manuscript No.
(will be inserted by the editor)

Harder, Better, Faster, Stronger - Elliptic Curve Discrete Logarithm Computations on FPGAs

Erich Wenger · Paul Wolfger

Received: date / Accepted: date / Compiled: February 20, 2015

Abstract Computing discrete logarithms takes time. It takes time to develop new algorithms, choose the best algorithms, implement these algorithms correctly and efficiently, keep the system running for several months, and, finally, publish the results. In this paper, we present a highly performant architecture that can be used to compute discrete logarithms of Weierstrass curves defined over binary fields and Koblitz curves using FPGAs. We used the architecture to compute for the first time a discrete logarithm of the elliptic curve `sect113r1`, a previously standardized binary curve, using 10 Kintex-7 FPGAs. To achieve this result, we investigated different iteration functions, used a negation map, dealt with the fruitless cycle problem, built an efficient FPGA design that processes 900 million iterations per second, and we tended for several months the optimized implementations running on the FPGAs.

Keywords: Elliptic curve cryptography, discrete logarithm problem, hardware design, FPGA, negation map.

1 Introduction

Cryptographic research is a constant race between designers and attackers. In the best case, the challenges given by the designers scale exponentially and the resources used by the attacker scale linearly. The consequential fundamental question is how big the cryptographic parameters need to be so that no attack is feasible.

One very efficient family of public-key systems is elliptic curve cryptography (ECC). Its performance is directly proportional to the size of the (security) parameters used. The security is based on the intractability of the elliptic curve discrete logarithm problem (ECDLP). A challenge, especially in constrained environments, is to choose elliptic curve parameters that simultaneously enable efficient implementations and reasonably security.

Standardisation committees [1,4] rely on ECDLP performance results to derive appropriately secure elliptic curve standards. Therefore, a constant evaluation of different techniques and technologies is necessary to keep track of the capabilities of the most sophisticated attackers. In the past, ECDLPs were computed using public participation [22] or Playstation-3 clusters [8]. In this paper, which is an extension of our SAC 2014 paper [35], we investigate the power of FPGAs to practically compute ECDLPs.

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 elliptic curve defined over a binary field.* This discrete logarithm was computed using a fully pipelined, high-speed, and extensively tested design, ECC Breaker, and 10 Kintex-7 FPGAs. Based on our SAC 2014 paper [35], this paper additionally investigates how to implement negation maps in hardware and how to apply the simultaneous inversion technique. We use the new, faster design to compute the discrete logarithm of a 10.6-times stronger elliptic curve. Further-

E. Wenger
Graz University of Technology
Inffeldgasse 16a/I, A-8010 Graz, AUSTRIA
E-mail: Erich.Wenger@iaik.tugraz.at

P. Wolfger
so-logic GmbH Co KG
Lustkandlgasse 52/22, A-1090 Vienna, AUSTRIA
E-mail: pwo1@so-logic.net

more we will show that it is even possible to compute discrete logarithms of even larger binary-field elliptic curves. Substantiated by practical experimentation, our results should be used by the community as basis for 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. Section 5 reflects on the best use of the negation map. The implementation of the most suitable iteration function is described in Section 6. As the design is flexible enough to attack larger elliptic curves, Section 7 gives runtime and cost approximations. Section 8 discusses future challenges and Section 9 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. They published 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. [22] 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.

The next larger Certicom challenge ECC2K-130 is 126 times more complex than ECC2-109. Therefore, researchers also computed discrete logarithms of custom elliptic curves. A discrete logarithm defined over a 112-bit prime-field elliptic curve was solved by Bos et al. [8], utilizing 200 PlayStations 3 for 6 months. A single PlayStation 3 reached a throughput of $42 \cdot 10^6$ iterations per second (IPS).

Several research teams investigated the potential speed of FPGAs to compute larger discrete logarithms. Dormale et al. [13] targeted ECC2-113, 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. [3], who attempt to break ECC2K-130 using Nvidia GTX 295 graphics cards, Intel Core 2 Extreme CPUs, Sony PlayStations 3, 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. [15] to process $111 \cdot 10^6$ IPS. Other FPGA architectures were proposed by Güneysu et al. [20], Mane et al. [25], and Judge et al. [24]. Güneysu et al.'s Spartan 3 architecture performs about $173 \cdot 10^3$ IPS, Mane et al.'s Virtex 5 architecture does $660 \cdot 10^3$ IPS, and Judge et al.'s Virtex 5 architecture executes around $14 \cdot 10^6$ IPS. In 2014, Engels [14] approximated that a discrete logarithm of the previously standardized [11] elliptic curve `sect113r1` can be computed in six months using 64 Spartan-6 FPGAs.

So far, none of their FPGA implementations have been successful in solving ECDLPs. This work on the other hand presents an architecture which has been used to successfully attack both a 113-bit Koblitz curve and the 113-bit binary-field Weierstrass curve `sect113r1`. The architecture performs $900 \cdot 10^6$ IPS on one of the 10 Kintex-7 FPGAs used.

3 Mathematical Foundations

To ensure a common vocabulary, it is important to revisit some of the basics. For further details, we refer to Hankerson et al. [21] and Cohen et al. [12].

