblitz curve, hardware design, FPGA, discrete logarithm record.

1 Introduction

It is possible to repeatedly fold a standard letter-sized sheet of paper at the midway point about six to seven times. In 2012, some MIT students [28] were able to fold an 1.2 kilometer long toilet paper 13 times. And every time the paper was folded, the number of layers on top of each other doubled. Therefore, the MIT students ended up with $2^{13} = 8192$ layers of paper on top of each other. And poor Eve's job was to manually count all layers one by one.

Similar principles apply in cryptography, although bigger numbers are involved. In Elliptic Curve Cryptography (ECC), where $\lceil \log_2 n \rceil$-bit private keys are used, Eve does not have to iterate through all possible n keys. Instead, Eve would use the more efficient parallelizable Pollard's rho algorithm that finishes in approximately $\sqrt{n}$ steps. The omnipresent question is how big n has to be such that even the most powerful adversaries are not able to reconstruct a private key. Especially in embedded, cost-sensitive applications, it is important to use keys that are only as large as necessary.

Discrete logarithms over elliptic curves were computed in the past [9, 18] and several experimental baselines were established. Since then, committees [3, 6] steadily increased the minimal n by simply applying Moore's law. However, it is

necessary to practically compute discrete logarithms to check to which margin the current standards hold.

The task of computing a discrete logarithm can be split into the work done by researchers and the work done by machines. This paper presents *both* a *novel hardware architecture* and the *discrete logarithm of a 113-bit Koblitz curve*. The highly pipelined, high-speed, and practically extensively tested ECC Breaker FPGA design was used to solve the discrete logarithm of a 113-bit Koblitz curve in extrapolated[1] 24 days using mere 18 FPGAs. Therefore, ECC breaker is the first FPGA design to be used to solve a large discrete logarithm. Further, based on ECC Breaker it is even possible to compute discrete logarithms of even larger binary-field elliptic curves. Substantiated by practical experimentation, this paper will make a notable contribution to the community's activity of breaking larger elliptic curves and carving new standards.

This paper is structured as follows: Section 2 gives an overview on related work. Section 3 revisits some mathematical foundations and Section 4 summarizes the experiments with different iteration functions. The most suitable iteration function was implemented in hardware, which is described in Section 5. As the design is flexible enough to attack larger elliptic curves, Section 6 gives runtime and cost approximations. Section 7 summarizes the learned lessons and Section 8 concludes the paper. Appendix A gives an overview of the targeted curve parameters and pseudo-randomly chosen target points.

2 Related Work

Certicom [10] introduced ECC challenges in 1997 to increase industry acceptance of cryptosystems based on the elliptic curve discrete logarithm problem (ECDLP). They published system parameters and point challenges for different security levels. Since then, the hardest solved Certicom challenges are ECCp-109 for prime-field based elliptic curves, done by Monico *et al.* using a cluster of about 10,000 computers (mostly PCs) for 549 days, and ECC2-109 for binary-field based elliptic curves, also done by Monico *et al.*, computing on a cluster of around 2,600 computers (mostly PCs) for around 510 days. Harley *et al.* [18] solved an ECDLP over a 109-bit Koblitz-curve Certicom challenge (ECC2K-108) with public participation using up to 9,500 PCs in 126 days.

As the Certicom challenges lie far apart (ECCp-131 is 2,048 times more complex than ECCp-109), also self-generated challenges have been broken. A discrete logarithm defined over a 112-bit prime-field elliptic curve was solved by Bos *et al.* [9], utilizing 200 Playstation 3s for 6 months. A single playstation reached a throughput of $42 \cdot 10^6$ iterations per second (IPS). *This work* presents the discrete logarithm of a 113-bit binary-field Koblitz curve that used 18 FPGAs for extrapolated 24 days and reached a throughput of $165 \cdot 10^6$ iterations per FPGA per second.

Further, several attempts and approximations were done in order to attack larger elliptic curves using FPGAs. Dormale *et al.* [24] targeted ECC2-113,

[1] The attack actually run for 47 days but not all FPGAs were active at all times.

ECC2-131, and ECC2-163 using Xilinx Spartan 3 FPGAs performing up to $20 \cdot 10^6$ IPS. Most promising is the work of Bailey *et al.* [5] who attempt to break ECC2K-130 using Nvidia GTX 295 graphics cards, Intel Core 2 Extreme CPUs, Sony PlayStation 3s, and Xilinx Spartan 3 FPGAs. Their FPGA implementation has a throughput of $33.7 \cdot 10^6$ IPS and was later improved by Fan *et al.* [12] to process $111 \cdot 10^6$ IPS. Other FPGA architectures were proposed by Güneysu *et al.* [16], Judge and Schaumont [20], and Mane *et al.* [21]. Güneysu *et al.*'s Spartan 3 architecture performs about $173 \cdot 10^3$ IPS, Judge and Schaumont's Virtex 5 architecture executes $2.87 \cdot 10^6$ IPS, and Mane *et al.*'s Virtex 5 architecture does $660 \cdot 10^3$ IPS.

So far, none of their FPGA implementations have been successful in solving ECDLPs. This work on the other hand presents a *practically tested architecture* which can be used to attack both *Koblitz curves* and *binary-field Weierstrass curves*.

3 Mathematical Foundations

To ensure a common vocabulary, it is important to revisit some of the basics. Hankerson *et al.* [17] and Cohen *et al.* [11] shall be consulted for further details.

