Faster discrete logarithms on FPGAs

Daniel J. Bernstein[1,2], Susanne Engels[3], Tanja Lange[2], Ruben Niederhagen[2], Christof Paar[3], Peter Schwabe[4], and Ralf Zimmermann[3]

[1] Department of Computer Science
University of Illinois at Chicago, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
tanja@hyperelliptic.org, ruben@polycephaly.org
[3] Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
susanne.engels@rub.de, christof.paar@rub.de, ralf.zimmermann@rub.de
[4] Institute for Computing and Information Sciences
Radboud University, The Netherlands
peter@cryptojedi.org

Abstract. This paper accelerates FPGA computations of discrete logarithms on elliptic curves over binary fields. As an illustration, this paper reports successful completion of an attack against the SECG standard curve `sect113r2`, a binary elliptic curve that was not removed from the standard until 2010 and was not disabled in OpenSSL until June 2015. This is a new size record for ECDL computations, using a prime order very slightly larger than the previous record holder. More importantly, this paper uses FPGAs much more efficiently, saving a factor close to 3/2 in the size of each high-speed ECDL core and allowing 3 cores to be squeezed into a low-cost Spartan-6 FPGA. The paper also covers much larger curves over 127-bit fields.

Keywords: attacks, FPGAs, ECC, binary curves, Pollard rho, negation

1 Introduction

FPGAs are the most energy-efficient mass-market devices for computations of discrete logarithms on elliptic curves defined over binary fields. For example, in 2009 a large cross-platform effort was initiated [1] to attack Certicom's ECC2K-130 challenge; an implementation optimized for a 300-watt NVIDIA GTX 295 GPU (dual 55nm GT200) performed 63 million iterations per second [2], while

Public domain. This work was supported by NSF (U.S.) under grant 1018836; by NWO (Netherlands) under grants 639.073.005, 613.001.011, and Veni 2013 project 13114; by the European Commission through the ICT program under contract INFSO-ICT-284833 (PUFFIN). Permanent ID of this document: `01ac92080664fb3a778a430e028e55c8`. Date: 2016.04.14.

an implementation of the same iteration function optimized for a 5-watt Xilinx XC3S5000 (90nm Spartan-3) FPGA performed 111 million iterations per second [13], obviously much better performance.

An FPGA is a well-connected mesh of a large number of programmable "lookup tables" (LUTs) surrounded by other useful resources such as "registers". For example, an XC3S5000 contains 74880 "LUT-4" units; each LUT-4 maps 4 input bits to 1 output bit. This paper focuses on newer Xilinx FPGAs with larger "LUT-6" units, each mapping 6 input bits to 1 output bit: e.g.,

– a Xilinx XC6SLX150 (45nm Spartan-6, typically clocked at 100MHz) contains 92152 LUT-6; and
– a Xilinx XC7K325T-2 (28nm Kintex-7, typically clocked at 180MHz) contains 203800 LUT-6.

A modern high-end GPU is also highly parallel, with thousands of 32-bit arithmetic units, but most of the operations provided by these units (e.g., floating-point multiplication) are not helpful for binary-field arithmetic, and the remaining operations (e.g., xor) are quite wasteful, spending most of their energy and chip area on data transfer rather than computation. The LUTs in FPGAs handle binary-field computations with much less overhead.

Recent binary-field elliptic-curve discrete-log work by Wenger and Wolfger (see the preliminary SAC 2014 paper [26] and the final journal paper [27]) fits 5 cores onto an XC7K325T-2, using a total of 151 KLUTs, i.e., 30 KLUTs per core. Each core runs at 180MHz and computes 1 iteration per cycle, so in total the FPGA computes 900 million iterations per second. Wenger and Wolfger used a cluster of ten KC705 development boards, each with an XC7K325T-2 FPGA, for 2.5 months to successfully compute a discrete logarithm on an elliptic curve defined over the field $\mathbb{F}_{2^{113}}$. The prime order here was slightly above 2^{112}, not much larger than the prime order from the previous ECDL record (approximately $2^{111.78}$; see [6] and [7]), but the previous ECDL record used PlayStations and would have needed about 300 PlayStations to be completed in 2.5 months.

Wenger and Wolfger refer to several prior ECDL implementations on FPGAs but claim in [27] that "none of their FPGA implementations have been successful in solving ECDLPs". This is contradicted by, e.g., [27, reference 14, Section 5.4], which reported "successfully" breaking a "target with a 60-bit ECDLP". It was already clear that FPGAs are very efficient for this task; the remaining question is whether they can be sped up even more.

Primary contribution of this paper: more efficient ECDL cores. In this paper we do better than [27] by a factor around 3/2: we squeeze ECDL cores into just 21 KLUTs per core for the same field $\mathbb{F}_{2^{113}}$, while maintaining high clock frequencies and maintaining a speed of 1 iteration per cycle on each core. For example, we fit 6 cores into 126 KLUTs, 7 cores into 145 KLUTs, and 8 cores into 163 KLUTs. This speedup combines three directions of improvements:

– Smaller high-speed multipliers. Our $\mathbb{F}_{2^{113}}$ multiplier takes just 3071 LUTs. The multiplier in [27] takes 3757 LUTs, 22% larger.

– Fewer multipliers. For example, we use 16 multipliers for 3 cores and 32 multipliers for 6 cores, while the approach of [27] needs 15 multipliers for just 2 cores and 30 multipliers for just 5 cores.
– Reduced area outside the multipliers. The total number of LUTs we use is only about 30% more than the number used for multiplications. For [27] the overhead is around 50%.

We do not have access to any Kintex-7 FPGAs for testing, but we do have access to low-cost Spartan-6 FPGAs. We fit 3 cores into just 64 KLUTs and tested those cores at 100MHz on an XC6SLX150; this is a total of 300 million iterations per second, achieving 1/3 the speed of [27] using an FPGA that costs only 1/5 as much.

We estimate that scaling the design of [27] down to 1 core would make it fit into an XC6SLX150 and would successfully run at 100MHz, but this would compute only 100 million iterations per second. It is not at all clear that 2 cores would fit: 15 multipliers at 3757 LUTs would already consume 56355 LUTs even without counting overhead. It is clear that 3 cores would not fit. For our design 3 cores fit easily. On the more powerful Kintex-7 used in [27], we do not know whether our 8-core design would run stably at 180MHz, but we expect the 7-core design to run stably at 180MHz, computing 1.26 billion iterations per second on an XC7K325T-2, 40% more than [27].

See Sections 3 and 4 for further details of our design, and Section 6 for analysis of how our results improve upon the results of [27].

Secondary contribution of this paper: a new ECDL record. The curve `sect113r2` was standardized by SECG (Standards for Efficient Cryptography Group) in version 1.0 of "SEC 2: Recommended Elliptic Curve Domain Parameters" [9] in 2000, and was included as one of the supported curves when OpenSSL added ECC support in 2005 (version 0.9.8). This curve is defined over $\mathbb{F}_{2^{113}}$; see Section 2 for further details of the curve.

This curve disappeared from version 2.0 of the SEC 2 standard [10] in 2010 (along with all other curves over field sizes below $\approx 2^{192}$ for odd characteristic and 2^{163} for even characteristic), and disappeared from OpenSSL's default curve list in June 2015 (version 1.0.2b). However, most installations are running older versions of OpenSSL that still support this curve. It is easy to imagine how an OpenSSL user seeking to minimize bandwidth for ECC would look through the output of `openssl ecparam -list_curves` and find this curve as one of the lowest-bandwidth options.

We used 120 XC6SLX150 FPGAs to successfully compute an ECDL of a random target point on `sect113r2`. This computation was slower than desired for three reasons: first, it used a preliminary 2-core version of our implementation; second, not all of the FPGAs were available all of the time; third, the number of iterations in these ECDL algorithms is a random variable with high variance, and we were moderately unlucky in the number of iterations used for this particular computation. The computation nevertheless finished in under 2 months.

Technically, this is a new ECDL record, since the prime order is (marginally) larger than the prime order in [27]. We do not mean to exaggerate the importance of setting ECDL size records; obviously such records are heavily influenced by hardware availability, obscuring the impact of algorithmic improvements and understating the amount of hardware actually available to attackers. We also do not mean to exaggerate the importance of this curve: we do not know how many people have actually used `sect113r2` to encrypt data. What really matters in this paper is being able to squeeze iterations into fewer LUTs, reducing costs not merely for this attack but also for much larger attacks against much larger curves.

