Breaking ECC2K-130

Abstract. Elliptic-curve cryptography is becoming the standard public-key primitive not only for mobile devices but also for high-security applications. Advantages are the higher cryptographic strength per bit in comparison with RSA and the higher speed in implementations.
To improve understanding of the exact strength of the elliptic-curve discrete-logarithm problem, Certicom has published a series of challenges. This paper describes breaking the ECC2K-130 challenge using a parallelized version of Pollard's rho method. This is a major computation bringing together the contributions of several clusters of conventional computers, PlayStation 3 clusters, computers with powerful graphics cards and FPGAs. We also give estimates for an ASIC design. In particular we present

- our choice and analysis of the iteration function for the rho method;
- our choice of finite field arithmetic and representation;
- detailed descriptions of the implementations on a multitude of platforms: CPUs, Cells, GPUs, FPGAs, and ASICs;
- timings for CPUs, Cells, GPUs, and FPGAs; and
- details about how we are running the attack.

Keywords: Attacks, ECC, binary fields, DLP, Koblitz curves, automorphisms, parallelized Pollard rho, Certicom challenges.

1 Introduction

The Elliptic-Curve Discrete-Logarithm Problem (ECDLP) has received a lot of attention since the suggestion of elliptic curves for cryptography by Koblitz and Miller. In 1997, Certicom issued several challenges [Cer97a] in order to "increase the cryptographic community's understanding and appreciation of the difficulty of the ECDLP." The challenges over fields of size less than 100 bits were posed as exercises and were solved quickly. All challenges over fields of 109 bits were solved between April 2000 and April 2004 in distributed efforts. The challenges over fields of 131 bits are open; in particular there are two challenges over $\mathbf{F}_{2^{131}}$ and one over $\mathbf{F}_p$, where p is a 131-bit prime. One of the curves over $\mathbf{F}_{2^{131}}$ is a Koblitz curve [Kob92]. Koblitz curves are of interest for implementations because they support particularly efficient scalar multiplication. NIST has standardized 15 elliptic curves of which 5 are Koblitz curves A clear understanding of the security of Koblitz curves is thus very important.

The strongest known attacks against the ECDLP are generic attacks based on Pollard's rho method. Improvements by Wiener and Zuccherato [WZ98], van Oorschot and Wiener [vOW99], and Gallant, Lambert, and Vanstone [GLV00] showed how the computation can be efficiently parallelized and how group automorphisms can speed up the attack. Combining these methods gives the theoretical estimate that solving the Certicom challenge ECC2K-130, a Koblitz curve challenge over $\mathbf{F}_{2^{131}}$, should take $\sqrt{\pi 2^{131}/(2 \cdot 2 \cdot 4 \cdot 131)}$ iterations — but it leaves open how long these iterations take and how exactly they are done. In particular it is unclear how to choose the iteration function in Pollard's rho method to work on orbits under the automorphisms and avoid fruitless short cycles without compromising speed. Certicom's documentation states as one of the objectives of their challenge "To determine whether there is any significant difference in the difficulty of the ECDLP for random elliptic curves over $\mathbf{F}_{2^m}$ and the ECDLP for Koblitz curves."

We have improved arithmetic in $\mathbf{F}_{2^{131}}$, optimizing it for the operations needed to run the attack; improved the choice and analysis of the iteration function in the parallelized Pollard rho attack; improved the communication infrastructure for the distributed attack;

and implemented the attack on a multitude of platforms, covering different CPUs, the Cell Broadband Engine, Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). For comparison we also provide cost estimates for a dedicated ASIC design.

We have gathered access to several clusters of CPUs, Cells, GPUs and FPGAs and are currently attacking the ECC2K-130 challenge; we (anonymously) report progress on the attack at `www.ecc-challenge.info`. Our estimate is that the attack is finished before the Eurocrypt conference, which means that we are taking Certicom's claim "The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered" (see [Cer97a, p.22,p.23]) as a real challenge.

2 Design of the iteration function

The ECC2K-130 challenge is to solve a discrete-logarithm problem on the curve $E : y^2 + xy = x^3 + 1$ over $\mathbf{F}_{2^{131}}$, i.e., given two points P and Q of prime order ℓ on E to find a scalar k such that $Q = [k]P$. The parameters are given in Appendix A and can also be found online on Certicom's website [Cer97b].

The negative of a point $R = (x_R, y_R) \in E(\mathbf{F}_{2^{131}})$ is $-R = (x_R, y_R + x_R)$. The map $\sigma : E(\mathbf{F}_{2^{131}}) \to E(\mathbf{F}_{2^{131}})$ defined by $\sigma(x, y) = (x^2, y^2)$ is called the *Frobenius endomorphism*. There exists an integer s such that $\sigma(R) = [s]R$ for all points R in the subgroup of prime order ℓ. If x_R is given in normal-basis representation then x_R^2 is a cyclic shift of x_R. In particular the Hamming weight $\mathrm{HW}(x_R)$, i.e., the number of non-zero coefficients of x_R, is invariant under squaring in this representation.

We use Pollard's rho method [Pol78] in the parallelized version suggested by van Oorschot and Wiener [vOW99]. We have compared the performance of various iteration functions and now present our design followed by reasons for this choice.

Iteration function and distinguished points. Our condition for a point P_i to be a distinguished point is that in type-2 normal-basis representation $\mathrm{HW}(x_{P_i}) \leq 34$. Note that in the subgroup of order ℓ each x-coordinate has even Hamming weight by [Ser98].

Our iteration function is also defined via the normal-basis representation; it is given by

$$P_{i+1} = \sigma^j(P_i) \oplus P_i,$$

where $j = ((\mathrm{HW}(x_{P_i})/2) \bmod 8) + 3$. Harley's iteration function used in his attacks on ECC2K-95 and ECC2K-108 [Har] also uses a restricted set of Frobenius powers; his exponents are limited to a set of 7 elements that do not form an interval.

Our iteration function is well defined on orbits of points under the Frobenius endomorphism and under negation because in normal-basis representation all the points $\pm\sigma^c(P_i)$ give the same Hamming weight and because $\pm\sigma^c(\sigma^j(P_i) \oplus P_i) = \sigma^j(\pm\sigma^c(P_i)) \oplus (\pm\sigma^c(P_i))$. Also the distinguished-point property respects the orbits. Wiener and Zuccherato [WZ98] and Gallant, Lambert, and Vanstone [GLV00] studied the rho method on orbits and showed that a walk on orbits of size $2 \cdot 131$ reduces the number of iterations by a factor of $\sqrt{2 \cdot 131}$.