3.1 Elliptic Curve Cryptography

This paper focuses on Weierstrass curves that are defined over binary extension fields $K = \mathbb{F}_{2^m}$. The curves are defined as $E/K : y^2 + xy = x^3 + ax^2 + b$, where a and b are system parameters and a tuple of x and y which fulfills the equation is called a point $P = (x, y)$. Using multiple points and the *chord-and-tangent* rule, it is possible to derive an additive group of order n, suitable for cryptography. The number of points on an elliptic curve is denoted as $\#E(K) = h \cdot n$, where n is a large prime and the cofactor h is typically in the range of 2 to 8. The core of all ECC-based cryptographic algorithms is a scalar multiplication $Q = kP$, in which the scalar $k \in [0, n-1]$ is multiplied with a point P to derive Q, where both points are of order n.

As computing $Q = kP$ can be costly, a lot of research was done on the efficient and secure computation of $Q = kP$. A subset of binary Weierstrass curves, known as Koblitz curves (also known as anomalous binary curves), have some properties which make them especially interesting for fast implementations. They may make use of a map $\sigma(x, y) = (x^2, y^2)$, $\sigma(\infty) = (\infty)$, an automorphism of order m known as a *Frobenius automorphism*. This means that there exists an integer λ^j such that $\sigma^j(P) = \lambda^j P$. Another automorphism, which is not only applicable to Koblitz curves, is the negation map. The negative of a point $P = (x, y)$ is $-P = (n-1)P = (x, x + y)$.

3.2 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The security of ECC lies in the intractability of the ECDLP: Given the two points Q and P, connected by $Q = kP$, it should be practically infeasible to

compute the scalar $0 \leq k < n$. As standardized elliptic curves are designed such that neither the Pohlig-Hellman attack [25], nor the Weil and Tate pairing attacks [13, 23], nor the Weil descent attack [15] apply, the standard algorithm to compute a discrete logarithm is Pollard's rho algorithm [26].

The main idea behind Pollard's rho algorithm is to define an iteration function f that defines a random cyclic walk over a graph. Using Floyd's cycle-finding algorithm, it is possible to find a pair of colliding triples, each consisting of a point X_i and two scalars c_i and d_i. As $X_1 = X_2 = c_1P + d_1Q = c_2P + d_2Q$ and $(d_2 - d_1)Q = (d_2 - d_1)kP = (c_1 - c_2)P$, it is possible to compute $k = (c_1 - c_2)(d_2 - d_1)^{-1} \bmod n$. Pollard's rho algorithm expects to encounter such a collision after $\sqrt{\frac{\pi n}{2}}$ steps.

In order to parallelize the attack efficiently, Van Oorschot and Wiener [30] introduced the concept of distinguished points. Distinguished points are a subset of points, which satisfy a particular condition. Such a condition can be a specific number of leading zero digits of a point's x-coordinate or a particular range of Hamming weights in normal basis. Those distinguished points are stored in a central database, but can be computed in parallel. The number of instances running in parallel is linearly proportional to the achievable speedup. Note that each instance starts with a random starting triple and uses one of the iteration functions f which are discussed in the following section.

4 Selecting the Iteration Function

As the iteration function will be massively parallelized and synthesized in hardware, it is crucial to evaluate different iteration functions and select the most suitable one. In this work, the iteration functions by Teske [29], Wiener and Zuccherato [31], Gallant *et al.* [14], and Bailey *et al.* [4] were checked for their practical requirements and achievable computation rates. Table 1 summarizes the experiments done in software on a 41-bit Koblitz curve.

What all iteration functions have in common is that they update a state, henceforth referred to as triple, consisting of two scalars $c_i, d_i \in [0..n-1]$ and a point $X_i = c_iP + d_iQ$. An iteration function f deterministically computes $X_{i+1} = f(X_i)$ and updates c_{i+1} and d_{i+1} accordingly, such that $X_{i+1} = c_{i+1}P + d_{i+1}Q$ holds. A requirement on f is that it should be easily computable and to have the characteristics of a random function.

Teske's *r-adding walk* [29] is a nearly optimal choice for an iteration function. It partitions the elliptic curve group into r distinct subsets $\{S_1, S_2, ..., S_r\}$ of roughly equal size. If a point X_i is assigned to S_j, the iteration function computes $f(X_i) = X_i + R[j]$, with $R[]$ being an r-sized table consisting of linear combinations of P and Q. After approximately $\sqrt{\frac{\pi n}{2}}$ steps, Teske's r-adding walk finds two colliding points for all types of elliptic curves.

The Frobenius automorphism of Koblitz curves can not only be used to speedup the scalar multiplication, but also to improve the expected runtime of a parallelized Pollard's rho by a factor of $\sqrt{m}$. Wiener and Zuccherato [31],

Table 1. Implementation results of all tested iteration functions.

Reference	Iteration function	Expected iterations	Measured iterations
Teske [29]	$f(X_i) = X_i + R[j]$	$929 \cdot 10^3$	$906 \cdot 10^3$
Wiener and Zuccherato [31]	$f(X_i) = \min_{0 \le l < m} \{\sigma^l(X_i + R[j])\}$	$145 \cdot 10^3$	$147 \cdot 10^3$
Gallant *et al.* [14]	$f(X_i) = X_i + \sigma^l(X_i)$	$145 \cdot 10^3$	$166 \cdot 10^3$
Bailey *et al.* [4]	$f(X_i) = X_i + \sigma^{(l \bmod 8)+3}(X_i)$	$145 \cdot 10^3$	$183 \cdot 10^3$

Gallant *et al.* [14], and Bailey *et al.* [4] proposed iteration functions which should achieve this $\sqrt{m}$-speedup.