Variations and extrapolations. We scaled our design from a 113-bit field up to a 127-bit field. This expanded 3 cores from 64401 LUTs to 74095 LUTs. We tested that this 3-core design still fits onto an XC6SLX150 and runs successfully at 100MHz. We are now running this design on 128 of these FPGAs to compute an ECDL on a curve over $\mathbb{F}_{2^{127}}$ in a subgroup of prime order approximately $2^{117.35}$; this is expected to take only 123 days.

Many components of an ECDL core grow linearly with the number of bits, and the area for a high-speed multiplier grows superlinearly. There is also a noticeable extra reduction cost for fields defined by pentanomials rather than trinomials, such as $\mathbb{F}_{2^{163}}$. Scaling to larger and larger fields will eventually force any particular size of FPGA to use fewer and fewer cores. Beware that the extrapolations in [27, Section 7.1 and Table 4] assume that the number of FPGA-years scales as a simple square root of the prime order, without regard to the increasing cost of arithmetic and the resulting loss of cores.

Fortunately for the attacker, our speedups compensate for these caveats considerably beyond 127 bits. Furthermore, given the agility of FPGAs to promptly and cost-effectively tackle new problems, any serious attacker should be expected to be operating a large FPGA cluster; and, given economies of scale, the cost per FPGA in a large cluster should be expected to be much lower than indicated in [27, Table 4]. We do not agree, for example, that a 5-million-FPGA cluster for breaking a 163-bit Koblitz curve in a year would cost $10 \cdot 10^9$ USD, an entire year of NSA's budget. A more plausible estimate is under $2 \cdot 10^9$ USD, similar in cost (and power consumption) to one of NSA's existing data centers. Note also that standard batch techniques break subsequent keys more cheaply, bringing additional benefit to the attacker; see [17], [16], and [3].

A well-funded attacker facing years of predictable large-scale computations will do even better by building application-specific integrated circuits (ASICs). FPGA optimization techniques are well known to be much better than CPU (and GPU) optimization techniques as a predictor of ASIC optimization techniques. Our multiplier details should be reoptimized for ASICs but we expect the overall architecture to perform very well.

Binary fields vs. prime fields. The standard NIST curves fall into three different categories: Koblitz curves over binary fields $\mathbb{F}_{2^n}$, "random" curves over binary fields $\mathbb{F}_{2^n}$, and "random" curves over prime fields $\mathbb{F}_p$. There are five NIST curves in each category, spread across five different sizes of 2^n or p. The smallest 2^n is 2^{163}, while the smallest p is approximately 2^{192}. See [18] and [19].

It is easy to see how an ASIC designer concerned with the costs of an ECC coprocessor (chip area, power, energy, etc.) for constructive use will end up choosing $\mathbb{F}_{2^{163}}$. Taking a binary field rather than a prime field eliminates all the circuitry for carries, and taking the smallest allowable field has obvious performance benefits. It is not as clear whether the designer will prefer a Koblitz curve or a "random" curve: the extra endomorphisms in Koblitz curves reduce the number of field multiplications inside scalar multiplication, saving energy, but managing these endomorphisms comes at a cost in chip area.

There are several common arguments that prime fields should be preferred, but it is also easy to imagine counterarguments from the ASIC designer:

– Prime fields provide better software performance, since they take better advantage of the integer multipliers provided by CPUs. Counterarguments: CPUs evolve to meet the needs of applications, and Intel's new `PCLMULQDQ` instruction already provides excellent performance for curves over binary fields; see [21]. Software performance is ultimately less important than hardware performance.
– Prime fields are the safest choice, since binary fields have extra structure that might be exploitable. Some recent papers have suggested the possibility of an asymptotically subexponential ECDL algorithm for curves over $\mathbb{F}_{2^n}$. Counterarguments: Other recent papers have disputed this possibility. None of the papers have claimed relevance to the range of n actually used in ECC. Speculations about security problems are less important than meeting the performance requirements of the applications.
– For Koblitz curves there are extra endomorphisms that speed up known attacks by a factor close to $\sqrt{n}$. Counterarguments: This speedup does not apply to "random" curves, and $\sqrt{n}$ is a limited factor in any case.
– Some ECC standards require prime fields: consider, e.g., NSA's Suite-B [20] and the Brainpool standard [12], both from 2005. Counterargument: Those standards do not articulate reasons to avoid binary fields: e.g., [12] says that subsequent editions "may also contain elliptic curves over fields of characteristic 2".

In this paper we do not take a position in this debate. We merely observe that the performance of binary-field ECC continues to attract attention, so the community also needs to understand the cost of solving binary-field ECDLP. Standard extrapolations (see above) suggest that breaking a "random" curve over $\mathbb{F}_{2^{163}}$ is an order of magnitude more expensive than breaking a Koblitz curve over $\mathbb{F}_{2^{163}}$, which in turn is millions of times more expensive than breaking a random curve over $\mathbb{F}_{2^{113}}$, but these are not infeasible computations. Some of our area-optimization techniques are also applicable to prime-field ECDLP, although obviously the details of arithmetic will be different.

Acknowldegements. The authors would like to thank Bo-Yin Yang of Academia Sinica for providing access to his FPGA clusters for all computations and to Sci-Engines for doing computations on their clusters for the 117-bit DLP.

2 A typical target curve: `sect113r2`

The SECG curve `sect113r2` is defined over $\mathbb{F}_{2^{113}} \cong \mathbb{F}_2[w]/(w^{113}+w^9+1)$ by an equation of the form $E : y^2 + xy = x^3 + ax^2 + b$ and basepoint $P = (x_P, y_P)$, where

$$\begin{aligned} a &= \texttt{0x0689918DBEC7E5A0DD6DFC0AA55C7},\\ b &= \texttt{0x095E9A9EC9B297BD4BF36E059184F},\\ x_P &= \texttt{0x1A57A6A7B26CA5EF52FCDB8164797}, \text{ and}\\ y_P &= \texttt{0x0B3ADC94ED1FE674C06E695BABA1D}. \end{aligned}$$

using hexadecimal representation for elements of $\mathbb{F}_{2^{113}}$, i.e., taking the coefficients in the binary representation of the integer as coefficients of the powers of w, with the least significant bit corresponding to the power of w^0. The order of P is $\ell = 5192296858534827702972497909952403$, which is prime. The order of the curve $|E(\mathbb{F}_{2^{113}})|$ equals 2ℓ.

It is possible to transform the elliptic curve to isomorphic ones by maps of the form $x' = c^2x + u, y' = c^3y + dx + v$. These maps do not change the general shape of the curve (the highest terms are still y^2, x^3, and xy) but allow mapping to more efficient representations. The security among isomorphic curves is identical: the DLP can be transformed using the same equations. Curve arithmetic depends on the value of a and for fields of odd extension degree it is always possible to find an isomorphic curve with $a \in \{0, 1\}$. It is unclear why this optimization was not applied in SECG but we will use it in the cryptanalysis.

For `sect113r2` we have $\mathrm{Tr}(a) = 1$ so there is an element $t \in \mathbb{F}_{2^{113}}$ satisfying $t^2 + t + a + 1 = 0$. Now $(x_P, y_P + tx_P)$ is on $y^2 + xy = x^3 + x^2 + b$ for every (x_P, y_P) on E because

$$\begin{aligned} (y_P + tx_P)^2 + x_P(y_P + tx_P) &= y_P^2 + x_Py_P + (t^2x_P^2 + tx_P^2)\\ &= x_P^3 + ax_P^2 + b + (t^2+t)x_P^2 = x_P^3 + x_P^2 + b. \end{aligned}$$

The specific value for t is given in Appendix B. The base point is transformed to (x_P, y'_P) with

$$y'_P = \texttt{0x17D5618CD2EE81F84FAB74B1EB19F}.$$

Addition of two points (x_1, y_1) and (x_2, y_2) on this curve $y^2+xy = x^3+x^2+b$ produces a result (x_3, y_3) with

$$\begin{aligned} (x_3, y_3) &= (\lambda^2 + \lambda + 1 + x_1 + x_2, \quad \lambda(x_1 + x_3) + y_1 + x_3), \text{ where}\\ \lambda &= \begin{cases} (x_1^2 + y_1)/x_1 & \text{if } P_1 = P_2 \neq -P_2\\ (y_1 + y_2)/(x_1 + x_2) & \text{if } P_1 \neq \pm P_2 \end{cases}. \end{aligned}$$

The negative of a point is $-(x_1, y_1) = (x_1, y_1 + x_1)$ and $(x_1, y_1) + (x_1, y_1 + x_1) = \infty$.