Our attack runs many iterations in parallel on many different platforms. Each path starts at a random point derived from a 64-bit random seed as follows: The seed is fed through AES to expand it to a 128-bit string $(c_{127}, c_{126}, \ldots, c_1, c_0)$; these coefficients are used to compute the starting point $Q \oplus \sum_{i=0}^{127} c_i \sigma^i(P)$. Once a distinguished point R is found we normalize it, i.e. we compute a unique representative of its orbit under negation and Frobenius map by taking the lexicographically smallest value among $x_R, x_R^2, \ldots, x_R^{2^{130}}$ in normal-basis representation.

We then report a 64-bit hash of the normalized point along with the 64-bit seed to the server. Since the walk is deterministic it can be recomputed from the 64-bit seed.

Disadvantages of the iteration function. The iteration function is defined via a normal-basis representation. Squarings are efficient in normal basis but multiplications are slower than in polynomial basis. If the computation is performed in polynomial-basis representation each x_{P_i} needs to be converted to normal-basis representation. This means that this iteration function imposes conversions or slower multiplications on the implementation.

Using $\mathrm{HW}(x_P)$ reduces the randomness of the walk — there are more points with weight around 65 than at the extremes. Additionally limiting the exponents j to $\{3, 4, \ldots, 10\}$ reduces the randomness further. We analyze the exact implications in Appendix B. These effects increase the running time of the attack to $\sqrt{\pi\ell/(2 \cdot 2 \cdot 131)} \cdot 1.069993 \approx 2^{60.9}$ iterations.

Advantages of the iteration function. Even though the iteration function is defined via a normal-basis representation it does not force the clients to compute in this representation. If for some platform the conversion of x_P to normal basis is less overhead than the slowdown in the multiplications in normal basis this client can compute in polynomial basis while others use normal basis. The walks and distinguished points will be the same.

Basing the iteration function on the Frobenius endomorphism avoids the need for normalizing each P_i that would be necessary in more traditional "adding walks" of the form $P_{i+1} = \mathrm{canon}(P_i) \oplus R_j$, where $\mathrm{canon}(P_i)$ is the canonical representative of the orbit of P_i, j is derived from the representation of $\mathrm{canon}(P_i)$ and R_j is from some set of precomputed scalar multiples of P and Q. Our choice also avoids the issue of short fruitless cycles of the form $P_{i+2} = -P_i$ that happen in adding walks and require extra scheduling and tests there.

Restricting j to only 8 values has the advantage that each of these powers can have its own routine in hardware and also helps in bitsliced implementations. We checked that none of the $(s^j + 1)$ has a small order modulo ℓ — otherwise this could lead to small cycles — and we also checked that there is no linear combination with small coefficients among these 8 values. The shortest vector in the lattice spanned by the logarithms of $(s^j + 1)$ modulo ℓ has four-digit coefficients which makes it highly unlikely that any walk will enter a cycle.

Expected number of distinguished points. Each point has probability almost exactly $\left(\binom{131}{34} + \binom{131}{32} + \binom{131}{30} + \cdots\right)/2^{130} \approx 2^{-25.27}$ of being a distinguished point. This means that a walk takes on average $2^{25.27}$ iterations to find a distinguished point. Recall that $2^{60.9}$ iterations are needed to find a collision; thus we expect to use $2^{60.9-25.27} = 2^{35.63}$ distinguished points. In our compressed representation this amounts to about 850 GB of storage.

We chose the cutoff for the Hamming weight w as large as possible subject to the constraint of not overloading the servers; using, e.g., $w \leq 36$ for the distinguished points would increase the number of distinguished points by a factor of 8, exceeding the capacity of our servers. See Section 9, Appendix C, and Appendix D for further discussion of the server load. The advantages of choosing a large cutoff are that it reduces the latency before we detect a collision and that it reduces the number of iterations wasted when a computation stops (whether because a collision is found or because the cluster is needed otherwise). The latter is particularly an issue for highly parallel platforms such as GPUs (see Section 5) which run millions of iterations in parallel.

Benefits of recomputation. Standard practice in ECDLP computations is for clients to report each distinguished point as a linear combination of P and Q. Consider, for example, [BKK+09, Appendix A], which describes a successful computation of a 112-bit prime-field

ECDLP this year using 200 Cell processors at a single site. Each distinguished point was reported and stored as "$4 \cdot 112$ bits": 224 bits for the point (x, y), and 224 additional bits for the expression of the point as a linear combination of P and Q. Scaling to ECC2K-130 would require more than 64 bytes for each distinguished point, much more than the 128 bits we use.

We emphasize that our clients do not monitor the number of occurrences of $\sigma^3 + 1$, $\sigma^4 + 1$, etc., and do not report any linear-combination information. This noticeably reduces the amount of storage handled inside the main client loop, and has much larger effects on network traffic, server storage, etc. (See Appendix D for details of how each 128-bit packet is communicated to the servers; note that we are generating an order of magnitude more distinguished points than were used in [BKK+09], and we are communicating them through the Internet rather than through a local network.)

This client-side streamlining imposes two minor costs. First, once a client has found a distinguished point, it must briefly pause to replace that point by a new point generated from a new seed; the client cannot simply continue iterating past the distinguished point. This does not take much time but does require some extra care in the implementations. Second, the server has to recompute distinguished points — but only for the occasional hashes that collide. This has a negligible effect on the total cost of the computation.

Server storage is not a new issue in this context. Gallant, Lambert, and Vanstone in [GLV00, Section 3] proposed (for Koblitz curves) recording "distinguished tuples" that take "roughly half as much space as algorithms where complete points are stored, or roughly two-thirds of the space in algorithms storing only the x-coordinates of points (or points in compressed form)." The Gallant–Lambert–Vanstone tuples include a client identifier, the x-coordinate of a distinguished point, and an integer modulo the group order, overall consuming more than 32 bytes for ECC2K-130. Our reports are under half the size.

3 Efficient field arithmetic

Our iteration function, defined in Section 2, requires one normal-basis Hamming-weight computation, one application of σ^j where j is between 3 and 10, and one elliptic-curve addition.

In affine coordinates the elliptic-curve addition costs $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S} + 7\mathbf{a}$, where $\mathbf{I}$, $\mathbf{M}$, $\mathbf{S}$, and $\mathbf{a}$ are, respectively, the costs of inversions, multiplications, squarings, and additions in the field. We follow standard practice of handling N iterations in parallel and batching $N\mathbf{I}$ into $1\mathbf{I} + (3N - 3)\mathbf{M}$ ("Montgomery's trick" [Mon87]), where N is chosen per platform to balance communication costs, storage, etc. The repeated Frobenius map σ^j costs at most $20\mathbf{S}$ when computed as a series of squarings. To summarize, each iteration costs at most $(1/N)(\mathbf{I} - 3\mathbf{M}) + 5\mathbf{M} + 21\mathbf{S} + 7\mathbf{a}$ plus a Hamming-weight computation in normal basis.