Wiener and Zuccherato [31] proposed to calculate $f(X_i) = \sigma^l(X_i + R[j]) \; \forall \; l \in [0, m-1]$ and choose the point, which has the smallest x-coordinate when interpreted as an integer. Gallant *et al.* [14] introduced an iteration function based on a labeling function $\mathcal{L}$, which maps the equivalence classes defined by the Frobenius automorphism to some set of representatives. The iteration function is then defined as $f(X_i) = X_i + \sigma^l(X_i)$, where $l = \text{hash}_m(\mathcal{L}(X_i))$. Bailey *et al.* [4] suggested to compute $f(X_i) = X_i + \sigma^{(l \bmod 8)+3}(X_i)$ to reduce the complexity of the iteration function.

Additionally to the Frobenius automorphism, it is possible to use a negation map to improve the expected runtime by a factor of $\sqrt{2}$. The negation map compares X_i with $-X_i$ and selects the point with the smaller y-coordinate when interpreted as an integer. Although the potential speed-up seems very promising, there is an unfortunate challenge associated with the negation map; the problem of fruitless cycles which is discussed in Section 7.

In order to make sure that the potential iteration functions work as promised, a 41-bit Koblitz curve was used to evaluate the iteration functions with a C implementation on a PC (cf. Table 1). As labeling function $\mathcal{L}$, the Hamming weight of the x-coordinate in normal basis was used. The hash function was disregarded. Table 1 summarizes the average number of iterations (computing 100 ECDLPs) of all tested iteration functions using four parallel threads. The experiments showed that the average number of iterations of Gallant's and Bailey's iteration functions are 13-24 % higher compared to the iteration function by Wiener and Zuccherato. Additionally, with a probability of 14-20 % some of the parallel threads produced identical sequences of distinguished points. Restarting the threads regularly or on-demand would counter this problem. Not countering the problem of fruitless threads would increase the average runtime of Gallant's iteration function by another 29 %.

As Wiener and Zuccherato's iteration function achieved the best speed and does not have the problem of fruitless threads, it was chosen to be implemented in hardware. Additionally, by leaving out the automorphism, the hardware can be used to attack general binary-field Weierstrass curves as well.

5 ECC Breaker Hardware

Before the actual hardware architecture and its most critical components are discussed, it is important to review the basic design assumptions and core ideas of the ECC Breaker design.

5.1 Basic Assumptions and Decisions

ASIC vs FPGA Design. In literature it is possible to find a lot of FPGA and ASIC designs optimized for some crucial characteristic. Some authors even dare to compare FPGA and ASIC results. However, several of the largest components in ASIC designs, e.g., registers, RAM, or integer multipliers, are for free in an FPGA design. For instance, every slice comes with several registers. Therefore, adding pipeline stages in a logic-heavy FPGA design is basically for free. For this paper, Xilinx Virtex-6 ML605 evaluation boards were chosen because of availability. Note that all following design decisions were made to maximize the performance of ECC Breaker on this particular board.

Design Goals. As Pollard's rho algorithm is perfectly parallelizeable, the design goal clearly is to maximize the throughput per (given) area. Note that the speed (iterations per seconds) of an attack is linearly proportional to the throughput and inversely proportional to the chip area (more instances per FPGA also increase the speed). Therefore, the most basic design decision was whether to go for many small or a single large FPGA design.

Core Idea. In earlier designs, many area-efficient architectures were considered, each coming with a single $\mathbb{F}_{2^m}$ multiplier, $\mathbb{F}_{2^m}$ squarer, and $\mathbb{F}_{2^m}$ adder per instance. The main problems of these designs were the costly multiplexers and the low utilization of the hardware. Therefore the design principle of ECC Breaker is a single, fully unrolled, fully pipelined iteration function. *In order to keep all pipeline stages busy, the number of pipeline stages equals the amount of triples processed within ECC Breaker.*

ECC Breaker versus Related Work. (i) In the current setup, the interface between ECC Breaker and desktop is a simple, slow, serial interface. This might be a challenge for related implementations but not for ECC Breaker. The implemented iteration function does not require mechanisms to detect fruitless cycles or threads and the on-chip distinguished points (triple) storage assures that only distinguished triples have to be read. (ii) The proposal by Fan *et al.* [12] to perform simultaneous inversion and therefore save some finite-field multipliers was not picked because their proposal introduces many multiplexers and a complex control logic, and might not fully utilize the available hardware. (iii) Further, ECC Breaker stands on its own by coming with prime field $\mathbb{F}_n$ arithmetic which has only a minor impact on the size of the hardware ($< 3\,\%$). Additionally, it proved indispensable during development that the generated distinguished triples could be easily verified.

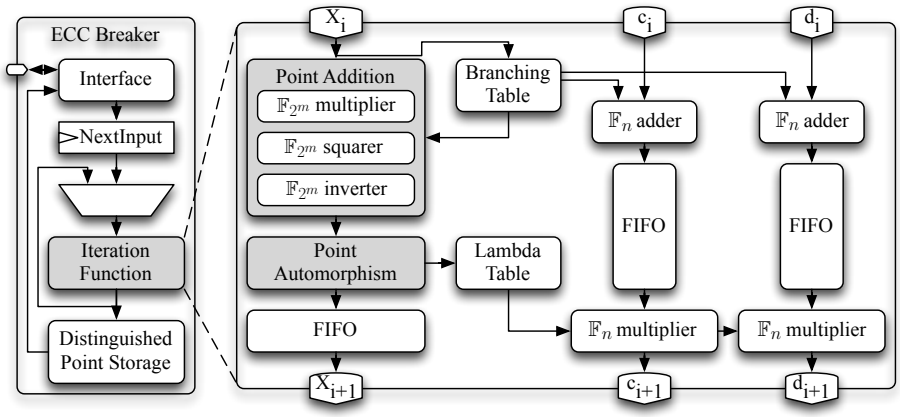

Fig. 1. Top-level view of ECC breaker on the left. Iteration Function on the right.