All curves of the form $y^2+xy = x^3+x^2+b$ have a co-factor of 2, with $(0, \sqrt{b})$ being a point of order 2. Varying b varies the group order but the term x^2 means that there is no point of order 4. Essentially all integer orders within the Hasse interval $[2^{113}+1-2\cdot 2^{113/2}, 2^{113}+1+2\cdot 2^{113/2}]$ that are congruent to 2 modulo 4 are attainable by changing b within $\mathbb{F}_{2^{113}}$.

Cryptographic applications work in the subgroup of order ℓ. Because ℓ is odd, 2 is invertible modulo ℓ, so there exists an s with $2s \equiv 1 \bmod \ell$ and each point R in this subgroup is the double of sR. Seroussi showed in [23] that $\mathrm{Tr}(x) = \mathrm{Tr}(a)$ whenever (x, y) is the double of any point. For $\mathbb{F}_{2^{113}} \cong \mathbb{F}_2[w]/(w^{113} + w^9 + 1)$ one can prove using Newton's identities (see Appendix A) that $\mathrm{Tr}(w^i) = 0$ for $1 \le i \le 112$ and, of course, $\mathrm{Tr}(1) = 1$. If $x = \sum_{i=0}^{112} x_i w^i$ then $\mathrm{Tr}(x) = \sum_{i=0}^{112} x_i \mathrm{Tr}(w^i) = x_0$ since the trace is additive. This implies that for our curve having $a = 1$, each point in the subgroup of order ℓ has $x_0 = \mathrm{Tr}(x) = \mathrm{Tr}(a) = 1$, i.e., the least significant bit in the representation of x is 1.

3 Pollard iterations

Our attack uses the parallel version of Pollard's rho algorithm [22] by van Oorschot and Wiener [25] to compute the discrete logarithm of Q to the base P. This algorithm works in a client-server approach.

Each client (in our case each FPGA process) receives as input a point R_0 which is a known linear combination in P and Q, i.e., $R_0 = a_0P + b_0Q$. From this input point it starts a pseudorandom walk, where each step depends only on the coordinates of the current point R_i and preserves knowledge about the linear combination in P and Q, i.e., $R_i = a_iP + b_iQ$. The walk ends when it reaches a so-called “distinguished point” R_d, where the property of being distinguished is a property of the coordinates of the point. This distinguished point is then reported to a server together with information that allows the server to obtain a_d and b_d.

The server searches through incoming points until it finds a collision, i.e., two walks that ended up in distinguished points $R_{d_1} = a_{d_1}P + b_{d_1}Q$ and $R_{d_2} = a_{d_2}P + b_{d_2}Q$ that are equal. With very high probability, the coefficients b_{d_1} and b_{d_2} are distinct modulo ℓ, so we can compute the discrete logarithm as $\log_P Q \equiv (a_{d_1} - a_{d_2})/(b_{d_2} - b_{d_1}) \bmod \ell$.

In the following, we describe the construction of our iteration function. We start with a simple version, which does not make use of the negation map, and then modify this walk to perform iterations modulo negation.

Non-negating walk. Our iteration function follows the standard approach of an additive walk (see e.g. [24]) with some improvements following [5]. We precompute a table $(T_0, \ldots, T_{n-1})$ of random multiples of the base point P; our implementation uses $n = 1024$. Older descriptions often define steps to be combinations of P and Q, but Q is a multiple of P itself, so taking random multiples of P has the same effect and makes the step function independent of the target discrete logarithm. This means the design including the precomputed points can be synthesized for the FPGA and then used to break multiple discrete logarithms. Inputs to the iteration function are random multiples of the target point Q. Our iteration function f is defined as

$$R_{i+1} = f(R_i) = R_i + T_{I(R_i)},$$

where $I(R_i)$ takes the coefficients of $w^{10}, w^9, \ldots, w^1$ of the x-coordinate of R_i, interpreted as an integer. We ignored the coefficient of w^0 because it is 1 for all points (see Section 2) and chose the next 10 least significant bits in order to avoid overlap with the distinguished-point property defined in the next paragraph.

After each iteration, we check whether we have reached a distinguished point. We call a point *distinguished* when the 30 most significant bits of the x-coordinate are zero. If the point is a distinguished point, it is output, otherwise the iteration proceeds.

In the literature, there are two different approaches of how to continue after a distinguished point has been found. The traditional approach is to report the point and the linear combination leading to it and then to simply continue with the random walk. This approach has been used, for example, in [15], [6], [7], [26], and [27]. The disadvantage of this approach is that the iteration function needs to update the coefficients of the linear combination of P and Q (at least the coefficient of P, and also the coefficient of Q with older definitions of steps); in our case this would mean that the FPGAs not only have to perform arithmetic in $\mathbb{F}_{2^{113}}$ but also big-integer arithmetic modulo the 113-bit group order ℓ.

A more efficient approach was suggested in [1] and [5]. Once a distinguished point has been found the walk stops and reports the point. The processor then starts with a fresh input point. This means that all walks have about the same length, in this case about 2^{30} steps. The walks do not compute the counters for the multiples of P and Q; instead they remember the initial multiple of Q. The server stores this initial multiple (in the form of a seed) and the resulting distinguished point. Once a collision between distinguished points has been found, we simply recompute the two colliding walks and this time compute the multiples of P. We wrote a non-optimized software implementation based on NTL for this task, which took time on the scale of an hour to recompute the length-2^{30} walks and solve the DLP.

Walks modulo negation. Like [27] and various earlier papers, we reduce the expected number of iterations for an ECDL computation by computing iterations modulo the efficiently computable negation map. This improvement halves the search space of Pollard rho and thus gives a theoretic speedup of $\sqrt{2}$. The use of the negation map has been an issue of debate: see [8] for arguments against and [5] for an implementation that achieves essentially the predicted speedup.

Changing the walk to work modulo the negation map requires two changes. First, we have to map $\{P, -P\}$ to a well-defined representative. We denote this representative $|P|$. Recall that the least significant bit of x is always 1 (see the trace discussion at the end of Section 2) and that $-(x, y) = (x, x + y)$. We pick the point that has the least significant bit of y being 0 as representative. After each step of the iteration function we inspect the y-coordinate of the reached point R_i and continue with $-R_i$ in case the least significant bit of y is 1. This requires one bit comparison and one field addition.

The traditional approach would instead be to take whichever of R_i and $-R_i$ has a lexicographically smaller y coordinate. Our approach, relying on the $x_0 = 1$ observation, replaces a lexicographic comparison with a single bit comparison, noticeably reducing area overhead.

Second, we need a mechanism to escape so-called *fruitless cycles*. These mini-cycles stem from the combination of additive walks and walks defined modulo negation. The most basic and most frequent case of a fruitless cycle is a 2-cycle. Such a cycle occurs whenever $I(R_i) = I(R_{i+1})$ and $R_{i+1} = |(R_i + T_{I(R_i)})| = -(R_i + T_{I(R_i)})$. In this case, R_{i+2} is again R_i and the walk is caught in a cycle consisting of R_i and R_{i+1}. The probability of this to occur is $1/(2n)$, where n is the number of precomputed points. There also exist larger fruitless cycles of lengths $4, 6, 8$ etc., but the frequency of those is much lower [11].

Bernstein, Lange and Schwabe suggest in [5] detecting fruitless cycles by checking frequently for cycles of length 2 and increasingly less frequently for cycles of higher length. However, they are using a vectorized software implementation where frequent checks for cycles are expensive. We are using an unrolled hardware design; checking for cycles has no impact on the computational throughput and only a small impact on area demand. Also, cycle checking can be done individually for each independent walk in the pipeline without impact on the other walks (in contrast to a vectorized implementation where the same operation must be applied to all walks jointly in the data vectors).

We use a simple 4-bit counter, allowing us to detect all cycles of length up to 16. This prevents practically all infinite loops during the computation: cycles of length more than 10 are extremely unlikely. Once a cycle is detected, a deterministic way of leaving the cycle is required regardless of where the cycle was entered; two independent walks that enter the same cycle at a different entry point must leave the cycle at the same point in order to eventually end in the same distinguished point. Therefore, we record the current minimum x-coordinate in the 16-step cycle-detection window. Whenever we reach a point with a smaller x-coordinate than the current minimum, the cycle counter is reset and the minimum x-coordinate is updated. When we reach the same point, i.e. due to the negation map the same x-coordinate as the stored minimum then we are in a cycle and have to escape the cycle by doubling the current point. If the counter has an overflow to 0, i.e., we did not encounter a cycle in the last 16 steps, the current minimum is reset to the current x-coordinate and the cycle-detection is restarted.