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 by the Weierstrass equation $E/K : y^2 + xy = x^3 + ax^2 + b$, 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 such that $\sigma^\ell(P) = \lambda^\ell P \; \forall \; \ell$. 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

The security of ECC relies on 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 [29], nor the Weil and Tate pairing attacks [16,27], nor the Weil descent attack [18] apply. A standard algorithm to compute a discrete logarithm is Pollard's rho algorithm [30] or its parallelized version by van Oorschot and Wiener [33].

These algorithms are based on an iteration function f that defines a random cyclic walk over a graph. An iteration function updates 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.

Using either Pollard's rho or van Oorschot and Wiener's algorithm, the goal is to find a pair of colliding triples. 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 runs in a single tread, uses Floyd's cycle-finding algorithm and expects to encounter a collision after $\sqrt{\frac{\pi n}{2}}$ steps.

In order to parallelize an attack efficiently, van Oorschot and Wiener [33] introduced an algorithm based on 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 and can be computed in parallel. The achievable speedup is linearly proportional to the number of instances running in parallel. 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 optimized for performance in hardware, it is crucial to evaluate different iteration functions and select the most suitable one. In this work, the iteration functions by Teske [32], Wiener and Zuccherato [36], Gallant et al. [17], and Bailey et al. [2] were checked for their practical requirements and achievable computation rates. Table 1 summarizes the experiments done in software on a 41-bit Koblitz curve.

Teske's *r-adding walk* [32] 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 cannot only be used to speed up 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 [36], Gallant et al. [17], and Bailey et al. [2] proposed iteration functions which should achieve this $\sqrt{m}$-speedup.

Wiener and Zuccherato [36] proposed to calculate $f(X_i) = \sigma^{\ell}(X_i + R[j]) \; \forall \; \ell \in [0, m-1]$ and choose the point X, which has the smallest x-coordinate when interpreted as an integer. Gallant et al. [17] 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 $f(X_i) = X_i + \sigma^{\ell}(X_i)$, where $\ell = \mathrm{hash}_m(\mathcal{L}(X_i))$. Bailey et al. [2] suggested to compute $f(X_i) = X_i + \sigma^{(\ell \bmod 16)/2+3}(X_i)$ to reduce the complexity of the iteration function.

In order to investigate the practical differences of the iteration functions, a 41-bit Koblitz curve was used to evaluate them 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 identity function was used as hash function. 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 et al.'s and Bailey et al.'s iteration functions are 13 % 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 handling the problem of fruitless threads would increase the average runtime of Gallant et al.'s iteration function further.

Table 1 Expected and simulated total number of iterations to compute the discrete logarithm of a 41-bit Koblitz curve.

Reference	Iteration function	Expected iterations	Measured iterations
Teske [32]	$f(X_i) = X_i + R[j]$	$929 \cdot 10^3$	$906 \cdot 10^3$
Wiener and Zuccherato [36]	$f(X_i) = \min_{0 \leq l < m} \{\sigma^l(X_i + R[j])\}$	$145 \cdot 10^3$	$147 \cdot 10^3$
Gallant et al. [17]	$f(X_i) = X_i + \sigma^l(X_i)$	$145 \cdot 10^3$	$166 \cdot 10^3$
Bailey et al. [2]	$f(X_i) = X_i + \sigma^{(l \bmod 16)/2+3}(X_i)$	$145 \cdot 10^3$	$166 \cdot 10^3$

As Wiener and Zuccherato's iteration function achieved the best speed and does not have the problem of fruitless threads, it we selected it 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 Handling the Negation Map

In addition to the Frobenius automorphism, it is possible to use a negation map. The negation map compares X_i with $-X_i$ and selects the point with the smaller y-coordinate when interpreted as an integer. Consequently, the expected runtime of the parallelized Pollard's rho algorithm improves by a factor of $\sqrt{2}$. The drawback of the negation map is the high probability of fruitless cycles. A fruitless cycle happens if consecutive applications of the iteration function f results in the original triple $X_i = X_{i+L} = f^L(X_i)$. The collision of X_i with X_{i+L} is fruitless, cannot be used to compute the discrete logarithm, and the cycle cannot be left with a repeated application of f.

Figure 1 (a) depicts a normal iteration of points, where each node represents a computed triple. Figure 1 (b) and (c) show a fruitless 2-cycle and a fruitless 4-cycle, respectively. A 2-cycle happens if two applications of the iteration function map to the original point: $f(f(X_i)) = X_{i+2} = -(-(X_i + R[j]) + R[j]) = X_i$. This happens with a probability of $\frac{1}{2rm}$, with r being the size of the branching table $R[]$. Larger fruitless cycles happen with smaller probability.

When the negation map is used, it is essential to cope with fruitless cycles as these cycles render a thread useless. There are multiple ways to cope with a cycle once it is detected (cf. [7,9]). Table 2 summarizes several experiments done with a 30-bit prime curve (repeated 10,000 times). The iteration function was $f(X_i) = X_i + R[j]$ and the size of the branching table was $r = 16$. For reference, also an experiment without negation map was performed. The average runtimes are highly dependent on the particular method used, once a fruitless cycle is detected. Choosing a random branching index or a new random triple actually deteriorates the performance. Only the point doubling approach and the finally chosen method give the expected speed-up factor of 1.32 (similar to the speed-up reported by Bos et al. [9]). The advantage of the latter method is that no on-chip point doubling circuit is necessary (which is a huge advantage when a hardware design is done).