Generalization of ECC Breaker. Although the current version of ECC Breaker is carefully optimized for a 113-bit binary-field Koblitz curve, the underlying architecture and design approach is also suitable for larger elliptic curves, e.g, a 131-bit Koblitz curve. In Section 6, approximations of the expected runtimes and potential costs to attack such a larger curve are given.

5.2 The Architecture

The basic architecture of ECC Breaker is presented in Figure 1. The core of ECC Breaker is a circular, self-sufficient, fully autonomous iteration function. A (potentially slow) interface is used to write the `NextInput` register. If the current stage of the pipeline is not active, the pipeline is fed with the triple from the `NextInput` register. This is done until all stages of the pipeline process data. If a point is distinguished, it is automatically added to the distinguished point storage (a sufficiently large block RAM). At periodic but time-insensitive intervals the host computer can read all distinguished points that accumulated within the storage.

The iteration function itself consists of several components: a point addition module, a point automorphism module, two $\mathbb{F}_n$ adders, two $\mathbb{F}_n$ multipliers, two block-RAM based tables and several block-RAM based FIFOs to care for data-dependencies.

5.3 ECC Breaker Components

Point addition module. No matter which iteration function is selected, an affine point addition module is always necessary. In the case of binary Weierstrass curves, the formulas for a point addition $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ are $x_3 = \mu^2 + \mu + x_1 + x_2 + a$ and $y_3 = \mu(x_1 + x_3) + x_3 + y_1$, with $\mu = (y_1 + y_2)/(x_1 + x_2)$.

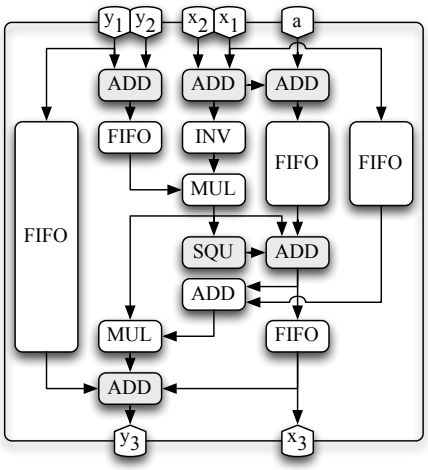

Fig. 2. Simplified point addition module. The grey shaded blocks are without registers.

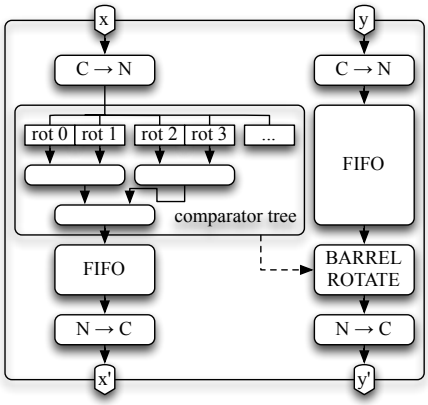

Fig. 3. Point automorphism unit with m comparator units.

Special cases of points being equivalent, inverse of each other, or the identity are not handled by the hardware as they are very unlikely to occur in practice.

Figure 2 shows the implemented point addition module which directly maps the formulas from above. Two $\mathbb{F}_{2^m}$ multiplier, one $\mathbb{F}_{2^m}$ inverter, and five FIFOs are necessary to compute a point addition in 184 cycles. Note that it is not possible to get rid of the costly inversion as the result of the point addition must be available in affine coordinates (cf. [24]).

Point automorphism module. In order to speed-up Pollard's rho algorithm for Koblitz curves, it is necessary to uniquely map m points from the same equivalence class to a single point. As ECC Breaker follows Wiener and Zuccherato's [31] approach of interpreting the field elements as integers and comparing them, it was necessary to design a module that does m squarings and m comparisons as efficiently as possible. This module relies on normal basis representation and is depicted in Figure 3. It converts x and y into normal basis, finds the smallest x within the normal basis, rotates y appropriately, and transforms x and y back into a canonical polynomial representation. As the m exponents of x ($x^{(2^i)}$) are computed by simple rewiring (x rotated by i steps), and the smallest x is found using a binary comparison tree, no canonical $\mathbb{F}_{2^m}$ squarer is needed.

As optimization, only the $t = 70$ most significants bits of x are compared. This means that if two numbers with t equivalent most significant bits are compared, no unique minimum is found. However, the probability for that is only

2^{-t}. For $i = \sqrt{\frac{\pi n}{2m}}$ iterations and $m \cdot i$ comparisons, the probability for not selecting the smaller value is only $1 - (1 - 2^{-t})^{m \cdot i} = 0.00081$ for $m = 113$.

In respect to the overall design, the point automorphism module requires 14% of all slices and is about 5.6 times smaller than the point addition module (in terms of slices). The majority of the point automorphism module is the comparator tree. The basis transformations are fairly cheap and make up only 20% of the point automorphism module.