This section shows that all of these computations can be carried out using a straight-line (branchless) sequence of $71529 + (67600/N)$ bit operations; e.g., 72829 bit operations for $N = 52$. To put this in perspective, observe that one schoolbook multiplication of two 131-bit polynomials uses 34061 bit operations (131^2 ANDs and 130^2 XORs).

The next several sections report the speeds that our implementations have achieved on various platforms. Beware that bit operations are only a simplified model of real-world performance; our implementations often sacrifice some bit operations to save space, reduce communication costs, etc. The model is nevertheless useful: each of the major optimizations described in this section is reused in some, although not always all, of our implementations.

Fast polynomial multiplication. We reuse polynomial multipliers from `http://binary.cr.yp.to/m.html`; those multipliers integrate various improvements to Karatsuba's method,

Toom's method, etc. announced in a Crypto 2009 paper [Ber09]. The 131-bit multiplier uses 11961 bit operations to compute the product $h_0 + h_1 z + \cdots + h_{260} z^{260}$ of two polynomials $f_0 + f_1 z + \cdots + f_{130} z^{130}$, $g_0 + g_1 z + \cdots + g_{130} z^{130}$ with coefficients in $\mathbf{F}_2$.

Fast normal-basis multiplication. We could (and in some implementations do) use a standard polynomial-basis representation for all elements of $\mathbf{F}_{2^{131}}$, converting to normal basis only for Hamming-weight computation. However, we obtain smaller bit-operation counts by instead using a normal-basis representation.

The advantages of normal basis are obvious and well known: the 21**S** are free, and the conversion to normal basis is free. However, the conventional wisdom is that these advantages are overwhelmed by the slowness of normal-basis multiplication. Harley used polynomial basis for the successful attacks on ECC2K-95 and ECC2K-108, and reported that it was much faster than normal basis.

We use a type-2-normal-basis multiplication algorithm published by Shokrollahi in [Sho07, Section 4] and by von zur Gathen, Shokrollahi, and Shokrollahi in [vzGSS07]. This algorithm is known to be *asymptotically* twice as fast as the algorithms analyzed in [vzGN05], but does not yet appear to be widely known among implementors. We give a concise description of this algorithm in the special case $\mathbf{F}_{2^{131}}$, including some new improvements that we found. We refer to this algorithm as the Shokrollahi multiplier.

The central observation in [Sho07, Section 4] is that if

$$f_0 + f_1\left(b + \frac{1}{b}\right) + f_2\left(b + \frac{1}{b}\right)^2 + f_3\left(b + \frac{1}{b}\right)^3 = g_0 + g_1\left(b + \frac{1}{b}\right) + g_2\left(b^2 + \frac{1}{b^2}\right) + g_3\left(b^3 + \frac{1}{b^3}\right)$$

and

$$f_4 + f_5\left(b + \frac{1}{b}\right) + f_6\left(b + \frac{1}{b}\right)^2 + f_7\left(b + \frac{1}{b}\right)^3 = g_4 + g_5\left(b + \frac{1}{b}\right) + g_6\left(b^2 + \frac{1}{b^2}\right) + g_7\left(b^3 + \frac{1}{b^3}\right)$$

then

$$\begin{aligned} f_0 + f_1\left(b + \frac{1}{b}\right) + f_2\left(b + \frac{1}{b}\right)^2 + f_3\left(b + \frac{1}{b}\right)^3 \\ + f_4\left(b + \frac{1}{b}\right)^4 + f_5\left(b + \frac{1}{b}\right)^5 + f_6\left(b + \frac{1}{b}\right)^6 + f_7\left(b + \frac{1}{b}\right)^7 \\ = g_0 + (g_1 + g_7)\left(b + \frac{1}{b}\right) + (g_2 + g_6)\left(b^2 + \frac{1}{b^2}\right) + (g_3 + g_5)\left(b^3 + \frac{1}{b^3}\right) \\ + g_4\left(b^4 + \frac{1}{b^4}\right) + g_5\left(b^5 + \frac{1}{b^5}\right) + g_6\left(b^6 + \frac{1}{b^6}\right) + g_7\left(b^7 + \frac{1}{b^7}\right). \end{aligned}$$

Converting from coefficients of $1, b + 1/b, (b + 1/b)^2, \ldots, (b + 1/b)^7$ to coefficients of $1, b + 1/b, b^2 + 1/b^2, b^3 + 1/b^3, \ldots, b^7 + 1/b^7$ can thus be done with two half-size conversions and three XORs: first convert f_0, f_1, f_2, f_3 to g_0, g_1, g_2, g_3; separately convert f_4, f_5, f_6, f_7 to g_4, g_5, g_6, g_7; and then add g_7 to g_1, g_6 to g_2, and g_5 to g_3. The inverse, converting from coefficients of $1, b + 1/b, b^2 + 1/b^2, \ldots, b^7 + 1/b^7$ to coefficients of $1, b + 1/b, (b + 1/b)^2, \ldots, (b + 1/b)^7$, has exactly the same cost. These conversions are extremely efficient.

Instead of splitting 8 into $(4, 4)$ one can split $2k$ into (k, k) where k is any power of 2, or more generally $k + j$ into (k, j) where k is any power of 2 and $1 \leq j \leq k$. This splitting can and should be applied recursively.

Now consider the type-2 normal basis $b+1/b$, b^2+1/b^2, b^4+1/b^4, etc. of $\mathbf{F}_{2^{131}}$, where b is a primitive 263rd root of 1. The normal-basis multiplier works as follows:

- Permute each input, obtaining coefficients of $b+1/b$, b^2+1/b^2, b^3+1/b^3, ..., $b^{131}+1/b^{131}$. Note that the order of 2 modulo 263 is 131 (as required for a type-2 normal basis).
- Apply the inverse conversion (explained above) to each input, obtaining coefficients of $1, b+1/b, (b+1/b)^2, \ldots, (b+1/b)^{131}$.
- Use the best available polynomial multiplier, obtaining the product as coefficients of $1, b+1/b, (b+1/b)^2, \ldots, (b+1/b)^{262}$. This might appear to be a size-132 polynomial multiplication, but we point out that the inputs always have first coefficient 0, so a size-131 polynomial multiplication suffices. More generally, the "$M(n+1)$" in [vzGSS07, Theorem 8] can be improved to $M(n)$.
- Apply the conversion to the output, obtaining coefficients of $1, b+1/b, b^2+1/b^2, b^3+1/b^3, \ldots, b^{262}+1/b^{262}$.
- Discard the first coefficient. A more expensive step appears in [vzGSS07], adding the first coefficient to many other coefficients, but we point out that the coefficient is always 0, rendering those computations unnecessary.
- Convert to the basis $b+1/b$, b^2+1/b^2, b^3+1/b^3, ..., $b^{131}+1/b^{131}$ by adding appropriate pairs of coefficients: for example, $b^{200}+1/b^{200}$ is the same as $b^{63}+1/b^{63}$.
- Permute back to the usual normal basis.