Table 2 Average number of iterations to compute the discrete logarithm of a 30-bit elliptic curve in dependence of how a fruitless cycle is handled.

Method	Iterations		Cycles
	Average	Stddev	Average
No negation map	33,117	21,453	0
New random triple	55,636	28,326	1,683
Random index	54,483	28,031	1,882
Point doubling	26,385	15,650	787
Determ. index $r = 16$	26,141	15,352	848
Determ. index $r = 128$	25,047	15,151	98

Our method works by deterministically choosing the index of the branching table based on the point X_i and the size of the largest detected cycle L: $f(X_i) = X_i + R[j+L]$. As depicted in Figure 1 (d) and (e), initially, a branch j resulting in a cycle is taken. Once the cycle is detected, a different branch $j+2$ or $j+4$ is selected. If the branch $j+2$ results in another 2-cycle, the branch $j+4$ is chosen (see Figure 1 (d)).

For the hardware design that was used to compute the discrete logarithm of `sect113r1`, a combination of methods was used to minimize the problem of fruitless cycles. (i) The branching index is deterministically chosen as described above. (ii) A branching table with $r = 1024$ entries minimizes the probability of loops. (iii) Block-RAM-based FIFOs are used to detect up to 10-cycles. Larger cycles are not detected, but they are very unlikely to occur in practice. (iv) Just in case that the hardware runs into a larger fruitless cycle, the hardware is restarted every 24 hours.

6 ECC Breaker Hardware

Based on these investigations on iteration functions, a hardware architecture was designed. This hardware architecture is based on the following assumptions.

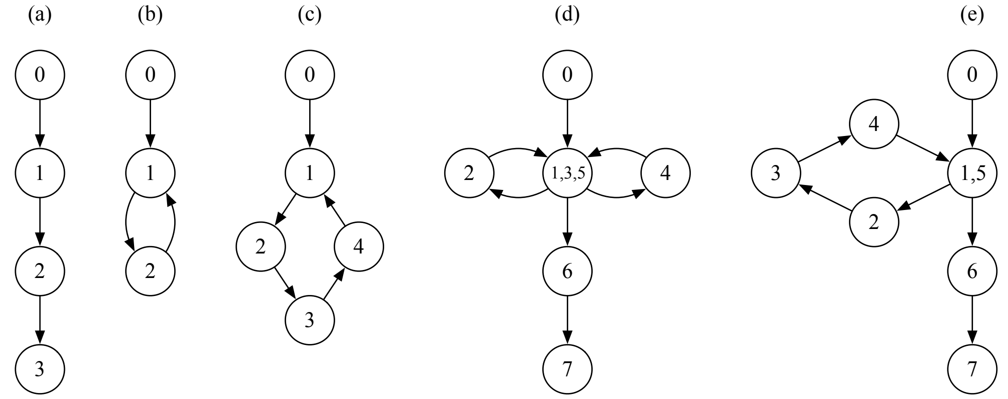

Fig. 1 Sequence of iteration functions: (a) normal sequence. (b) fruitless 2-cycle. (c) fruitless 4-cycle. (d) two fruitless 2-cycles that end in normal sequence. (e) fruitless 4-cycle that ends in normal sequence.

6.1 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 objective. Some authors even dare to compare FPGA and ASIC results. However, several of the largest components in ASIC designs, e.g., registers, RAMs, 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 and Kintex-7 evaluation boards were chosen as development platform. Note that all following design decisions were made to maximize the performance of ECC Breaker on these particular boards.

Design Goals. As Pollard's rho algorithm is perfectly parallelizable, the design goal clearly is to maximize the throughput per (given) area. Note that the speed (iterations per second) 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, we considered many area-efficient architectures, each coming with a single $\mathbb{F}_{2^m}$ multiplier, a $\mathbb{F}_{2^m}$ squarer, and a $\mathbb{F}_{2^m}$ 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 number of triples processed within ECC Breaker.* Therefore, the hardware is fully utilized in every cycle.

ECC Breaker versus Related Work. (i) In the current setup, the interface between ECC Breaker and a desktop is a simple, slow, serial interface. This might be a challenge for related implementations, but not for ECC Breaker. The implemented design detects fruitless cycles on-chip and the on-chip distinguished points (triple) storage assures that only distinguished triples have to be read. (ii) Unlike Fan *et al.* [15], our simultaneous inversion design is not iterative but fully unrolled. Therefore, our implementation is significantly larger, but also faster. (iii) Further, ECC Breaker comes with prime field $\mathbb{F}_n$ arithmetic which has only a minor impact on the size of the hardware. It proved indispensable during development that the generated distinguished triples could be easily verified. (iv) Variations of ECC Breaker were used to solve both the discrete logarithm of a 113-bit Koblitz curve [35] and the discrete logarithm of the elliptic curve `sect113r1`, which was part of a previous elliptic curve standard [11].