We use the same criterion for a distinguished point (30 zeros) and the same table of precomputed steps as described in the previous subsection.

Justification of distinguished point property. For the `sect113r2` curve the expected number of group operations is roughly 2^{56}. Each walk takes about 2^{30} steps to reach a distinguished point and so we expect about 2^{26} distinguished points before we find a collision. This amount of data poses no problem for the host PC and for the I/O part of the hardware. For larger DL computations a less frequent property needs to be chosen. A benefit of relatively short walks is that they are easily recomputed on a PC, which we use for finding the DL after

```
 1 def random_step(x, y, ctr, double, x_min):
 2 # point addition/doubling
 3   T_x = get_precomputed_x(x)
 4   l1 = x if double else (x + T_x)
 5   l1_inv = 1/l1
 6   T_y = get_precomputed_y(x)
 7   l0 = (x^2 + y) if double else (y + T_y)
 8   l = l0 * l1_inv
 9   x3 = l^2 + l + 1
10   x3 = x3 if double else x3 + l1
11   dist = check_dist(x)
12   tmp = l * (x + x3)
13   x = x if dist else x3
14   y = y if dist else tmp + y + x3
15   c_x_y = (get_lsb(y) == 1)
16   y = (x + y) if c_x_y else y
17
18 # cycle detection
19   ctr = (ctr + 1) % 16
20   c_ctr = (ctr == 0)
21   c_lt = (x < x_min)
22   c_new_min = c_lt or c_ctr or double
23   double = (x == x_min)
24   x_min = x if c_new_min else x_min
25   ctr = 0 if c_lt else ctr
26   return (x, y, ctr, double, x_min)
```

Fig. 4.1: Sage code for the iteration function.

a collision of distinguished points occurs. This also helped in verifying that the FPGA code computed the same walks as a software implementation.

4 Implementation

The main core of the iteration function is a point addition, either the addition of the current state point with a point from the precomputed table or in case a cycle was detected the doubling of the current state point.

Doubling of a point is quite similar to addition of two distinct points (see Section 2 for the standard addition formula) but removes one finite-field addition and includes one extra finite-field squaring. This can easily be expressed using conditional assignments. Figure 4.1 shows Sage code for the iteration function; the point addition/doubling part is in lines 3 to 16. The code doubles the current state point (x, y) in case the `double` flag is true or adds a point $T = (T_x, T_y)$ from a precomputed table (depending on some bits of the current x-coordinate). The current state point is updated unless a distinguished point has been reached (`check_dist` returns true if (x, y) is a distinguished point). Lines 15 and 16 implement the negation map using a conditional assignment: if the least significant bit of y is 1, the current point is replaced with $-(x, y) = (x, x + y)$.

```
def GF113_inv(x):
  r1 = x^(2^1)
  r2 = x^(2^2)
  r1 = r2 * r1
  r0 = r1^(2^2)
  r1 = r0 * r1
  r0 = r1^(2^4)
  r1 = r0 * r1
  r0 = r1^(2^8)
  r2 = r0 * r1
  r0 = r2^(2^16)
  r3 = r0 * r2
  r0 = r3^(2^32)
  r1 = r0 * r3
  r0 = r1^(2^32)
  r1 = r0 * r3
  r0 = r1^(2^16)
  r0 = r0 * r2
  return r0
```

Fig. 4.2: Sage code for finite-field inversion in $\mathbb{F}_{2^{113}}$.

The Sage code for cycle detection is shown in lines 19 to 25 in Figure 4.1. The variable ctr is a 4-bit counter. To leave a cycle, we store the x-coordinate of the minimum point (i.e., the point with the smallest x-coordinate) of a cycle in x_min. Whenever within 16 steps we reach a smaller point than the current minimum, the flag c_lt is set to true, x_min is updated, and the counter value is reset to 0. A side effect of using the negation map is that we do not need to store the y-coordinate of the minimum point. In case the counter has an overflow to 0, i.e., we did not encounter a cycle within 16 iteration steps, the flag c_ctr is set to true and we move the detection window forward by setting the minimum x_min to the current point. In case we re-visit a point, i.e., the current x-coordinate is equal to x_min, the flag double is set to true for the next iteration resulting in a point doubling in the top part of the code in Figure 4.1. If there just was a point doubling in the current iteration, x_min is updated with the current x-coordinate as well in order to restart cycle detection.

The state of the iteration function consists of the x and y coordinates of the current point. For cycle detection, additionally we require a 4-bit counter, the flag double, and the x-coordinate of the minimum point of the cycle window. In total the state requires $2 \times 113 + 4 + 1 + 113 = 344$ bits.

The functional description of the iteration function shows that we need several finite-field operations for the FPGA design, i.e., addition, squaring, multiplication, inversion, and comparison. Inversion in the finite field is an expensive operation that can be implemented using a sequence of squarings and multiplications to compute $a^{-1} = a^{2^{113}-2}$. Figure 4.2 shows the inversion ladder that we are using. The number of multiplications is 8, which is the length of the shortest addition chain for 112. This inversion procedure requires 8 multiplications and

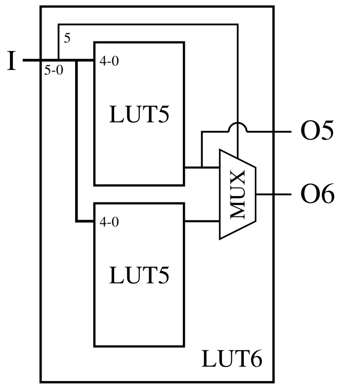

Fig. 4.3: Structure of a LUT-6 implemented as two LUT-5. Input I is a 6-bit bus.

113 squarings. Consecutive squarings can be combined to powers of higher order depending on which power is suitable and most efficient for the implementation.

Low level functions. The main components for implementing logical expressions on an FPGA are lookup tables (LUTs). The LUTs in the Spartan-6 are LUT-6 with 6 input wires. However, internally each LUT-6 is implemented with two LUT-5 using the same input wires. The sixth input wire selects the final output by controlling a 2-bit multiplexer (see Figure 4.3). The LUTs can be configured either as `LUT_6` providing LUT-6 functionality or as `LUT_6_2` that gives access to each output of the two LUT-5.

Addition. This operation requires a very small amount of logic. 57 LUTs in `LUT_6_2` configuration (providing 2×57 output bits) are sufficient for the implementation. However, often addition can be combined with follow-up operations like squaring such that the logic might be absorbed. We do not explicitly implement addition with `LUT_6_2` components but leave it to the Xilinx tool chain to map the VHDL code.

Squaring. This operation requires simply inserting zeros between the coefficients and performing a reduction modulo $w^{113} + w^9 + 1$. Consecutive squarings can be combined in order to absorb logic into a smaller number of LUTs. Single squarings appear in combination with addition in our design. Therefore, we express single squarings as VHDL code and leave it to the Xilinx tool chain to combine the logic. For sequences from 2 to 8 squarings, we generate optimized logic and explicitly use `LUT_6_2` components.

Multiplication. This operation is the most expensive operation in terms of area. We use three levels of Karatsuba multiplication. To have an even number of bits per chunk on all levels of our Karatsuba multiplication, we handle the top bits of the operands separately and perform Karatsuba on 112-bit operands, resulting in 27 multiplications of 14-bit polynomials.