Our implementation of normal-basis multiplication for $\mathbf{F}_{2^{131}}$ uses just 1559 bit operations plus a size-131 polynomial multiplication, for a total of 13520 bit operations. This is still marginally more expensive than polynomial-basis multiplication, but the benefits now outweigh the costs.

We comment that this multiplier should be of interest in other applications. Existing evaluations of normal-basis speed clearly need to be revisited.

Fast iterations. Computing the Hamming weight of a 131-bit x-coordinate takes just 654 bit operations, including the cost of comparing the weight to a distinguished-point cutoff (producing a 0 or 1). We use the obvious recursive approach here: for example, we compute a 4-bit sum of 15 bits by recursively computing two 3-bit sums of 7 bits each and then using a standard 3-bit full adder to add those 3-bit sums to the remaining bit.

This computation produces the weight of x in the form $128w_7+64w_6+32w_5+16w_4+8w_3+4w_2+2w_1+w_0$, where $w_0=0$ because all weights are even. The next step in the iteration is to compute x^{2^j} where $j=4w_3+2w_2+w_1+3$. This takes just 1179 bit operations (without any conditional branches): computing $x_3=x^8$ is free; computing $x_4=x_3^{2^{w_1}}=x_3+w_1(x_3^2-x_3)$ takes 393 bit operations; computing $x_6=x_4^{4^{w_2}}=x_4+w_2(x_4^4-x_4)$ takes 393 bit operations; and computing $x^{2^j}=x_6^{16^{w_3}}=x_6+w_3(x_6^{16}-x_6)$ takes 393 bit operations. Similarly, computing y^{2^j} takes 1179 bit operations. Note that we do not have to spend any bit operations computing j.

If inversion is implemented by Fermat/Itoh–Tsujii, using a minimal-length addition chain in the exponents $2^1-1=1, 2^2-1, 2^4-1, 2^8-1, 2^{16}-1, 2^{32}-1, 2^{64}-1, 2^{128}-1, 2^{130}-1$, then $1\mathbf{I}=8\mathbf{M}+130\mathbf{S}=108160$. Finally, $\mathbf{a}=131$ and so the overall iteration cost is $(1/N)(108160-3\cdot 13520)+5\cdot 13520+7\cdot 131+654+2\cdot 1179=71529+(67600/N)$ bit operations. These operation counts have been computer-verified.

4 Core 2 Extreme Q6850 CPU, 4 cores, 3 GHz: 22.45 million iterations/second

The Core 2 line of microprocessors has been sold since 2006 and is now widely available in computer clusters. Our software implementation of the ECC2K-130 attack for the Core 2 takes just 529 cycles per iteration on a single core of a "3 GHz" (actually 2.997 GHz) Core 2 Extreme Q6850 CPU. There is only a minor slowdown from running the implementation in parallel on all four cores: each core takes 534 cycles per iteration, so the CPU as a whole performs 22.45 million iterations per second.

A 2.394 GHz Core 2 Quad Q6600 produces essentially the same cycle counts, 533 cycles per iteration per core when all four cores are running in parallel, although the lower clock speed means that the CPU performs only 18 million iterations per second. Our software also performs reasonably well on other popular microarchitectures, for example using 637 cycles per iteration per core on a 2.2 GHz Phenom 9550 when all four cores are running in parallel, but at this point the software has been tuned only for a Core 2.

For comparison, Harley's software for the successful ECC2K-95 and ECC2K-108 attacks performed, respectively, "up to 177 K iterations per second" on a 0.6 GHz Alpha 21164 workstation and "454 k operations per second" on a 0.75 GHz Alpha 21264 workstation. Of course, the 49× speed ratio between 22.45 million iterations/second and 0.454 million iterations/second is partially explained by the 4× ratio between 3 GHz and 0.75 GHz, and by the 4× ratio between 4 cores and 1 core, but there is still a 3× gap between Harley's 1651 Alpha cycles per iteration for ECC2K-108 and our 533 Core 2 cycles per iteration for ECC2K-130, even without accounting for the difference in size between 109 bits and 131 bits.

To simplify the comparison we ran Harley's ECC2K-108 software and found that it took 1800 Core 2 cycles per iteration. We also wrote our own polynomial-basis ECC2K-108 software and found that it took fewer than 500 Core 2 cycles per iteration, almost a 4× speedup. This speedup is not the result of any compromise in the iteration function: in fact, according to the analysis in Appendix B, our function requires slightly fewer iterations than Harley's when scaled.

Lavasani and Mohammadi stated recently in [LM08] that their software would break ECC2K-130 in "less than 2 years using 20000 partially active computers." For comparison, to break ECC2K-130 in an expected time of 2 years, our software would need only 1520 CPUs. The phrase "partially active" is undefined but seems unlikely to explain a factor of 13 in the number of CPUs.

The importance of bitslicing. The standard software representation of a 131-bit binary-field element is as a sequence of 131 bits in polynomial basis. The sequence of bits is then packed into a short sequence of CPU words. To multiply x by y one looks up (typically) 4 bits of x in a table of precomputed multiples of y, looks up the next 4 bits of x, etc.

Hankerson, Karabina, and Menezes in [HKM08, Section 6] report that a state-of-the-art implementation takes 270 cycles per multiplication in $\mathbf{F}_{2^{127}} = \mathbf{F}_2[z]/(z^{127} + z^{63} + 1)$ on a 3.16 GHz Xeon 5460; this CPU has the same microarchitecture as the Core 2. For comparison, our software takes only 94 Core 2 cycles per multiplication in the larger field $\mathbf{F}_{2^{131}}$.