Generalization of ECC Breaker. Although the current version of ECC Breaker is carefully optimized for a 113-bit binary-field elliptic curve, the underlying architecture and design approach is also suitable for larger elliptic curves, e.g, a 131-bit Koblitz curve. In Section 7, approximations of the expected runtimes and potential costs to attack larger elliptic curves are given.

6.2 The Architecture

The basic architecture of ECC Breaker is presented in Figure 2. 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`

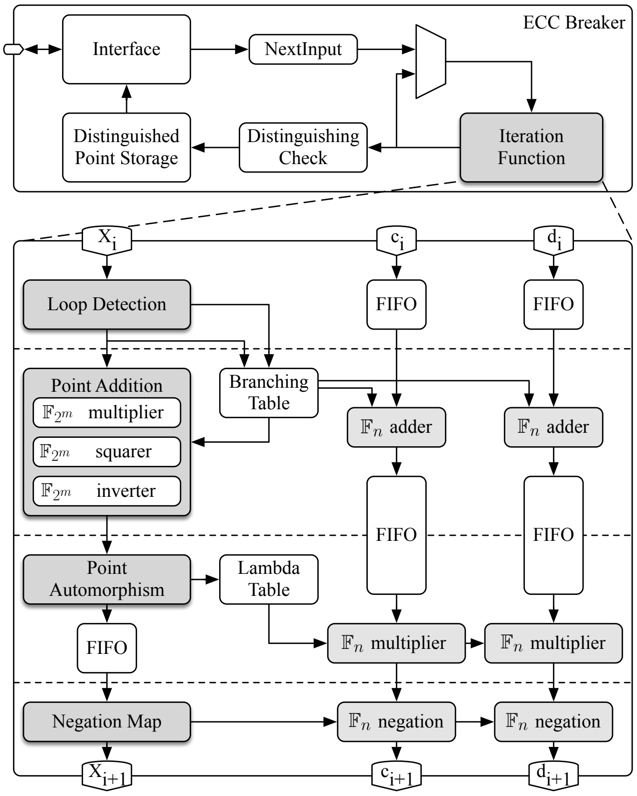

Fig. 2 Top-level architecture view. ECC Breaker is shown on top. The modularized iteration function is shown below.

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 were collected within the storage.

The iteration function itself consists of four major components: a point addition module, a point automorphism module, a negation map, and a loop detection module. Other components deal with $\mathbb{F}_{2^m}$ and $\mathbb{F}_n$ arithmetic, or are block-RAM-based tables and FIFOs. Because of this modularity it is easily possible, e.g., to use the point automorphism module only when a Koblitz curve is attacked. The components are described in the following.

6.3 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 $y_3 = \mu \cdot (x_1 + x_3) + x_3 + y_1$, with $\mu = (y_1 + y_2)/(x_1 + x_2)$. 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.

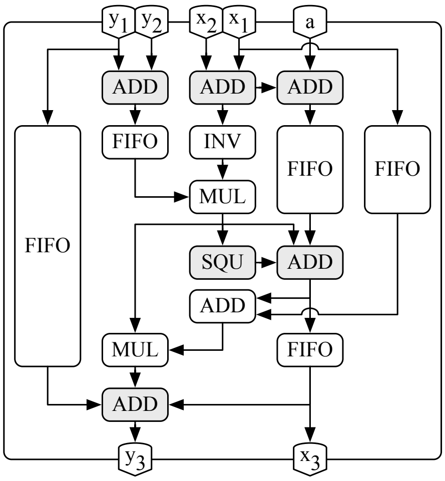

Fig. 3 Simplified point addition module. The grey shaded blocks are without registers.

Figure 3 shows the implemented point addition module which directly maps the formulas from above. Two $\mathbb{F}_{2^m}$ multipliers, 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. Dormale et al. [13]). However, it is possible to share the inversion module across multiple ECC Breaker instances at the cost of additional multipliers by taking advantage of the simultaneous inversion technique introduced by Montgomery [28]. If this is done, the latency of the point addition module increases. However, this has no impact on the overall throughput given the fully pipelined design.

6.4 $\mathbb{F}_{2^m}$ Inverse

The runtime of an Euclidean-based inversion algorithm is data-dependent 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 the technique by Itoh and Tsujii's [23], needing 112 squarers and 8 multipliers for $m = 113$: $a = a^{2^1-1} \to a^{2^2-1} \to a^{2^3-1} \to a^{2^6-1} \to a^{2^7-1} \to a^{2^{14}-1} \to a^{2^{28}-1} \to a^{2^{56}-1} \to a^{2^{112}-1} \to a^{2^{113}-2} = a^{-1}$.