For the low-level multiplications, we generate optimized logic using `LUT_6_2` components. Figure 4.4 shows an example of how we cover the terms; adding up

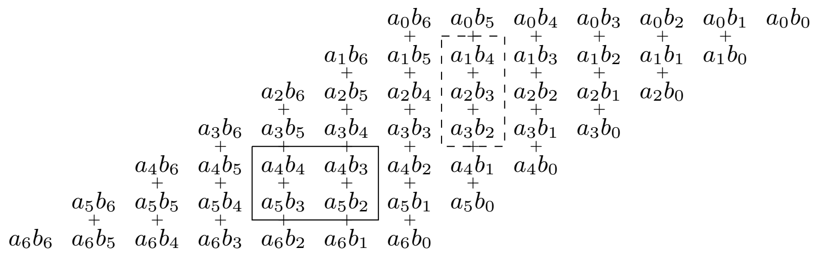
a0b6 + a0b5 + a0b4 + a0b3 + a0b2 + a0b1 + a0b0
a1b6 + a1b5 + a1b4 + a1b3 + a1b2 + a1b1 + a1b0
a2b6 + a2b5 + a2b4 + a2b3 + a2b2 + a2b1 + a2b0
a3b6 + a3b5 + a3b4 + a3b3 + a3b2 + a3b1 + a3b0
a4b6 + a4b5 + a4b4 + a4b3 + a4b2 + a4b1 + a4b0
a5b6 + a5b5 + a5b4 + a5b3 + a5b2 + a5b1 + a5b0
a6b6 a6b5 a6b4 a6b3 a6b2 a6b1 a6b0

Fig. 4.4: Example for assigning terms of a 7×7 polynomial multiplication to LUT-6's. The dashed box requires six inputs but covers only three terms. The solid box requires only five inputs for four terms and thus can be implemented using a `LUT_6_2`.

the columns in the figure gives the result of a 7×7 polynomial multiplication. Using a LUT-6 as in the dashed box covers only three terms using 6 inputs; using a `LUT_6_2` as in the solid box requires only five separate inputs but covers 4 terms, the two output wires of the `LUT_6_2` are used for the two involved columns (requiring independent sums). Special care needs to be taken at the boundaries. Additional logic is required to sum up over each column.

Also for the preparation of the inputs for the low-level multiplications and for the computation of the total result we are generating optimized logic using `LUT_6_2` components whenever possible. All in all, one $\mathbb{F}_{2^{113}} \times \mathbb{F}_{2^{113}}$ multiplication requires on average 3071 LUTs (after placement and routing). The implementation of the multiplication is pipelined and requires three clock cycles.

Comparison. For cycle detection we require a less-than comparison and an equality check on the same inputs. We implemented optimized logic to compute both operations at once using `LUT_6_2` components.

Implementing the iteration function. Our goals are high throughput with low overhead. Therefore, we implement the main part of the iteration function with fully pipelined, unrolled code. All components are busy all the time: the design computes one step of the random walk in each cycle while working on many independent random walks in parallel in a pipelined fashion.

However, using this approach for finite-field inversion as well would require a large amount of resources. The iteration function requires two multiplications, one inversion, and various additions and squarings. The inversion itself requires 8 multiplications, thus demanding more than 80% of the total resources. We do better by using Montgomery's trick for inversion, combining several independent inversions and thus reducing the relative cost of the inversion. Montgomery's trick requires 3 additional multiplications.

We use a dual-buffer design to implement a pipelined version of the Montgomery inversion. The buffers are used as follows:

1. Fill buffer 1 with data, using one multiplier to compute the overall product of the buffer.
2. Invert the product of buffer 1 while filling buffer 2. The size of the buffers must be large enough to hide the latency of one inversion.
3. Once the inverse of the first product has been computed, empty buffer 1 in reverse order by computing the individual inverses using two multipliers while at the same time filling buffer 1 again using the just emptied slots. At the same time, invert the overall product of buffer 2.
4. Continue iteratively by filling buffer 1 and buffer 2 periodically, alternating in ascending and descending order.

Given a sufficient amount of buffer memory, the latency of the actual inversion within the Montgomery inversion does not matter. Therefore, we implemented the inversion not as unrolled code, but by using a core design with a custom-made instruction set and one single finite-field multiplier. This results in a lower area consumption than when using an unrolled core for the price of a higher overall latency and lower utilization of the logic for the finite-field operations within the inverter.

To simplify development and to increase flexibility, we wrote tools that automatically generate unrolled code (for the iteration function) and a core design (for the inversion) from Sage code. Therefore, the logic of the iteration function can easily be tested using Sage and also easily be altered at any time.

In total, we can put two complete iteration functions with individual Montgomery inverters on one Spartan-6 XC6SLX150 FPGA. However, Montgomery's trick can also be used to share one inverter between several instances of the iteration function, giving space for more instances. By sharing one inverter between all instances, we are able to put three instances of the iteration function on the FPGA. The additional operations (multiplications in the aggregation and crossmultiplications; logic) for Montgomery's trick are simply implemented on the inversion core without cost for further multipliers. This increases the overall latency of the inversion, but as explained above we use buffering to hide this latency.

Overall architecture. To streamline the design, we arrange the instances of the iteration function in a circle: the output of one instance is the input for the next one. Therefore, the design requires only one single IO point. Figure 4.5 shows the overall layout of our design.

The host computer randomly computes starting points of independent random walks using a 64-bit seed. During computation of the random walks on the FPGAs, the seeds are stored on the host computer; only a 12-bit temporal ID is sent to the FPGAs along with the point coordinates in order to associate random walks with their seeds.

Each IO instance stores incoming data in a buffer. If there is an empty slot in the pipeline (either during setup phase in the beginning or because a distinguished point has been returned to the host computer), the IO interface stets up the state of a new random walk using a fresh stating point and its 12-bit ID and puts it into the pipeline. Every clock cycle, the pipeline feeds a state into

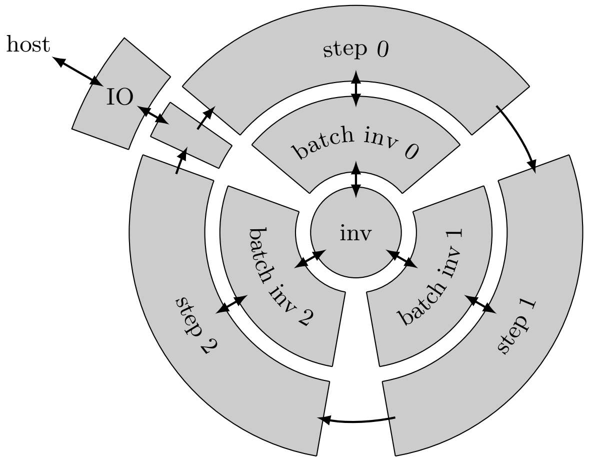

Fig. 4.5: Overall layout of the design with one single inverter (inv) and three unrolled iteration functions (step 1-3) using Montgomery's trick for inversion (batch inv 1-3).

the first instance of the iteration function (step 0) which computes one step of the random walk. During the inversion, the state data of the random walk is stored in a buffer (batch inv 0). Once the first iteration step is computed, the pipeline forwards the state to the next instance of the iteration function (step 1). The state circles through the instances, step by step computing a random walk, until a distinguished point is reached.

Now, the distinguished point is forwarded back to the IO interface. Random-walk computations are still performed on the state on the way through the instances, but the state is not updated anymore (see lines 13 and 14 in Figure 4.1). Once a state with a distinguished point arrives at the IO interface, the interface returns the x-coordinate and the ID of the distinguished point to the host and fills the pipeline slot using a fresh input point. The host associates the original 64-bit seed with the distinguished point using the 12-bit ID and sends the seed and the x-coordinate to the server which sorts incoming points, detects collisions, and finally computes the discrete logarithm.

The first column of Table 4.6 shows the area demand of the design for the different components.

The final 3-core design routes for and runs at 100MHz. We also tested a design with 4 iteration cores on one XC6SLX150 FPGA. However, the power consumption of the design was too high and the design did not run stably, producing incorrect results. Furthermore, we tried to increase the frequency of the design. We were able to place and route and also run a single-core design at up to 160MHz; however, our attempts to increase the frequency on the 3-core design failed to produce a stable, operational design.

			$\mathbb{F}_{2^{113}}$			$\mathbb{F}_{2^{127}}$		
Module	Inst.	Mult. per instance	LUTs per instance	LUTs total	FPGA util.	LUTs per instance	LUTs total	FPGA util.
3 cores with inv.				57,956	63%		68,376	74%
iteration func.	3	2	7,789	23,368	25%	9,123	27,369	30%
$\mathbb{F}_{2^{113}}$ inv.	1	1	5,817	5,817	6%	7,151	7,151	8%
batching	3	3	9,535	28,605	31%	11,397	34,191	37%
$\mathbb{F}_{2^{113}}$ mult.	16		3,071	49,131	53%	3,620	57,927	64%
total (incl. IO)				63,388	69%		72,919	79%

Fig. 4.6: Area consumption by component. All values are post placement.