To achieve these speeds we reuse, and refine, the techniques that were used in [Ber09] to set new speed records for elliptic-curve cryptography. There is a productive synergy between (1) fast CPU vector operations, (2) "bitsliced" data structures, (3) techniques to minimize bit operations for polynomial multiplication, and (4) the normal-basis multiplication method

discussed in Section 3. The software speedups from Karatsuba, Toom, etc. are much larger in a bitsliced representation than in a non-bitsliced representation (as pointed out in [Ber09]), and the conversions discussed in Section 3 are much less expensive in a bitsliced representation than in a non-bitsliced representation.

Bitslicing is a simple matter of transposition: 128 separate field elements $x_0, x_1, \ldots, x_{127} \in \mathbf{F}_{2^{131}}$ are represented as 131 vectors $X_0, X_1, \ldots, X_{130}$, where the jth bit of X_i is the ith bit of x_j. "Logical" vector operations act on bitsliced inputs as 128-way bit-level SIMD instructions: e.g., a vector XOR of X_0 with X_{32} simultaneously computes, for all j, the XOR of the 0th bit of x_j with the 32nd bit of x_j. Given any decomposition of $\mathbf{F}_{2^{131}}$ multiplication into a straight-line sequence of bit XORs and bit ANDs, one can carry out 128 multiplications on 128 pairs of bitsliced inputs in parallel by performing the same series of vector XORs and vector ANDs. The same comment applies to more complicated operations, such as our iteration function.

Beware that bitslicing is not a panacea. Simple operations such as branches and variable-index table lookups become quite expensive operations in a bitsliced representation. Purely bitsliced computations also require 128 times as much parallelism as non-bitsliced computations; in the ECDLP context, any increase in parallelism forces a corresponding increase in the number of distinguished points transmitted to the central servers.

Bottlenecks. Recall from the previous section that one can apply the iteration function to a batch of N points using a total of $71529N + 67600$ bit operations. One can therefore apply the iteration function to a batch of $128N$ points in bitsliced representation using a total of $71529N + 67600$ 128-bit vector operations.

Our software takes $N = 48$, handling 6144 points in parallel while fitting all computations into half a megabyte of RAM. The software also supports checkpointing the computation into just 252 KB of disk space to allow a pause and a subsequent resume.

The Core 2 has 3 vector ALUs, each capable of performing one 128-bit vector operation per cycle, so one can hope to carry out these $71529N + 67600$ vector operations in just $(71529N + 67600)/3$ cycles; in other words, one can hope to perform each iteration in just $(71529N + 67600)/(3 \cdot 128N) \approx 186 + 176/N$ cycles. Our 533 cycles per iteration are a factor 2.8 larger than the desired $186 + 176/48 \approx 190$. Two difficulties discussed in [Ber09] are the limited Core 2 register set (just 16 vector registers), forcing frequent loads and stores (which can occur only once per cycle), and the lack of three-operand instructions, forcing frequent copies. Another difficulty is that our main loop is considerably more complicated than the main loop in [Ber09], putting extra pressure on the 32 KB Core 2 instruction cache.

We ended up almost completely rewriting the polynomial-multiplication software from [Ber09], sacrificing 1668 bit operations per multiplication so that the most frequently used field-arithmetic code would fit into 20 KB. This sacrifice means that our software actually uses $81451+1/12$ bit operations per iteration, requiring at least 212 cycles; this count has also been computer-verified. At the same time we found some scheduling improvements both inside and outside the multipliers; we found, for example, a way to fit the size-32 conversion discussed in the previous section into 16 registers with no spills. These improvements increased the overall ALU utilization to 1.2 vector operations per cycle, compared to 1.1 in [Ber09], although 1.2 obviously falls far short of the desired 3.0.

5 GTX 295 Video Card: 2 GT200b GPUs, 1.242GHz: 25.12 million iterations/second

Our GPU implementation runs a complete iteration function on 7 864 320 points in parallel in 313 ms. There have been several articles [SG08,BCC+09b,BCC+09a] on exploiting GPUs for elliptic-curve arithmetic, but they all involve large-integer rather than binary-field arithmetic.

The design philosophy behind the current generation of Graphics Processing Units (GPUs) is to allocate the enormous transistor budget afforded by Moore's law to raw arithmetic units. As an example, an NVIDIA GTX 295 video card contains two GT200b chips each holding 240 ALUs. Each ALU can potentially do one fused 32-bit multiply-and-add per cycle at 1.242 GHz. This makes massively parallel computations run faster, rather than speed up sequential programs, which is what modern CPUs are doing to most desktop applications such as web browsing and word processing. GPUs are hence very cost-effective for high arithmetic throughput, including, e.g., big-integer arithmetic [BCC+09b].

However, arithmetic in $\mathbf{F}_{2^{131}}$ is more challenging. There are many limitations on the hardware level: The 120 subsidiary multipliers on the GT200b do not dispatch logical operations; while GPUs seem to have huge bandwidth, in reality it averages only 1 byte per cycle per ALU; latencies are huge (200+ cycles from main card or "device" memory, around 20 cycles from fast or "shared" memory, or even from registers). Binary-field arithmetic means logical operations with much data movement (hence a low compute-to-memory-access ratio) and is intrinsically difficult on this platform. The most widely used GPU development toolchain, NVIDIA's CUDA [GPU08] is still far from mature which adds extra complications.

We use the normal-basis representation from Section 3. The implementation is bitsliced with a bit vector size of $60 \times 64 \times 32 \times 32 = 3\,932\,160$ on an NVIDIA GT200 series GPU. This is a rather large vector size which goes against the traditional wisdom in parallel computing which suggests the use of minimal vector sizes in order to relieve pressures on registers and fast on-die memories [VD08]. To understand the rationale behind this design choice, we first briefly explain CUDA's programming model and GPU architecture. In CUDA, the GPU is used as a coprocessor to the CPU for massively data-parallel computations, which are executed by a grid of GPU *threads* which must run the same program (the *kernel*). This is the SPMD (single-program-multiple-data) programming model, similar to SIMD (single-instruction-multiple-data) but with more flexibility (such as in changing of data size on a per-kernel-launch basis, or deviation from SIMD to MIMD at a performance penalty).

This SPMD paradigm also exposes a lot of the inner workings of GPU architecture to the programmers. For example, the scalar ALUs on a CUDA GPU are divided into groups of 8, called multiprocessors (MPs). The minimal hardware scheduling unit is a *warp* of 32 threads, and at least two active warps are needed on each MP at any time. Hence, any kernel should launch a minimum of 64 threads on each of the 30 MPs on a GT200b GPU. During most of the execution we actually run 4 096 threads per MP. This is to fit the use of Montgomery inversion with a batch size of 32. On normal CPUs, one would use a larger batch. Here we are limited by the amount of available device memory on a GT200b.