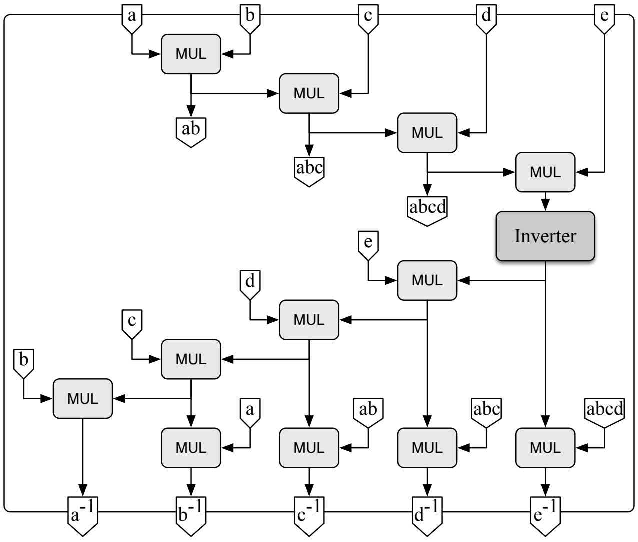

Fig. 4 Simultaneous inversion of 5 finite-field elements. The FIFOs that are needed for synchronization are not shown.

6.5 Simultaneous Inversion

With 8 multipliers, the inversion module is roughly 4 times larger than the rest of the point addition module which only requires 2 additional multipliers. Based on the simultaneous inversion technique [28], the inverter can be shared across multiple ECC Breaker instances at the rough cost of $3(d-1)$ multipliers for d inputs to invert. Figure 4 shows the use of 12 multipliers to invert 5 finite-field elements. Several FIFOs, which are not depicted, are used to deal with data-dependencies. Additional hardware is needed to deal with uninitialized ECC Breaker instances that have the number zero in the pipeline.

6.6 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 [36] 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 5. 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.

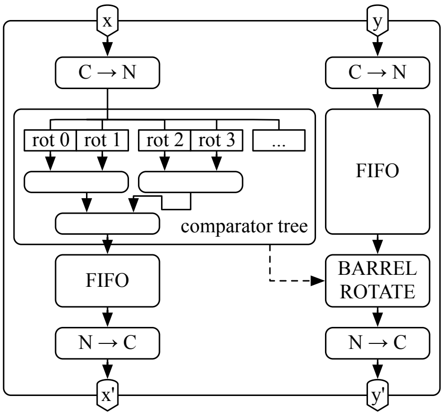

Fig. 5 Point automorphism unit with m comparator units.

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 $m = 113$.

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.

6.7 $\mathbb{F}_{2^m}$ 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 is 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 our previous results [35], 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 $\mathbb{F}_{2^m}$ inversion. Doing two basis transformations and a rotation within normal basis would probably save area. Also, accumulating the two transformation matrices into a single matrix would further reduce the area. However, as all squarers together only need 3% of all LUTs, the potential area improvement is rather limited. Therefore, contrary to related attempts [3,15], ECC Breaker only uses a normal basis number representation within the point automorphism module.

6.8 $\mathbb{F}_{2^m}$ Multiplier

The $\mathbb{F}_{2^m}$ multipliers have the largest impact on the area footprint of the ECC Breaker design. For ECC Breaker, the following multiplier designs were evaluated using a Virtex-6 FPGA (post-synthesis): (i) A simple 113-bit parallel polynomial multiplier needs 5,497 LUTs. (ii) A Mastrovito multiplier [26] 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. (iii) Bernstein [5] combines some refined Karatsuba and Toom recursions for his batch binary Edwards multiplier. His code [6] 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 [31]. 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. Finally the design was equipped with several pipeline stages such that it can be clocked with high frequencies.

6.9 $\mathbb{F}_n$ Multiplier

Computing prime-field multiplications in hardware can be a troublesome and very resource-intensive task. Dedicated DSP slices were used for integer multiplications. As a result, the two $\mathbb{F}_n$ multipliers are very resource efficient, requiring very few LUTs and only 2×145 DSP slices.

7 Results and Transferability of Results

The construction of the ECC Breaker design was an iterative process in which the speed, the area, and the power characteristics were continuously optimized. To exploit all available resources, the available block RAMs and DSP slices were used whenever possible. The design that was used to compute the discrete logarithm of `sect113r1` was optimized for Kintex-7 FPGAs (KC705 development boards coming with XC7K325T-2 FPGAs). The best stable performance was achieved with 5 ECC Breaker instances running at 180 MHz. With additional ventilation, 10 Kintex-7 boards operated stably at an operating temperature of around 90° C (194° F). Both increasing the number of instances (and reducing the clock frequency) or increasing the clock frequency (and reducing the number of instances) either deteriorated the performance or was not routable.

Table 3 Post place-and-route Kintex-7 utilization with 5 ECC Breaker instances.

Module	Instances	Slices p. Ins.	Slices Total	FPGA Util.
top			40,915	80%
ECC Breaker	5	4,203	21,016	41%
iteration function	5	3,703	18,517	36%
point addition	5	2,236	11,182	22%
$\mathbb{F}_{2^m}$ multiplier	5×2	962	9,624	19%
simul. inversion			19,836	39%
$\mathbb{F}_{2^m}$ multiplier	4×3	877	10,523	21%
$\mathbb{F}_{2^m}$ inverter	1	9,022	9,022	18%
$\mathbb{F}_{2^m}$ multiplier	8	904	7,232	14%
$\mathbb{F}_{2^m}$ squarer	112	14.8	1,653	3%

By using Xilinx ISE 14.7 as toolchain, the following utilizations were achieved. ECC Breaker requires (post place-and-route) 80% of all available slices (40,915/50,950), 73% of all LUTs (150,750/203,800), 41% of all registers (170,799/407,600), and 31% of all block RAMs (408/1,335). Table 3 gives the number of slices needed for all components. The biggest components are the 5 ECC Breaker instances and the simultaneous inversion module. 67% of the slices are needed for $\mathbb{F}_{2^m}$ multipliers (5×2 within the point addition modules, 12 for the simultaneous inverter, and 8 for the inverter itself). As the place-and-route tool performs optimizations across module borders, the slice counts of all components are just approximations by the mapping tool.