For testing our design we used Spartan-6 development boards from SciEngines and from Opal Kelly. For running the attack, we used two "Rivyera" FPGA-cluster computers from SciEngines with 64 Spartan-6 FPGAs each. A Rivyera is a classical "host" computer combined with to up to 128 FPGAs (in the high-density version up to 256 FPGAs). The FPGAs are connected to the host computer using a PCIe host interface. SciEngines provides an API for programming the FPGAs and for communication between the FPGAs and between FPGAs and the host computer.

Solving DLPs. We solved a 112-bit DLP on the sect113r2 curve with an earlier 2-core, 100MHz design within about 48 days and 3 hours using up to 120 Spartan-6 FPGAs. (Not all FPGAs were available for our computations all the time.) The solution required the computation of 82,177,699 distinguished points. The expected duration was 30 days and 19 hours for the computation of $\sqrt{\pi \cdot 2^{112}/4}/2^{30} \approx$ 59,473,682 distinguished points.

Currently, we are doing computations for solving a 117.35-bit DLP on an elliptic curve over $\mathbb{F}_{2^{127}}$ (see Appendix C for the specific curve parameters). The design has three cores and runs at 100MHz. We are using the same property for distinguished points as for our sect113r2 computations, namely the top 30 bits from the x-coordinate being zero. Solving the DLP is expected to require $\sqrt{\pi \cdot 2^{117.35}/4}/2^{30} \approx 379,821,956$ distinguished points. Using all 128 FPGAs of our cluster, we expect the computation to be finished after about 123 days. For details on the area consumption see the second column of Table 4.6.

Power consumption. We measured the power consumption of our designs in the following way: One of our Rivyera FPGA clusters with 64 FPGAs requires about 215W when the FPGAs are not programmed. Under full usage, while our $\mathbb{F}_{2^{113}}$ design is running, the total power demand is about 725W. Thus, 64 FPGAs require about 510W while running the $\mathbb{F}_{2^{113}}$ design; a single FPGA requires about 8W. For the $\mathbb{F}_{2^{127}}$ design the total power consumption is 755W. Therefore, 64 FPGAs require about 540W, a single FPGA about 8.4W.

5 Experiments

The obvious way to verify the performance and functionality of our implementation is to repeat the following procedure many times: generate a random point Q on the curve `sect113r2`, use the implementation to find k such that $Q = kP$, see how long this takes, and check that in fact $Q = kP$.

The reason for repeating this procedure many times is that the performance is a random variable. Checking the performance of a *single* DL computation would obviously be inadequate as a verification tool. For example, if the claimed *average* DL time is T while the observed time of a single DL computation is $2.3T$, then it could be that this particular computation was moderately unlucky, or it could be that the claim was highly inaccurate.

There are two reasons that more efficient verification procedures are important. First, it was feasible for us to carry out a `sect113r2` DL computation, but performing *many* such computations would have been quite expensive. Second, and more importantly, verification is not merely something to carry out in retrospect: it provides essential feedback during the exploration of the design space. Below we describe the verification steps that we took for our final implementation, but there were also many rounds of similar verification steps for earlier versions of the implementation.

Running hundreds or thousands of walks (a tiny fraction of a complete `sect113r2` DL computation; recall that we expect orders of magnitude more distinguished points for our selected parameters) produces reasonably robust statistics regarding the number of iterations required to find a distinguished point, and regarding the time used for each iteration. However, it does not provide any evidence regarding the number of distinguished points required to compute a DL. A recurring theme of several recent papers is that standard heuristics overestimate the randomness of DL walks, and thus underestimate the number of distinguished points required; see, e.g., the correction factors in [1, Appendix B] and the further correction factors in [4, Section 4].

To efficiently verify performance *including* walk randomness and successful DL computation, we adapt the following observation from Bernstein, Lange, and Schwabe [5]. The fastest available ECDL algorithms use the fastest available formulas for adding affine points, and those formulas are independent of some of the curve coefficients: specifically, [5] used formulas that are independent of b in $y^2 = x^3 - 3x + b$, and we use formulas that are independent of b in $y^2 + xy = x^3 + ax^2 + b$. The same algorithms thus work without change for points (and precomputed tables) on other curves obtained by varying b. Searching many curves finds curves with different sizes of prime-order subgroups, allowing tests of exactly the same ECDL algorithms at different scales.

For example, applying an isomorphism to `sect113r2` to obtain $a = 1$ as described earlier, and then changing b to `10010111`, produces a curve with a subgroup of prime order $1862589870449786557 \approx 2^{60.69}$. This group is large enough to carry out reasonably large experiments without distractions such as frequent self-colliding walks, and at the same time small enough for experiments to complete quickly.

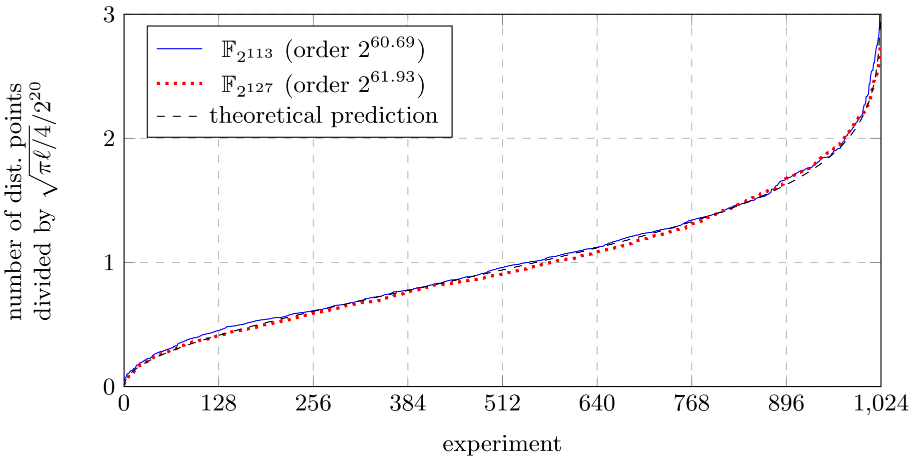

Fig. 5.1: Blue curve: 1024 independent experiments for $\mathbb{F}_{2^{113}}$. The experiments are sorted by the number of distinguished points they required to find a discrete logarithm, and are then placed at $x = 0$, $x = 1$, …, $x = 1023$ respectively. The y-axis is the number of distinguished points divided by $\sqrt{\pi\ell/4}/2^{20}$. Dotted red curve: 1024 independent experiments for $\mathbb{F}_{2^{127}}$, organized the same way. Dashed black curve: $y = \sqrt{-(4/\pi)\log(1 - x/1024)}$ from standard rho theory. This curve shows that our experiments are close to the expected distribution.

We performed 1024 DL computations on this curve, in each case using 20 bits to define distinguished points. These computations used a total of 1201100 walks. The average number of walks per DL was slightly over 1173. For comparison, the predicted average is $\sqrt{\pi\ell/4}/2^{20} \approx 1153$ for $\ell = 1862589870449786557$, and the predicted standard deviation is on the same scale as the predicted average; the gap between 1173 and 1153 is unsurprising for 1024 experiments. Each computation successfully produced a verified discrete logarithm.

We defined the first DL computation to use seed $0, 1, 2, \ldots$ until finding a collision between seed s and a previous seed; the second DL computation to use seeds $s+1, s+2, \ldots$ until finding a collision within those seeds; etc. We post-processed seeds with AES before multiplying them by Q, so (if AES is strong) choosing consecutive seeds is indistinguishable from choosing independent uniform random 128-bit scalars.

The advantage of choosing consecutive seeds is that, without knowing in advance which seeds would be used in each computation, we simply provided a large enough batch of seeds $0, 1, 2, \ldots$ to our FPGAs. Retroactively attaching each seed to the correct computation was a simple matter of sorting the resulting distinguished points in order of seeds and then scanning for collisions. The sorting step here is important: if we had scanned for collisions using the order of points output by the FPGAs then we would have incorrectly biased the initial computations towards short walks.

We performed the same experiment on a curve over $\mathbb{F}_{2^{127}}$ with a subgroup of order $\ell_2 = 4389832188282442501 \approx 2^{61.93}$. In this case 1024 DL computations required 1,792,905 distinguished points. The average number of distinguished points per DL was 1751. The predicted average is $\sqrt{\pi\ell_2/4}/2^{20} \approx 1771$. Figure 5.1 shows the number of required distinguished points divided by the predicted average for both experiments. The experiments have been ordered by the number of required distinguished points.