Despite these limitations in hardware and language/compiler we are able to complete each batch of iterations (including checking for distinguished points) in 313 ms. Ergo, our raw throughput is $30 \times 128 \times 32 \times 32 \times 2/0.313 = 25.12$ million point-iterations per second.

This huge batch size has another implication: We do not immediately replace the seeds that have reached distinguished points because we would need to swap data in and out of the GPU every 10 iterations. Instead, we run 2^{16} iterations before we swap in new seeds. This

number is chosen such that each time about 0.2% of the seeds reach distinguished points and get replaced. The efficiency loss is small, roughly 0.1%.

6 Cell CPU, 6 SPEs, 1 PPE, 3.2 GHz: 27.67 million iterations/second

The Cell Broadband Engine (Cell) [Hof05], [Son06] is a microprocessor architecture that consists of one central 64-bit PowerPC core and 8 additional cores called Synergistic Processor Elements (SPEs). It can be found in different blade servers of the QS series by IBM, extension cards, and the Sony PlayStation 3 gaming console. The first generation of the latter is the cheapest option to get access to a Cell. When running Linux, six out of the eight SPEs are available to the programmer since one SPE is disabled and one SPE is reserved by the system.

The main computational power of the processor is in the SPEs, so we target these cores with our implementation. Each SPE has a register file with 128 128-bit vector registers, and has two pipelines executing disjoint sets of instructions. Most arithmetic and all bit-logical instructions go to the "even" pipeline while loading and storing go to the "odd" pipeline. Instructions are executed in order. Typical optimization techniques for in-order CPUs, such as function inlining and loop unrolling, are limited by the fact that the code segment, heap segment, data segment, and stack segment together have to fit in a local storage (LS) of just 256 kilobytes. In particular bitsliced implementations are affected by this restriction of local storage because they require a much higher level of parallelism and therefore more storage for inputs, outputs, and intermediate values.

We investigated the performance of both bitsliced and non-bitsliced implementations for the Cell. It turned out that a bitsliced implementation using normal-basis representation for elements of $\mathbf{F}_{2^{131}}$ yields the best performance.

Non-bitsliced implementation. Our non-bitsliced implementation uses polynomial basis for $\mathbf{F}_{2^{131}}[z]/(z^{131}+z^{13}+z^2+z+1)$ and represents each 131-bit polynomial using two 128-bit vectors. In order to use 128-bit lookup tables and to get 16-bit-aligned intermediate results we break the multiplication of two polynomials A, B into parts as follows:

$$A = A_l + A_h \cdot z^{128} = \widetilde{A}_l + \widetilde{A}_h \cdot z^{121},\ B = B_l + B_h \cdot z^{128} = \widetilde{B}_l + \widetilde{B}_h \cdot z^{15},$$
$$C = A \cdot B = \widetilde{A}_l \cdot B_l + \widetilde{A}_l \cdot B_h \cdot z^{128} + \widetilde{A}_h \cdot \widetilde{B}_l \cdot z^{121} + \widetilde{A}_h \cdot \widetilde{B}_h \cdot z^{136}.$$

For the first two multiplications a four-bit lookup table and for the third and fourth a two-bit lookup table are used. The squaring is implemented by inserting a 0 bit between each two consecutive bits of the binary representation. In order to hide latencies two steps are interleaved aiming at filling both pipelines. The squarings needed for $\sigma^j(P_i)$ are implemented using lookup tables for $3 \leq j \leq 10$ and take for any such value of j a constant number of cycles.

The optimal number of concurrent walks is as large as possible, i.e., such that the executable and all the required memory fit in the LS to reduce the cost of the inversion. In practice 256 walks are processed in parallel. The non-bitsliced implementation computes 16.72 million iterations/second using the six available SPEs in the PlayStation 3 video game console.

Bitsliced implementation. The bitsliced implementation is based on the Core 2 implementation described in Section 4. That implementation was in C++, involving a large overhead in the size of binaries, so we first ported it to C and then optimized all speed-critical parts in assembly.

Our implementation can fit all data and code into the LS using a batch size of $N = 14$ for Montgomery inversions. Each of the 14 computations requires storage for only 4 finite-field elements x, y, x_2, and r. This is achieved by first sequentially computing $x_2 \leftarrow x + \sigma^j(x)$ for all of the 14 x-coordinates, then using parallel Montgomery inversion to compute $r \leftarrow x_2^{-1}$ for all 14 inputs and then continuing with sequential computations to update x and y. Note that the sequential computations need storage for 3 more finite-field elements for intermediate values, so in total storage for $4 \cdot N + 3$ field elements is needed.

With a batch size of 14 we need 666 bit-logical operations per iteration. (The bit-operation count in Section 3 is about 10% smaller; we sacrificed some bit operations to improve code size, as in Section 4.) We thus have a lower bound of 666 cycles per iteration. Our code runs at 789 cycles per iteration, only 1.18 times this theoretical lower bound, and computes 24.33 million iterations per second on the 6 SPEs of a PlayStation 3.

To increase the batch size further to 512 we are using direct memory access (DMA) calls to swap the active set of variables between the main memory and the local storage. This decreases the number of bit operations per iteration to 625, and the number of cycles per iteration to 749. Running the code on all 6 SPEs in parallel does not involve any penalty. This means that we can compute 25.63 million iterations per second on the six SPUs of a PlayStation 3.

Utilizing the PPE. The PPE of the Cell processor is used to compute the input for the iteration function. This means that the core is busy only during a short intial phase. To also utilize the computing power of the PPE, we ported the Core 2 implementation to the 64-bit Power architecture, replacing SSE instructions with Altivec instructions.

We run two threads in parallel on the PPE, each requiring 3140 cycles per iteration, so the PPE computes 2.04 million iterations per second. In total we can thus compute 27.67 million iterations per second on a PlayStation 3.

Comparison with previous results. To the best of our knowledge there were no previous attempts to implement fast binary-field arithmetic on the Cell. The closest work that we are aware of is [BKM09], in which Bos, Kaihara and Montgomery solved an elliptic-curve discrete-logarithm problem using a cluster of 200 PlayStations. Comparing these two ECDLP attacks is difficult for several reasons: (1) [BKM09] attacks a prime field, while we attack a binary field; (2) the field size in [BKM09] is $(2^{128} - 3)/(11 \cdot 6949) \approx 2^{112}$, while the field size in this paper is 2^{131}; (3) textbook optimizations reduce the [BKM09] iteration count by a factor $\sqrt{2}$ (for negation), while they reduce our iteration count by a factor $\sqrt{2 \cdot 4 \cdot 131}$; (4) both [BKM09] and this paper choose streamlined iteration functions, reducing the cost of each iteration but also increasing the number of iterations required. The increase is more severe in [BKM09], a factor of $\sqrt{2}$, while our increase is only 1.07; see Section 2 and Appendix B.