7.1 Extrapolating the Results

The results above are just a snapshot of a much larger picture. Based on the current VHDL design, one could optimize the design for different FPGAs, different elliptic curves, or for ASICs. The ECC Breaker design that was used to attack `sect113r1` processes $5 \times 180 = 900$ million iterations per second and 77,760,000,000,000 iterations per day. Assuming that the same performance can be reached for larger elliptic curves as well (using the same FPGA), the budgets to break them within a year are shown in Table 4.

This budgets does not include the man-power needed for development or upkeep, or the electrical en-

Table 4 Approximations of costs to compute large discrete logarithms within a year. Budget is in USD.

Elliptic curve	Standard	Group size	Iterations		FPGA days	#FPGAs	FPGA Budget
113-bit Koblitz		2^{112}	$10^{15.8}$	$2^{52.4}$	77	1	1,700
113-bit Weierstrass	`sect113r1`	2^{112}	$10^{16.8}$	$2^{55.8}$	821	3	5,100
127-bit Koblitz		2^{114}	$10^{16.1}$	$2^{53.4}$	156	1	1,700
127-bit Weierstrass		2^{125}	$10^{18.8}$	$2^{62.3}$	74,330	204	346,800
131-bit Koblitz		2^{129}	$10^{18.3}$	$2^{60.8}$	25,977	72	122,400
131-bit Weierstrass	`sect131r1`	2^{129}	$10^{19.4}$	$2^{64.3}$	297,320	815	1,385,500
163-bit Koblitz	`sect163k1`	2^{162}	$10^{23.2}$	$2^{77.2}$	$2.16 \cdot 10^{9}$	$5.19 \cdot 10^{6}$	$10.1 \cdot 10^{9}$
163-bit Weierstrass	`sect163r2`	2^{162}	$10^{24.3}$	$2^{80.8}$	$27.6 \cdot 10^{9}$	$75.5 \cdot 10^{6}$	$128 \cdot 10^{9}$
256-bit Weierstrass	`secp256r1`	2^{256}	$10^{38.5}$	$2^{127.8}$	$3.89 \cdot 10^{24}$	$10.6 \cdot 10^{21}$	$18.1 \cdot 10^{24}$

ergy needed to run the FPGAs. Additionally, there is a lot of room for improvement to reduce the necessary budgets. It is possible to increase the performance of the hardware design, or to reduce the costs of the FPGAs. If the number of necessary FPGAs reaches the millions, it is probably more cost-efficient to build a dedicated ASIC design to compute discrete logarithms more efficiently. An ASIC design potentially provides a better cost-performance-ratio, but the development process is potentially much more time consuming.

With that in mind, it has to be emphasized that Table 4 summarizes what we could do *now* using KC705 development boards, which cost around USD 1,700 in the beginning of 2015 [37]. The expected iteration count includes the potential speed-ups of both negation maps and group automorphisms when applicable. Using our current setup of 10 KC705 development boards, it is possible to compute the discrete logarithms of 113-bit Koblitz and 113-bit Weierstrass curves in around 8 and 82 days, respectively. Using 72 KC705s, it would be possible to solve the discrete logarithm of a 131-bit Koblitz curve in a year, one of the official Certicom challenges [10].

At the 80-bit security level, the necessary budget to break an elliptic curve is around 10-100 billion USD (for FPGAs). This seems like an extraordinary amount of money, but under the assumption that there is a fair amount of optimization potential and that there are some organizations with huge funds, elliptic curves at the 80-bit security-level should not be used any more. However, nobody (c.f. [19]) recommends long-term use of 160-bit elliptic curves anyways.

At the 128-bit security level, budgets in the range of $18.1 \cdot 10^{24}$ USD give elliptic curves, such as `secp256r1`, a sufficient cushion to be safe for the next decades. Assuming that every year the necessary budget halves (which it probably will not), an elliptic curve at the 128-bit security level will be secure for the next 40-50 years; unless there is an algorithmic breakthrough, a breakthrough with quantum computers, or a backdoor in the elliptic curve standard.

8 Future Challenges

Solving cryptographic challenges is a process, in which every optimization step results in a potentially better design. In order to support other researchers, all our code is available online [34]. There are still plenty of challenges to be investigated:

It is possible to improve the *performance* by reducing the critical path or by shrinking the size of ECC Breaker. Especially a smaller finite field multiplier would enable to place more ECC Breakers per FPGA. However, ECC Breaker is a fairly complex and large design. The hardware synthesizer reached its limit when it came to maximum frequency approximations. In most cases, it was only possible to reach a fraction of the theoretically given frequency after mapping and routing.