We also carried out various experiments over $\mathbb{F}_{2^{113}}$ with

- a group of size $2149433571795004101539 \approx 2^{70.86}$ with $b = \texttt{110}$,
- a group of size $2608103394926752635062767 \approx 2^{81.11}$ with $b = \texttt{100111}$, and
- a group of size $1534122330555159121115288777 \approx 2^{90.31}$ with $b = \texttt{10000111}$.

We spot-checked walks against a separate software implementation, verified correctness of 16 DL computations for the 70-bit group, and verified correctness of 1 DL computation for the 80-bit group.

6 Comparison

Section 1 summarized how our results improve upon the recent results [27] from Wenger and Wolfger. This section compares the results and techniques in more detail.

Multiplier area. A few different multiplier structures are considered in [27, Section 6.8 and Appendix B]. The best results, 3757 LUTs, rely on traditional power-of-2 Karatsuba multipliers, for example building a 64-bit multiplier from three 32-bit multipliers. We do much better, 3071 LUTs, by exploring a much wider range of optimizations: in particular, we drop the power-of-2 restriction, allowing efficient use of three Karatsuba levels, and we optimize the low-level usage of LUTs.

Number of multipliers. The design in [27] applies an inverter to a batch of inputs, where each core provides one input from its first walk; then applies the same inverter to another batch of inputs, where each core provides one input from its second walk; etc. This requires a high-throughput inverter: [27] uses a fully unrolled inverter, requiring 8 multipliers and 112 squarers.

We instead use a dual-buffer memory to batch inversions across cores *and* across many random walks from each core. This lets us use a high-latency inverter without slowing down the rest of the design. This, in turn, allows us to use a low-area "core" design for the inverter, requiring only one multiplier and one module each to compute a^2, a^{2^2}, a^{2^4}, and a^{2^8}.

A slight disadvantage is that for c cores we need $3c$ multipliers to batch inversions, whereas in [27] one core can skip 3 of these multipliers, for a total of just $3c - 3$ multipliers to batch inversions. Furthermore, to simplify routing we synthesized 5-core, 6-core, 7-core, and 8-core designs as two separate clusters, with a separate inverter in each cluster. However, overall we still save the area for 3 multipliers in $\leq$8-core designs, or 4 multipliers in $\leq$4-core designs.

cores	LUTs	registers	RAMs	slices	source
5	73%	41%	31%	80%	[27]
6	62%	28%	15%	84%	new
7	71%	33%	17%	91%	new
8	80%	37%	19%	96%	new

Fig. 6.1: Resource utilization of several ECDL designs synthesized for the XC7K325T-2 at 180MHz.

Total area. All in all, our improvements and optimizations reduce the area cost significantly compared to [27]. As noted in Section 1, we do not have access to any Kintex-7 FPGAs for testing, but for comparability we nevertheless synthesized our design for the XC7K325T-2 used in the KC705 development boards in [27]. Table 6.1 shows that our 7-core design uses fewer LUTs, fewer registers, and fewer block RAMs than the 5-core design in [27]; it uses more slices but we expect it to run stably at the same 180MHz. We assume that [27] synthesized the design for larger frequencies (resulting in a closer placement and thus requiring fewer slices) and later experimentally tried out the maximum frequency delivering stable results. Note that overheating is the main bottleneck identified in [27], and heat is generated primarily by computation and memory access, not by chip area per se. With access to this FPGA we could verify stability for 7 or 8 cores and possibly fine-tune the design to allow higher frequencies.

Fruitless cycles. A further advantage of our design is that we waste fewer iterations on fruitless cycles. Specifically, we use doublings to escape fruitless cycles, while [27] uses additions. The detailed analysis in [8] indicates that additions create new types of fruitless cycles, whereas doublings avoid this problem.

[27, p. 4] argues that using additions to escape fruitless cycles has “a huge advantage when a hardware design is done” since “no on-chip point doubling circuit is necessary”. However, in our unrolled design, the doubling circuit reuses the addition circuit with miniscule additional area cost.

We performed many experiments to check that our iteration function is correctly computed by our implementation and that it is as effective as expected, gaining a factor $\sqrt{2}$ in the average observed number of iterations compared to not using the negation map; see Section 5. The small-scale negation-map experiments reported in [27, Table 2] show a speedup factor only 1.32, i.e., 6% worse than $\sqrt{2}$. This gap is consistent with the analysis in [8].

[27, Table 2, “Point doubling” entry] reports doubling experiments that were also 6% worse than $\sqrt{2}$. This is inconsistent with the analysis and experiments in [8] (and with our own experiments); [27] does not discuss this inconsistency. Presumably the “Point doubling” experiment in [27] actually used a slightly different cycle-escape method from what we call doubling, but no further details are provided in [27].

Target curve. [27] illustrates its techniques by attacking the curve `sect113r1`, while we illustrate our techniques by attacking the curve `sect113r2`; both of these curves are defined in the same SEC 2 standard. The prime orders are

$$5192296858534827689835882578830703 \approx 2^{112.00000000000000001703}$$
$$5192296858534827702972497909952403 \approx 2^{112.00000000000000002068}$$

respectively. The `sect113r1` group size was summarized as "2^{112}" in [27] but as "bit size 113" in [14]. Note that rounding the exponent to the nearest integer also produces 112 for the earlier ECDL record in [6], whereas rounding the exponent up creates a large separation (41% difference in estimated attack cost) between practically identical orders marginally above and below a power of 2. We suggest instead rounding to two digits after the decimal point in the exponent: $2^{111.78}$ for [6], $2^{112.00}$ for [27], and $2^{112.00}$ for this paper.

References

1. D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. V. Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gürkaynak, 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. Cryptology ePrint Archive, Report 2009/514, 2009. https://eprint.iacr.org/2009/541/. 1, 8, 17
2. D. J. Bernstein, H. Chen, C. Cheng, T. Lange, R. Niederhagen, P. Schwabe, and B. Yang. ECC2K-130 on NVIDIA GPUs. In G. Gong and K. C. Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 328–346. Springer, 2010. https://cr.yp.to/papers.html#ecc2k130. 1
3. D. J. Bernstein and T. Lange. Computing small discrete logarithms faster. In S. D. Galbraith and M. Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012: 13th International Conference in Cryptology in India*, volume 7668 of *Lecture Notes in Computer Science*, pages 317–338, Kolkata, India, Dec. 9–12, 2012. Springer, Heidelberg, Germany. https://eprint.iacr.org/2012/458. 4
4. D. J. Bernstein and T. Lange. Two grumpy giants and a baby. In E. W. Howe and K. S. Kedlaya, editors, *ANTS X: proceedings of the tenth algorithmic number theory symposium*, pages 87–111. Mathematical Sciences Publishers, 2013. https://eprint.iacr.org/2012/294. 17
5. D. J. Bernstein, T. Lange, and P. Schwabe. On the correct use of the negation map in the Pollard rho method. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography – PKC 2011*, volume 6571 of *LNCS*, pages 128–146. Springer, 2011. https://cryptojedi.org/papers/#negation. 7, 8, 9, 17
6. J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Playstation 3 computing breaks 2^{60} barrier; 112-bit prime ECDLP solved, 2009. http://lacal.epfl.ch/112bit_prime. 2, 8, 21
7. J. W. Bos, M. E. Kaihara, and P. L. Montgomery. Pollard rho on the PlayStation 3. In *Workshop Record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, pages 35–50, 2009. https://hyperelliptic.org/tanja/SHARCS/record2.pdf. 2, 8
8. J. W. Bos, T. Kleinjung, and A. K. Lenstra. On the use of the negation map in the Pollard rho method. In G. Hanrot, F. Morain, and E. Thomé, editors,