$\mathbb{F}_{2^m}$ inverse. An Euclidean-based inversion algorithm is not deterministic and therefore hard to compute with a pipelined hardware module. Therefore, ECC Breaker computes the inverse using Fermat's little theorem; an inversion by exponentiation. Fortunately, an exponentiation with 2^{m-2} can be computed very efficiently using Itoh and Tsujii's [19] exponentiation trick, needing 112 squarers and 8 multipliers for $m = 113$: $a = a^{2^1-1} \rightarrow a^{2^2-1} \rightarrow a^{2^3-1} \rightarrow a^{2^6-1} \rightarrow a^{2^7-1} \rightarrow a^{2^{14}-1} \rightarrow a^{2^{28}-1} \rightarrow a^{2^{56}-1} \rightarrow a^{2^{112}-1} \rightarrow a^{2^{113}-2} = a^{-1}$.

$\mathbb{F}_{2^m}$ normal basis. The advantage of a normal basis is that a squaring is a simple rotation operation. The disadvantage of a normal basis is that a $\mathbb{F}_{2^m}$ multiplication is fairly complex to compute. ECC breaker uses per default a normal, canonical polynomial representation.

Only within the point automorphism module the normal basis rendered advantageous. The necessary matrix multiplication for a basis transformation can be implemented very efficiently. As the matrix is constant, on average $m/2$ of the input signals are xored per output signal. Based on the results from Table 2, 666 LUTs are needed per basis transformation.

Experiments show that the normal basis could also reduce the area of the consecutive squaring units within the $\mathbb{F}_{2^m}$ inverse. The 14, 28, and 56 squarers currently need 1 582, 3 164, and 6 328 LUTs, respectively. Doing two basis transformations and a rotation within normal basis would actually save area. Also, accumulating the two transformation matrices into a single matrix would further reduce the area. However, as all squarers together only need 11% of all slices and 10% of all LUTs, the potential area improvement is rather limited. Therefore, contrary to [5, 12], ECC Breaker only uses a normal basis number representation within the point automorphism module.

$\mathbb{F}_{2^m}$ multiplier. As in total ten $\mathbb{F}_{2^m}$ multipliers are needed for the point addition module and the $\mathbb{F}_{2^m}$ inversion module, the $\mathbb{F}_{2^m}$ multipliers have the largest effect on the area footprint of the ECC Breaker design. For ECC Breaker, the following multiplier designs on a Virtex-6 FPGA were evaluated (post-synthesis): (i) A simple 113-bit parallel polynomial multiplier needs 5,497 LUTs. (ii) A Mastrovito multiplier [22] interprets the $\mathbb{F}_{2^m}$ multiplication as matrix multiplication and performs both a polynomial multiplication and the reduction step simultaneously. Unfortunately, it needs 7,104 LUTs. A polynomial multiplication and reduction with the used pentanomial can be implemented much more efficiently.

Table 2. Hierarchical representation of final hardware design (post place-and-route).

Entity	Instances	Cycles	Registers	LUTs	Slices
top	1		58,784	62,655	100%
iteration function	1	210	57,332	60,826	98%
point addition	1	184	35,691	43,177	79%
$\mathbb{F}_{2^m}$ inverse	1	168	29,809	35,126	65%
$\mathbb{F}_{2^m}$ multiplier	8	7	14,958	28,273	51%
$\mathbb{F}_{2^m}$ squarer	112	1	12,543	6,325	11%
$\mathbb{F}_{2^m}$ multiplier	2	7	3,738	7,127	13%
point automorphism	1	16	15,189	14,372	14%
comparator tree	1	7	13,238	10,529	10%
basis transformation	4	1	452	2,664	3%
$\mathbb{F}_n$ multiplier	2	26	3,650	2,000	2%
$\mathbb{F}_n$ adder	2	9	1,308	1,051	1%

(iii) Bernstein [7] combines some refined Karatsuba and Toom recursions for his batch binary Edwards multiplier. The code from [8] for a 113-bit polynomial multiplier needs 4,409 LUTs. (iv) Finally, the best results were achieved with a slightly modified binary Karatsuba multiplier, described by Rodrıguez-Henrıquez and Koç [27]. Their recursive algorithm was applied down to a 16×16-bit multiplier level, which is synthesized as standard polynomial multiplier. The formulas for the resulting multiplier structure are given in Appendix B. The design only requires 3,757 LUTs. At last the design was equipped with several pipeline stages such that it can be clocked with high frequencies.

$\mathbb{F}_n$ multiplier. Computing prime-field multiplications in hardware can be a troublesome and very resource-intensive task. In the case of a Virtex-6, dedicated DSP slices were used for integer multiplications. As a result, the two $\mathbb{F}_n$ multipliers are very resource efficient, requirering only 2×145 DSP slices and 2% of all slices.

6 Results and Transferability of Results

The construction of the current ECC Breaker design was an iterative process that continuously optimized the speed, the area, and the power consumption of all components. To make maximal use of the available resources, the available block RAMs and DSP slices were used whenever possible. Table 2 gives the number of registers and LUTs needed for all components of a 113-bit Koblitz-curve ECC Breaker design. The design was synthesized and mapped with Xilinx ISE 14.6.

ECC Breaker requires (post place-and-route) 47% of all available slices (17,782/37,680), 41% of all LUTs (62,657/150,720), 19% of all registers (58,788/301,440), 37% of all DSP macros (290/768), and less than 10% of all block RAMs. The biggest components are the point addition module and the

Table 3. ECC Breaker on different FPGAs (post synthesis).