An additional design dimension is the *power consumption*. Every pipeline stage within ECC Breaker is active in every cycle, and therefore every utilized slice is active in every cycle. Both, the power supply and cooling system, which is responsible for dissipating the heat, run at full capacity. For the attack on Koblitz curves [35], where we used ML605 boards, it was necessary to reduce the clock frequency to 165 MHz even though the synthesizer approximated a maximum clock frequency of 275 MHz.

This paper demonstrates that FPGAs are well suited to compute discrete logarithms of elliptic curves defined over binary extension fields. It has yet to be answered how well FPGAs can be used to attack *elliptic curves* defined over *prime fields*. Assuming that a Mersenne-like prime that enables fast reduction is used, then roughly 64 DSP slices are necessary for a 128-bit prime-field multiplier. Consequently, 13 finite-field multiplier would fit within the 840 DSP slices of a KC705. Assuming the inversion is built using only general purpose slices, then 2–3 instances would fit per FPGA.

Ultimately, at some point it makes sense to develop an *ASIC*. An ASIC does not come with 'free' DSP slices or 'free' block-RAMs. While the basic architecture of ECC Breaker can be reused, it would be necessary to re-evaluate some components: (i) It would be worth investigating the best finite-field multiplier design for ASICs. (ii) The block RAMs would have to be replaced by RAM macros. However, RAM macros are designed to host a lot of entries and not to access hundreds of bits at once. Therefore, it is questionable whether RAM macros are actually smaller than register-based RAMs. (iii) To some degree, an ASIC can be built arbitrarily large, but the larger it is, the more serious the power issues are. The optimal number of ECC Breaker instances that share a common inversion module has to be evaluated. (iv) Different manufacturing technologies enable different clock frequencies and have different costs per gate equivalent. (v) Finally, not only an ASIC but also a printed circuit board to host the ASIC have to be designed, implemented, and tested – a process that can potentially require several man-years.

9 Conclusion

We have shown the potential of FPGAs for solving elliptic curve discrete logarithm problems. We solved both a discrete logarithm of a 113-bit Koblitz curve [35] and a discrete logarithm of the elliptic curve `sect113r1` (see Appendix A), a 113-bit Weierstrass curve based on binary fields.

Our ECC Breaker design performs 900 million iterations per second on an off-the-shelf Kintex-7 FPGA. It distinguishes itself with good performance and little communication overhead. We invite fellow researchers to use our code [34] to adapt and optimize it for larger elliptic curves, and to use it to compute even more complex discrete logarithms over elliptic curves.

10 Acknowledgments

The authors are grateful to the University of Applied Sciences Upper Austria who provided 16 ML605 boards, the companies so-logic GmbH Co KG and Xilinx, Inc. who provided us with several Kintex-7 FPGAs, and colleagues who recommended to take advantage of the simultaneous inversion technique.

This work has been supported 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. 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.
2. 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.
3. 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.
4. E. Barker and A. Roginsky. Recommendation for Cryptographic Key Generation. *NIST Special Publication*, 800:133, 2012.
5. D. J. Bernstein. Batch Binary Edwards. In *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *LNCS*, pages 317–336. Springer, 2009.
6. D. J. Bernstein. Binary Batch Edwards 113-bit Multiplier. Accessed online in Oct. 2013 at `http://binary.cr.yp.to/bbe251/113.gz`, May 2009.
7. D. J. Bernstein, T. Lange, and P. Schwabe. On the Correct Use of the Negation Map in the Pollard rho Method. In *Public Key Cryptography - PKC 2011*, volume 6571 of *LNCS*, pages 128–146. Springer, 2011.
8. 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):212–228, 2012.
9. J. W. Bos, T. Kleinjung, and A. K. Lenstra. On the Use of the Negation Map in the Pollard Rho Method. In *Algorithmic Number Theory - ANTS-IX*, volume 6197 of *LNCS*, pages 66–82. Springer, 2010.
10. Certicom Research. The Certicom ECC Challenge. Available online at `https://www.certicom.com/index.php/the-certicom-ecc-challenge`, Nov 1997.
11. Certicom Research. Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography, Version 1.0. Available online at `http://www.secg.org/`, Sep 2000.
12. 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.
13. G. M. de Dormale, P. Bulens, and J.-J. Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *Cryptographic Hardware and Embedded Systems - CHES*, LNCS, pages 378–393. Springer, 2007.
14. S. Engels. Breaking ecc2-113: Efficient Implementation of an Optimized Attack on a Reconfigurable Hardware

Cluster. Master's thesis, Ruhr Universityät Bochum, Feb 2014.