Algebraic Number Theory, volume 6197 of *LNCS*, pages 66–82. Springer, 2010. http://www.joppebos.com/files/negation.pdf. 8, 20
9. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, version 1.0, 2000. http://www.secg.org/SEC2-Ver-1.0.pdf. 3
10. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, version 2.0, 2010. http://www.secg.org/sec2-v2.pdf. 3
11. I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In K. Lam, E. Okamoto, and C. Xing, editors, *Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18, 1999, Proceedings*, volume 1716 of *Lecture Notes in Computer Science*, pages 103–121. Springer, 1999. 9
12. ECC Brainpool. ECC Brainpool standard curves and curve generation, 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf. 5
13. J. Fan, D. V. Bailey, L. Batina, T. Güneysu, C. Paar, and I. Verbauwhede. Breaking elliptic curve cryptosystems using reconfigurable hardware. In *International Conference on Field Programmable Logic and Applications, FPL 2010*, pages 133–138. IEEE, 2010. https://www.cs.ru.nl/~lejla/FPL2010.pdf. 2
14. S. D. Galbraith and P. Gaudry. Recent progress on the elliptic curve discrete logarithm problem. Cryptology ePrint Archive, Report 2015/1022, 2015. https://eprint.iacr.org/2015/1022. 21
15. R. J. Harley. Solution to Certicom's ECC2K-95 problem (email message), 1998. http://cristal.inria.fr/~harley/ecdl5/ECC2K-95.submission.text. 8
16. Y. Hitchcock, P. Montague, G. Carter, and E. Dawson. The security of fixed versus random elliptic curves in cryptography. In R. Safavi-Naini and J. Seberry, editors, *ACISP 03: 8th Australasian Conference on Information Security and Privacy*, volume 2727 of *Lecture Notes in Computer Science*, pages 55–66, Wollongong, NSW, Australia, July 9–11, 2003. Springer, Heidelberg, Germany. 4
17. F. Kuhn and R. Struik. Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms. In S. Vaudenay and A. M. Youssef, editors, *SAC 2001: 8th Annual International Workshop on Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 212–229, Toronto, Ontario, Canada, Aug. 16–17, 2001. Springer, Heidelberg, Germany. http://grouper.ieee.org/groups/802/PrivRecsg/email/pdfJ3n8ucpxL8.pdf. 4
18. National Institute for Standards and Technology (NIST). Recommended elliptic curves for Federal government use, 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf. 4
19. National Institute for Standards and Technology (NIST). Digital signature standard (DSS) FIPS 186–4, 2013. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. 4
20. National Security Agency. Suite B cryptography / cryptographic interoperability, 2005. http://www.nsa.gov/ia/programs/suiteb_cryptography/. 5
21. T. Oliveira, J. López, D. F. Aranha, and F. Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In G. Bertoni and J.-S. Coron, editors, *CHES*, volume 8086 of *LNCS*, pages 311–330. Springer, 2013. https://eprint.iacr.org/2013/131. 5
22. J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978. http://www.ams.org/journals/mcom/1978-32-143/S0025-5718-1978-0491431-9/S0025-5718-1978-0491431-9.pdf. 7

23. G. Seroussi. Compact representation of elliptic curve points over $\mathbb{F}_{2^n}$. *HP Labs Technical Reports*, HPL-98-94R1, 1998. http://www.hpl.hp.com/techreports/98/HPL-98-94R1.html. 7
24. E. Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001. http://www.ams.org/journals/mcom/2001-70-234/S0025-5718-00-01213-8/S0025-5718-00-01213-8.pdf. 7
25. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999. http://www.scs.carleton.ca/~paulv/papers/JoC97.pdf. 7
26. E. Wenger and P. Wolfger. Solving the discrete logarithm of a 113-bit Koblitz curve with an FPGA cluster. In A. Joux and A. Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, volume 8781 of *LNCS*, pages 363–379. Springer, 2014. https://eprint.iacr.org/2014/368. 2, 8
27. E. Wenger and P. Wolfger. Harder, better, faster, stronger: elliptic curve discrete logarithm computations on FPGAs. *Journal of Cryptographic Engineering*, pages 1–11, 2015. https://eprint.iacr.org/2015/143. 2, 3, 4, 8, 19, 20, 21

A Trace calculations

Let $f \in \mathbb{F}_2[w]$ be an irreducible polynomial. Define α as the image of w in the finite field $\mathbb{F}_2[w]/(f)$. Newton's identities imply

$$\frac{f'}{f} = \sum_{i \geq 0} \frac{\mathrm{Tr}(\alpha^i)}{w^{i+1}}.$$

In particular, for $f(w) = w^{113} + w^9 + 1$, we have $f'(w) = w^{112} + w^8$, so

$$\begin{aligned}\frac{f'}{f} &= \frac{w^{112}+w^8}{w^{113}+w^9+1} = \frac{1/w + 1/w^{105}}{1 + 1/w^{104} + 1/w^{113}} \\ &= \left(\frac{1}{w} + \frac{1}{w^{105}}\right)\left(1 + \left(\frac{1}{w^{104}} + \frac{1}{w^{113}}\right) + \left(\frac{1}{w^{104}} + \frac{1}{w^{113}}\right)^2 + \left(\frac{1}{w^{104}} + \frac{1}{w^{113}}\right)^3 \cdots\right) \\ &= \frac{1}{w} + \frac{1}{w^{114}} + \cdots\end{aligned}$$

implying that $\mathrm{Tr}(\alpha^i) = 0$ for $1 \leq i < 113$. The third transformation uses the infinite geometric series $\sum_{i=0}^{\infty} \gamma^i = 1/(1-\gamma)$ with $\gamma = 1/w^{104} + 1/w^{113}$.

Similarly, for $f(w) = w^{127} + w + 1$, we have $f'(w) = w^{126} + 1$, so

$$\begin{aligned}\frac{f'}{f} &= \frac{w^{126}+1}{w^{127}+w+1} = \frac{1/w + 1/w^{127}}{1 + 1/w^{126} + 1/w^{127}} \\ &= \left(\frac{1}{w} + \frac{1}{w^{127}}\right)\left(1 + \left(\frac{1}{w^{126}} + \frac{1}{w^{127}}\right) + \left(\frac{1}{w^{126}} + \frac{1}{w^{127}}\right)^2 + \left(\frac{1}{w^{126}} + \frac{1}{w^{127}}\right)^3 \cdots\right) \\ &= \frac{1}{w} + \frac{1}{w^{128}} + \cdots\end{aligned}$$

implying that $\mathrm{Tr}(\alpha^i) = 0$ for $1 \leq i < 127$. Note that this is a general property of polynomials of this shape.

B Parameter for transforming the curve

$t = w^{112} + w^{111} + w^{108} + w^{107} + w^{106} + w^{104} + w^{101} + w^{96} + w^{95} + w^{91} + w^{89} + w^{88} + w^{87} + w^{86} + w^{83} + w^{82} + w^{81} + w^{80} + w^{78} + w^{75} + w^{74} + w^{67} + w^{64} + w^{63} + w^{62} + w^{61} + w^{60} + w^{58} + w^{57} + w^{53} + w^{50} + w^{49} + w^{46} + w^{43} + w^{42} + w^{41} + w^{39} + w^{37} + w^{36} + w^{33} + w^{32} + w^{31} + w^{30} + w^{28} + w^{26} + w^{24} + w^{23} + w^{19} + w^{17} + w^{15} + w^{14} + w^{13} + w^{11} + w^{10} + w^{9} + w^{7} + w^{5} + w^{4} + w^{3} + w^{2}$

Hexadecimal representation of t: `0x19D218BCF4C09F6264EB3D58AEEBC`.

C Parameters for the curve over $\mathbb{F}_{2^{127}}$

The curve is defined over $\mathbb{F}_{2^{127}} \cong \mathbb{F}_2[w]/(w^{127} + w + 1)$ by an equation of the form $E : y^2 + xy = x^3 + ax^2 + b$ and base point $P = (x_P, y_P)$, where

$$\begin{aligned} a &= \texttt{0x00000000000000000000000000000001}, \\ b &= \texttt{0x000000000000000000000000000001AB}, \\ x_P &= \texttt{0x3CF9CCD146B5E7440E9632F5D2B49679}, \text{ and} \\ y_P &= \texttt{0x43ED94FD97454C8197B6207C9A23C67E}. \end{aligned}$$

The order of P is 21214611404048532634861895907159 8183 $\approx 2^{117.35}$ which is prime. Figure C.1 shows Sage code for inversion in $\mathbb{F}_{2^{127}}$.

```
def GF127_inv(x):
  r0 = x^(2^1)
  r0 = r0*x
  r0 = r0^(2^1)
  r2 = r0*x
  r0 = r2^(2^3)
  r1 = r0*r2
  r0 = r1^(2^6)
  r1 = r0*r1
  r0 = r1^(2^3)
  r1 = r0*r2
  r0 = r1^(2^15)
  r1 = r0*r1
  r0 = r1^(2^30)
  r1 = r0*r1
  r0 = r1^(2^3)
  r1 = r0*r2
  r0 = r1^(2^63)
  r1 = r0*r1
  r0 = r1^(2^1)
  return r0
```

Fig. C.1: Sage code for finite-field inversion in $\mathbb{F}_{2^{127}}$.