Series	**Part Number**	**LUTs** of total		**Registers**	**max Freq.** [MHz]	**Develop. Kit**	**Price** [USD]
Point Addition w/o Automorphism							
Virtex-6	XC6VLX240T	57,294	38%	37,060	261	ML605	2,495
Spartan-6	XC6SLX150T	57,686	62%	37,715	147	LX150T	995
Point Addition w/ Automorphism							
Virtex-6	XC6VLX240T	86,409	57%	55,881	261	ML605	2,495
Artix-7	XC7A200T	86,478	64%	55,848	264	AC701	999
Virtex-7	XC7VX485T	86,391	28%	55,704	313	VC707	3,495
Kintex-7	XC7K325T	86,391	42%	55,704	313	KC705	1,695

$\mathbb{F}_{2^m}$ inverse module. Although already extensively optimized, the 10 $\mathbb{F}_{2^m}$ multipliers require about 64% of all slices. As the place-and-route tool performs optimizations across module borders, the slice counts of all components are just approximations.

6.1 ECC Breaker on Different FPGAs

As the VHDL code is portable, the suitability of ECC Breaker was also evaluated for other Xilinx FPGAs. Although size was a secondary optimization goal, ECC Breaker has been designed for a particular Virtex-6 FPGA. So it does not come with surprise that ECC Breaker does only fit into certain FPGAs that come with certain features. For instance, the used ML605 development board incorporates a Virtex-6 FPGA that comes with 768 DSPs (of which 290 are used).

An overview of synthesis results on different FPGAs is given in Table 3. Fortunately, the ECC Breaker design is perfectly suitable for all kind of the latest Xilinx Virtex-7 (targets high performance designs), Kintex-7 (targets best performance per cost), and Artix-7 (low cost) FPGA devices. The Virtex-7 and Kintex-7 development boards can even fit multiple ECC Breaker instances. However, the Kintex-7 KC705 development board fits the most instances per cost. The prices, taken from `www.avnet.com` [1], do not contain taxes and do not contain potential bulk discounts.

Also smaller FPGAs were considered. Especially the Spartan-6 LX150T-series is of special interest as they are part of SciEngines RIVYERA's S6-LX150 FPGA cluster [2]. Unfortunately, Spartan-6 FPGAs come with 180 DSPs at maximum and therefore the LX150T could be only used to attack non-Koblitz binary-field curves.

6.2 Expected Runtimes

Using the synthesis results from Table 3, several performance approximations of different elliptic-curve targets were performed (cf. Table 4). Note that the results are very optimistic as they are post-synthesis, the FPGAs are running

Table 4. Approximations of best-case runtimes and costs for different targets.

Series	Target	Freq. [MHz]	Inst-ances	FPGAs	Costs [10^3 USD]	Iterations	exp. Runt. [days]
Virtex-6	ECC2K-112	261	1	17	42	$8.5 \cdot 10^{15}$	22
Spartan-6	ECC2K-112	147	1	256	255	$8.5 \cdot 10^{15}$	3
Virtex-6	ECC2-112	261	2	17	42	$90.3 \cdot 10^{15}$	118
Spartan-6	ECC2-112	147	1	256	255	$90.3 \cdot 10^{15}$	28
Kintex-7	ECC2K-130	313	2	590	1,000	$4{,}055.4 \cdot 10^{15}$	127
Kintex-7	ECC2-131	313	2	5,900	10,001	$46{,}239.1 \cdot 10^{15}$	145
Kintex-7	ECC2-163	313	1	589,971	1,000,001	$3{,}030.3 \cdot 10^{21}$	189,934

at the maximum frequency, and a single FPGA contains multiple instances of ECC Breaker.

Computing a discrete logarithm of a 113-bit Koblitz curve (denoted as ECC2K-112) can be done in about 3 days, when a cluster of 256 Spartan-6 FPGAs would be available. With the same cluster, it would be possible to attack a 113-bit binary-field curve (denoted as ECC2-112) in 28 days.

The Certicom challenge ECC2K-130 targeted by Bailey *et al.* [5] and Fan *et al.* [12] can be computed in 127 days, assuming a budget of one million USD. The Certicom challenge ECC2-131 can be computed in 145 days, assuming a budget of ten million USD. Targeting the smallest standardized NIST curve B-163 would take 520 years, assuming a budget of one billion USD, which would be a reasonable budget of certain agencies. However, if there were one billion USD to be spent, it would be more reasonable to go for a dedicated ASIC design.

7 Lessons Learned

After spending months of research time on continuously improving the ECC Breaker design, there are some important insights that need to be discussed.

Maximum Achievable Frequency. As ECC Breaker is a fairly complex and large design, the hardware synthesizer reached its limit when it comes to maximum frequency approximations. In most cases, it was only possible to reach a fraction of the theoretically given frequency after mapping and routing.

Limited by Power Consumption. However, it was not even possible to run the ML605 development boards at maximum post-map-and-route frequency. An average power consumption of about 12 Ampere at the internal power supply resulted in a sporadic emergency switch-off with which the power controller protected itself. This is rather strange, considering that the internal power supply is designed to support 20 Ampere. Therefore it was necessary to reduce the clock frequency further in order to achieve a stable operation. Finally, ECC Breaker was running at 165 MHz even though the synthesizer approximated a maximum clock frequency of 275 MHz.

Multiple Instances per FPGA. It was previously mentioned that some FPGAs fit multiple ECC Breaker instances. However, it has yet to be tested whether two ECC Breaker instances per FPGA at lower clock frequency outperform a single instance, clocked with a higher frequency. Especially under consideration of the previously discussed routing and power problems, multiple ECC Breaker instances per FPGA might not be feasible.