15. 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.
16. 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.
17. 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.
18. 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.
19. D. Giry. BlueKrypt - v28.4 - Cryptographic Key Length Recommendation. Accessed online in Feb. 2015 at `http://www.keylength.com/en/`.
20. T. Güneysu, C. Paar, and J. Pelzl. Attacking Elliptic Curve Cryptosystems with Special-Purpose Hardware. In *ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)*, pages 207–215. ACM Press, 2007.
21. D. Hankerson, S. Vanstone, and A. J. Menezes. *Guide to Elliptic Curve Cryptography.* Springer, 2004.
22. R. Harley. Elliptic Curve Discrete Logarithms: ECC2K-108, 2000. Available online at `http://cristal.inria.fr/~harley/ecdl7/readMe.html`.
23. T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. *Information and Computation*, 78(3):171–177, 1988.
24. L. Judge, S. Mane, and P. Schaumont. A Hardware-Accelerated ECDLP with High-Performance Modular Multiplication. *International Journal of Reconfigurable Computing*, 2012, 2012.
25. 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 - ReConFig*, pages 198–203. IEEE, 2011.
26. 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. Springer, 1988.
27. 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.
28. P. L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, Jan 1987.
29. 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.
30. J. M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.
31. F. Rodríguez-Henríquez and Ç. Koç. On Fully Parallel Karatsuba Multipliers for GF(2^m). *Journal of Computer Science and Technology*, pages 405–410, 2003.
32. E. Teske. Speeding up Pollard's Rho Method for Computing Discrete Logarithms. In *Algorithmic Number Theory*, volume 1423 of *LNCS*, pages 541–554. Springer, 1998.
33. P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

Table 5 Curve parameters of targeted elliptic curve `sect113r1`.

m	113
irreducible polynomial	$x^{113} + x^9 + 1$
irreducible polynomial	`0x20000000000000000000000000201`
elliptic curve E	$y^2 + xy = x^3 + ax^2 + b$
curve parameter a	`0x3088250CA6E7C7FE649CE85820F7`
curve parameter b	`0xE8BEE4D3E2260744188BE0E9C723`
order n	`0x100000000000000D9CCEC8A39E56F`
cofactor h	2
point $P.x$	`0x1a89024c72cf8ea989c1f36bb960b`
point $P.y$	`0x15c3672f4d46a191965e39500e63d`
point $Q.x$	`0x13fb48ae2aaee444d7cbf744bbbc9`
point $Q.y$	`0x17024e9d2c3ba781bf9a5993cc232`
scalar k such that $Q = kP$	`0x8818aa79f0a6ec0eaef9bd414497`

34. E. Wenger and P. Wolfger. ECC Breaker Source Code. Accessed online in Feb. 2015 at `http://www.iaik.tugraz.at/content/research/opensource/ecc_breaker/`.
35. E. Wenger and P. Wolfger. Solving the Discrete Logarithm of a 113-Bit Koblitz Curve with an FPGA Cluster. In *Selected Areas in Cryptography - SAC*, volume 8781 of *LNCS*, pages 363–379. Springer, 2014.
36. M. J. Wiener and R. J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In *Selected Areas in Cryptography - SAC*, volume 1556 of *LNCS*, pages 190–200. Springer, 1999.
37. Xilinx Inc. Xilinx Kintex-7 FPGA KC705 Evaluation Kit. Accessed online in Feb. 2015 at `http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html`.

A Targeted Curve and Target Point Pair Selection

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 was used to deterministically and pseudo-randomly generate two points with order n using Sage. 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.

Listing 1 Sage code to verify P, Q, and $Q = kP$.

```
FF = sage.rings.finite_rings.finite_field_ext_pari.FiniteField_ext_pari;
m=113; h=2; n=0x100000000000000D9CCEC8A39E56F
K=FF(2**m, 'x', 0x20000000000000000000000000201.bits())
x=K.gen()

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))

k=0x8818aa79f0a6ec0eaef9bd414497
a=K(str_to_poly("3088250CA6E7C7FE649CE85820F7"))
b=K(str_to_poly("E8BEE4D3E2260744188BE0E9C723"))
E = EllipticCurve(K, [1,a,0,0,b])

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(2)).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()
```

Algorithm 1 Calculate $c = a \times b$, with a, b being 113-bit binary polynomials.

```
Require: $a, b$
Ensure: $c = a \times b$
$m_{ab1} \leftarrow (a[112..64] \oplus a[63..0]) \times (b[112..64] \oplus b[63..0])$   ▷ KS64
$c_{l1} \leftarrow a[63..0] \times b[63..0]$   ▷ KS64
$c_{l2} \leftarrow a[95..64] \times b[95..64]$   ▷ KS32
$c_{l3} \leftarrow a[111..96] \times b[111..96]$   ▷ KS16
$m_{ab2} \leftarrow (a[95..64] \oplus a[111..96]) \times (b[95..64] \oplus b[111..96])$   ▷ KS32

$m_{a3} \leftarrow b[112] \times a[111..96]$
$m_{b3} \leftarrow a[112] \times b[111..96]$
$m_3 \leftarrow m_{a3} \oplus m_{b3}$
$c_3[32] \leftarrow a[112] \times b[112]$
$c_3[30..0] \leftarrow c_{l3}$
$c_3[31..16] \leftarrow c_3[31..16] \oplus m_3$

$m_2 \leftarrow m_{ab2} \oplus c_{l2} \oplus c_3$
$c_2[62..0] \leftarrow c_{l2}$
$c_2[97..64] \leftarrow c_3$
$c_2[94..32] \leftarrow c_2[94..32] \oplus m_2$

$m_1 \leftarrow m_{ab1} \oplus c_{l1} \oplus c_2$
$c[126..0] \leftarrow c_{l1}$
$c[225..128] \leftarrow c_2$
$c[190..64] \leftarrow c[190..64] \oplus m_1$
```