[BKM09] reports 453 cycles per iteration for 112 bits; in other words, $453 \cdot \sqrt{2} \approx 641$ cycles per textbook iteration for 112 bits. We follow [BKK+09, Section 3.2] in extrapolating quadratically to larger sizes: the techniques of [BKM09] would take $453 \cdot \sqrt{2} \cdot (131/112)^2 \approx 876$ cycles per textbook iteration for 131 bits. One can criticize this quadratic extrapolation, since some parts of the arithmetic in [BKM09] scale better than quadratically, and in fact a "more precise extrapolation" in [BKK+09, Appendix A.5] yields slightly smaller estimates for 160-bit prime fields; on the other hand, the techniques in [BKM09] are word-oriented and would be considerably slower for 131 bits than for 128 bits.

For comparison, we use only $749 \cdot 1.07 \approx 801$ cycles per textbook iteration for 131 bits, significantly fewer than 876 cycles. We emphasize that [BKM09] performs arithmetic in *prime*

fields, taking advantage of the Cell's fast parallel integer multipliers, while we perform arithmetic in *binary* fields, without any binary-polynomial-multiplication instructions in the CPU.

7 Spartan-3 XC3S5000-4FG676 FPGA: 33.67 million iterations/second

A cluster of tens or hundreds of FPGA chips can be cobbled together to tackle this problem in parallel. Several such designs have been implemented in the academic community [KPP+06]. Our aim is to introduce a new cluster of 128 Xilinx Spartan-3 XC3S5000-4FG676 chips to carry out this attack; this particular chip is attractive for its low cost (about $100 as of this writing). The FPGA cluster will become available in November 2009.

In contrast to our software implementations which aim solely for highest throughput from a program, the design space for an FPGA aims to minimize the time-area product. Reducing the area required allows us to instantiate several engines per chip. As with the software implementations, performance of the multiplier is of paramount importance. We evaluated several different multiplier architectures. Table 1 gives the Post-Place & Route results of area and maximum frequency.

Design I: Polynomial-basis. In order to perform the step function, the design uses a digit-serial multiplier, a squarer, a base converter to generate the normal-basis representation of x coordinate and a Hamming weight counter. The design consumes 3656 slices, including 1468 slices for the multiplier, 75 slices for the squarer, 1206 slices for the base conversion, 117 slices for Hamming weight counting.

Design II: Kwon. This design includes a digit-serial normal-basis multiplier, a squarer and a Hamming weight counter. The digit-serial multiplier we use is derived from Kwon's systolic multiplier [Kwo03]. The multiplier takes input in permuted Type-II ONB and the output stays in the same basis. It has a highly regular structure. The drawback is that the multiplier is much larger in area than a polynomial-basis multiplier with the same digit size. The multiplier alone consumes 2093 slices.

Design III: Novotný. This design includes a digit-serial normal-basis multiplier, a squarer, and a Hamming weight counter. Our digit-serial multiplier is derived from Novotný's GL multiplier [NS06b]. By carefully re-mapping the data flow through the multiplier, Novotný, et al. propose a highly regular digit-serial architecture. It arranges the partial products in a shift register. In each stage, the choice of terms evaluated and then added to the partial products is switched based on which partial product is currently present. We implemented the Linear Multiplier variant of this idea which simplifies the switching logic [NS06a].

Table 1 represents a preliminary exploration of the design space, including as a baseline, polynomial-basis with classic conversion, and the Kwon and Novotný algorithms. In the case of polynomial-basis and Kwon, we have implemented the entire Pollard's rho iteration, while for Novotný we have implemented only the multiplier. Since Kwon and Novotný only differ in multiplier and squarer, we can accurately estimate the overhead cost for Novotný. Our current top performing engine (Novotný) can be instantiated 20 times in one FPGA chip, and requires 594 nsec per iteration per engine. Each chip thus performs 33 670 033 iterations per second and a cluster of 128 chips attains a speed of 4 309 764 309 iterations per second.

We also have an FPGA implementation of the Shokrollahi multiplier. The sequence of polynomial splits leads to a long critical path, so achieving a competitive clock rate requires pipelining. Although this choice results in higher area consumption, a full pipeline can yield one product per clock cycle at 120 MHz. The Shokrollahi multiplier's time-area product, 1 cycle $\times$ 4664 slices, is much smaller than Novotný's, 10 cycles $\times$ 1295 slices. Pipelining the

multiplier requires rearchitecting the rest of the iteration to keep the multiplier's pipeline full. This implementation is currently under development, and when completed should outperform all previous results from the literature.

Note that the current designs are only the core of step function. The other routines such as generating the starting point from a random seed and normalizing distinguished points are performed on the host PC that controls the FPGA cluster.

Comparison with previous results. To our knowledge, this is the first FPGA implementation using fast normal-basis multiplication to attack the ECDLP. As a point of comparison, we look to Meurice de Dormale, et al. [MdDBQ07]. They do not specifically target Koblitz curves and do not work with orbits at all, so their implementation does not even take advantage of the negation. Furthermore, they implement polynomial-basis arithmetic. Our iteration function is more complicated than theirs but requires far fewer iterations for the Koblitz curve.

For this application, the efficiency of multiplier or inverter alone does not give much information. Indeed, the iteration function, the choice of representations, the efficiency of the arithmetic unit and the memory architecture decide the throughput of distinguished points. Since their design is mounted on a different platform (Xilinx XC3S1200E), a fair comparison is difficult. We use the number of iterations per second to give an intuitive comparison. Their polynomial-basis engine, using a pipelined Karatsuba multiplier, takes 7900 slices on a Xilinx XC3S1200E FPGA and performs 10^7 iterations per second. By contrast, ours takes 1577 slices on a Xilinx XC3S5000 and performs 1.684×10^6 iterations per second. On a per-chip basis, given the fact that XC3S5000 is about four times larger than XC3S1200E, this amounts to 40 million iterations per second for theirs and 33.6 million for ours. Thanks to our improved iteration function, our speedup over the previous work is a factor of $\sqrt{2 \times 131}(33.6/40) = 13.6$.