Fruitless Cycles. An initial implementation made use of a negation map. However, the possibility of a fruitless cycle in which $X_{i+1} = f(X_i) = X_i + R[j] - R[j] = X_i$ rendered the hardware implementation with negation map useless. The probability that a fruitless cycle occurs is $p = \frac{1}{2 \cdot m \cdot r}$, r being the number of branches and m the size of the automorphism. The probability to encounter a fruitless cycle after i iterations is $1 - (1 - p)^i$. Given 1,024 branches, an automorphism of size 113, and a clock rate of 165 MHz, the iteration function was trapped in a cycle with a probability of 99% after less than one second. It is subject to future research how to efficiently get rid of the fruitless-cycle problem in a fully pipelined hardware design.

8 Conclusion

This work presents a circular, self-sufficient, highly pipelined, fully autonomous hardware design that was used to practically compute the discrete logarithm of a 113-bit Koblitz curve within extrapolated 24 days on mere 18 Virtex-6 FPGAs. However, because of the scalability and adaptability of ECC Breaker, even more complex results can be expected. This work will bring the community one step closer to solving the ECC2K-130 challenge.

Acknowledgments

The authors are really grateful to Wolfgang Kastl and Jürgen Fuß from the University of Applied Sciences Upper Austria who provided sixteen ML605 boards and continuously supported us and the authors would like to thank the reviewers for their helpful comments.

This work has been supported in part by the Austrian Government through the research program FIT-IT under the project number 835917 (project NewP@ss), by the European Commission through the FP7 program under project number 610436 (project MATTHEW), and the Secure Information Technology Center-Austria (A-SIT).

References

1. Avnet Inc. `http://www.avnet.com/`, Feb 2014.
2. SciEngines GmbH. `http://www.sciengines.com/`, Feb 2014.

3. S. Babbage, D. Catalano, C. Cid, B. de Weger, O. Dunkelman, C. Gehrmann, L. Granboulan, T. Güneysu, J. Hermans, T. Lange, A. Lenstra, C. Mitchell, M. Näslund, P. Nguyen, C. Paar, K. Paterson, J. Pelzl, T. Pornin, B. Preneel, C. Rechberger, V. Rijmen, M. Robshaw, A. Rupp, M. Schläffer, S. Vaudenay, F. Vercauteren, and M. Ward. ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012). Available online at `http://www.ecrypt.eu.org/`, Sep 2012.
4. D. V. Bailey, B. Baldwin, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, G. van Damme, G. de Meulenaer, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, C. Paar, F. Regazzoni, P. Schwabe, and L. Uhsadel. The Certicom Challenges ECC2-X. IACR Cryptology ePrint Archive, Report 2009/466, 2009.
5. D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. V. Herrewege, and B.-Y. Yang. Breaking ECC2K-130. IACR Cryptology ePrint Archive, Report 2009/541, 2009.
6. E. Barker and A. Roginsky. Recommendation for Cryptographic Key Generation. *NIST Special Publication*, 800:133, 2012.
7. D. J. Bernstein. Batch Binary Edwards. In *Advances in Cryptology-CRYPTO 2009*, pages 317–336. Springer, 2009.
8. D. J. Bernstein. Binary Batch Edwards 113-bit Multiplier. Available online at `http://binary.cr.yp.to/bbe251/113.gz`., May 2009.
9. J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction. *International Journal of Applied Cryptography*, 2(3):212, 2012.
10. Certicom Research. The Certicom ECC Challenge. `https://www.certicom.com/index.php/the-certicom-ecc-challenge`, Nov 1997.
11. H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and its Applications. Chapman & Hall/CRC, Boca Raton, FL, 2006.
12. J. Fan, D. V. Bailey, L. Batina, T. Güneysu, C. Paar, and I. Verbauwhede. Breaking Elliptic Curve Cryptosystems Using Reconfigurable Hardware. In *Field Programmable Logic and Applications (FPL)*, pages 133–138. IEEE, 2010.
13. G. Frey and H.-G. Rück. A Remark Concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. *Mathematics of Computation*, 62(206):865–874, 1994.
14. R. Gallant, R. Lambert, and S. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation of the American Mathematical Society*, 69(232):1699–1705, 2000.
15. P. Gaudry, F. Hess, and N. P. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. *Journal of Cryptology*, 15(1):19–46, 2002.
16. T. Güneysu, C. Paar, and J. Pelzl. Attacking Elliptic Curve Cryptosystems with Special-Purpose Hardware. In *FPGA*, page 207. ACM Press, 2007.
17. D. Hankerson, S. Vanstone, and A. J. Menezes. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
18. R. Harley. Elliptic Curve Discrete Logarithms: ECC2K-108, 2000. `http://cristal.inria.fr/~harley/ecdl7/readMe.html`.
19. T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

20. L. Judge and P. Schaumont. A Flexible Hardware ECDLP Engine in Bluespec. In *Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS)*, 2012.
21. S. Mane, L. Judge, and P. Schaumont. An Integrated Prime-Field ECDLP Hardware Accelerator with High-Performance Modular Arithmetic Units. In *Reconfigurable Computing and FPGAs*, pages 198–203. IEEE, Nov. 2011.
22. E. D. Mastrovito. VLSI Designs for Multiplication over Finite Fields GF(2^m). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 297–309, 1988.
23. A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. *Transactions on Information Theory*, 39(5):1639–1646, 1993.
24. G. Meurice de Dormale, P. Bulens, and J.-J. Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *CHES*, pages 378–393. Springer, 2007.
25. S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms over $GF(p)$ and Its Cryptographic Significance. *Transactions on Information Theory*, 24(1):106–110, 1978.
26. J. M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.
27. F. Rodrıguez-Henrıquez and Ç. Koç. On Fully Parallel Karatsuba Multipliers for GF(2^m). In *Computer Science and Technology*, pages 405–410, 2003.
28. C. Stier. Students break record by folding toilet paper 13 times. Available online at `http://www.newscientist.com/blogs/nstv/2012/01/paper-folding-limits-pushed.html`., Jan 2012.
29. E. Teske. Speeding up Pollard's Rho Method for Computing Discrete Logarithms. In *Algorithmic Number Theory*, volume 1423, pages 541–554. Springer, 1998.
30. P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.
31. M. J. Wiener and R. J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In *Selected Areas in Cryptography*, pages 190–200. Springer, 1999.

A Targeted Curve and Target Point Pair Selection

The selection of the curve parameters for a 113-bit Koblitz curve are quite straightforward. However, to proof that the discrete logarithm was actually computed without knowing it in advance, a point generation function was needed. The Sage code in Listing 1.1 was used to deterministically and pseudo-randomly generate two points with order n using Sage 5.12. As P and Q are generated pseudo-randomly, their discrete logarithm is unknown. The Sage script also checks the point orders and the validity of the computed result. Table 5 summarizes all parameters needed for the discrete logarithm computation.

B Binary Karatsuba $\mathbb{F}_{2^{113}}$ Multiplier

Algorithm 1 gives the top-level $\mathbb{F}_{2^{113}}$ multiplier formulas. KS64, KS32, and KS16 are 64-bit, 32-bit, and 16-bit binary Karatsuba multipliers, respectively.

Table 5. Curve parameters of targeted 113-bit elliptic curve.

m	113
irreducible polynomial	$x^{113} + x^5 + x^3 + x^2 + 1$
irreducible polynomial	`0x20000000000000000000000002d`
elliptic curve E	$y^2 + xy = x^3 + ax^2 + b$
curve parameter a	1
curve parameter b	1
order n	`0xffffffffffffffdbf91af6dea73`
cofactor h	2
point $P.x$	`0x0a27644cfced9667d2084f8be061c`
point $P.y$	`0x0d5acd887d5585dd75c5d07165699`
point $Q.x$	`0x189037f88aed8e32400b16d2b1a6e`
point $Q.y$	`0x00e4718fb1e9f50f845ff162ff59c`
scalar k such that $Q = kP$	`0x276c233740d817000b80478fde46`

Algorithm 1 Calculate $c = a \times b$, with a, b being m-bit binary polynomials.

Input: a, b
Output: $c = a \times b$

1: $m_{ab1} \leftarrow (a[112..64] \oplus a[63..0]) \times (b[112..64] \oplus b[63..0])$ ▷ KS64
2: $c_{l1} \leftarrow a[63..0] \times b[63..0]$ ▷ KS64
3: $c_{l2} \leftarrow a[95..64] \times b[95..64]$ ▷ KS32
4: $c_{l3} \leftarrow a[111..96] \times b[111..96]$ ▷ KS16
5: $m_{ab2} \leftarrow (a[95..64] \oplus a[111..96]) \times (b[95..64] \oplus b[111..96])$ ▷ KS32

6: $m_{a3} \leftarrow b[112] \times a[111..96]$
7: $m_{b3} \leftarrow a[112] \times b[111..96]$
8: $m_3 \leftarrow m_{a3} \oplus m_{b3}$
9: $c_3[32] \leftarrow a[112] \times b[112]$
10: $c_3[30..0] \leftarrow c_{l3}$
11: $c_3[31..16] \leftarrow c_3[31..16] \oplus m_3$
12: $m_2 \leftarrow m_{ab2} \oplus c_{l2} \oplus c_3$
13: $c_2[62..0] \leftarrow c_{l2}$
14: $c_2[97..64] \leftarrow c_3$
15: $c_2[94..32] \leftarrow c_2[94..32] \oplus m_2$
16: $m_1 \leftarrow m_{ab1} \oplus c_{l1} \oplus c_2$
17: $c[126..0] \leftarrow c_{l1}$
18: $c[225..128] \leftarrow c_2$
19: $c[190..64] \leftarrow c[190..64] \oplus m_1$

Listing 1.1. Sage code to verify P, Q, and $Q = kP$.

```
m=113
a=1
b=1
h=2
n=0xffffffffffffffdbf91af6dea73
k=0x276c233740d817000b80478fde46
FF = sage.rings.finite_rings.finite_field_ext_pari.\
     FiniteField_ext_pari;
K = FF(2**m, 'x')
x=K.gen()
E = EllipticCurve(K, [1,a,0,0,b])

def str_to_poly(str):
    I=Integer(str, base=16)
    v=K(0)
    for i in range(0,K.degree()):
        if (I >> i) & 1 > 0:
            v = v + x^i
    return v

def poly_to_str(poly):
    vec=poly._vector_()
    string = ""
    for i in range(0,len(vec)):
        string = string + str(vec[len(vec) - i - 1])
    return hex(Integer(string, base=2))

import hashlib
PX = str_to_poly(hashlib.sha256(str(0)).hexdigest())
PY=PolynomialRing(K, 'PY').gen()
P_ROOTS = (PY^2+PX*PY+PX^3+a*PX^2+b).roots()
P=E([PX,P_ROOTS[0][0]]); P=P*h

QX = str_to_poly(hashlib.sha256(str(1)).hexdigest())
Q_ROOTS = (PY^2+QX*PY+QX^3+a*QX^2+b).roots()
Q=E([QX,Q_ROOTS[0][0]]); Q=Q*h

print 'P.x:', poly_to_str(P[0])
print 'P.y:', poly_to_str(P[1])
print 'Q.x:', poly_to_str(Q[0])
print 'Q.y:', poly_to_str(Q[1])
print k*P==Q, is_prime(n), (n*P).is_zero(), (n*Q).is_zero()
```