8 16 mm² ASIC, 90nm process: estimated 800 million iterations/second

This section presents the performances and the area cost we estimated for an ASIC implementation of one iteration step function. Furthermore, based on these estimations, we calculated the effort needed to break the whole Certicom challenge ECC2K-130 using multiple instances of the proposed circuit.

Concerning the selection of the target design, the considerations and guidelines previously discussed for FPGA still hold for the ASIC case, thus here we considered the two complete implementations of Section 7, namely the one based on polynomial-basis with classic conversion and the normal-basis one based on Kwon's algorithm.

Table 2 summarizes the area and timing performance for the two designs. The estimations for the polynomial-basis are reported in the first row, while the ones for the normal-basis are in the second row. These results were obtained using the UMC L90 CMOS technology and the Free Standard Cell library from Faraday Technology Corporation characterized for HS-RVT (high speed regular Vt) process option with 8 metal layers. The specific technology was selected because of its wide availability in academic environments. The synthesis was

Table 1. Cost for one Pollard's rho iteration for various FPGA multipliers

	Digit-size	#slice	#BRAM	Freq. [MHz]	Cycles per step	Delay [ns]
Design I: Polynomial basis	22	3656	4	101	71	704
Design II: Type-II ONB	13	2578	4	125	81	648
Design III: Type-II ONB	13	1577	4	111	66	594

performed using the tool Synopsys design compiler 2008.09. Both the designs were synthesized at a frequency of 1 GHz.

Table 2. Cost for one Pollard's rho iteration on ASIC

	Area [mm^2]	GE	Computation block area [mm^2]	Computation block GE	Cycles	Frequency [GHz]
Polynomial basis	0.185	55K	0.120	35K	71	1
Normal basis	0.138	41K	0.075	21K	80	1

We consider using a standard 16 mm^2 die in the 90nm CMOS technology using a multi-project wafer (MPW) production in prototype quantities. The cost of producing and packaging 50–250 of such ASICs is estimated to be less than 60000 EUR. The net core area of such a die is around 12 mm^2 after the overhead of I/O drivers and power distribution is considered. We leave a healthy margin of 2 mm^2 for the necessary PLL and I/O interface (including memory for storing results), and consider that we have 10 mm^2 available for the core.

Note that the normal computation block for the normal-basis realization requires 0.075 mm^2 of area and has a latency of 80 clock cycles. It would be possible to instantiate 80 cores in parallel in this system, which would occupy around 6 mm^2 of area. Similarly, for the polynomial basis the core is around 0.120 mm^2 and has a latency of 71 cycles. Instantiating 71 cores would require 8.52 mm^2 area.

The above mentioned synthesis results were given for 1 GHz clock speed. Leaving a healthy 20% margin, we could estimate that for both designs 1 ECC operation can be calculated per clock in both cases, reaching a total of 800 million ECC operations per second. The estimation of the complete attack able to break ECC2K-130 in one year would require approximately 69000 million iterations/second. Even our overly pessimistic estimation shows that this performance can be achieved by less than 100 dedicated ASICs, thus making the attack feasible for approximately 60000 EUR.

9 Carrying out the attack

It is clear that ASICs provide the best price-performance ratio for breaking ECC2K-130. However, at the moment the authors of this paper do not have funding to build ASICs, while the authors do have access to existing Core 2 clusters, existing Cell clusters, some otherwise idle GPUs, and even some FPGAs.

Breaking ECC2K-130 takes an average time of $2^{60.9}$ iterations. Our attack implementations, described in previous sections, compute $2^{60.9}$ iterations per year using just 3039 3GHz Core 2 CPUs *or* 2716 GTX 295 GPUs *or* 2466 Cell CPUs *or* 2026 XC3S5000 FPGAs or any combination thereof.

We started running a distributed attack against ECC2K-130 in October 2009. The attack has two parts. First, we are running attack clients on Core 2 clusters, Cell clusters, etc. around the world. Second, we are running servers that, given enough data from the clients, will find the desired discrete logarithm. Appendix C explains the distribution and handling of data and Appendix D describes the communication between clients and servers.

Updates on performance data of actual attack. We have not issued a public call for donations of computer resources; we expect to be able to collect enough computer power from

our own resources, probably enough to break ECC2K-130 in the first half of 2010. See our anonymous web page `http://ecc-challenge.info` for the latest information regarding the progress of the attack.

References

[BCC+09a] Daniel J. Bernstein, Hsieh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Chin Lin, and Bo-Yin Yang. The billion-mulmod-per-second PC. In *Workshop record of SHARCS'09*, pages 131–144, 2009. `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`.

[BCC+09b] Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In *Eurocrypt 2009*, volume 5479 of *LNCS*, pages 483–501, 2009.

[Ber09] Daniel J. Bernstein. Batch binary Edwards. In *Crypto 2009*, volume 5677 of *LNCS*, pages 317–336, 2009. `http://binary.cr.yp.to/edwards.html`.

[BKK+09] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. On the security of 1024-bit RSA and 160-bit elliptic curve cryptography: version 2.1. Cryptology ePrint Archive, Report 2009/389, 2009. `http://eprint.iacr.org/2009/389`.

[BKM09] Joppe W. Bos, Marcelo E. Kaihara, and Peter L. Montgomery. Pollard rho on the PlayStation 3. In *Workshop record of SHARCS'09*, pages 35–50, 2009. `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`.

[BP81] Richard P. Brent and John M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36:627–630, 1981.

[Cer97a] Certicom. Certicom ECC Challenge. `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`, 1997.

[Cer97b] Certicom. ECC Curves List. `http://www.certicom.com/index.php/curves-list`, 1997.

[GLV00] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.

[GPU08] NVIDIA CUDA programming guide version 2.3.1. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`, August 2008.

[Har] Robert Harley. Elliptic curve discrete logarithms project. `http://pauillac.inria.fr/~harley/ecdl/`.

[HKM08] Darrel Hankerson, Koray Karabina, and Alfred Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. Cryptology ePrint Archive, Report 2008/334, 2008. `http://eprint.iacr.org/2008/334`.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA 2005*, pages 258–262. IEEE Computer Society, 2005.

[Kob92] Neal Koblitz. CM-curves with good cryptographic properties. In *Advances in Cryptology – Crypto 1991*, volume 576 of *Lecture Notes in Comput. Sci.*, pages 279–287. Springer-Verlag, Berlin, 1992.

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA – a cost-optimized parallel code breaker. *Lecture Notes in Computer Science*, 4249:101, 2006.

[Kwo03] Soonhak Kwon. A low complexity and a low latency bit parallel systolic multiplier over GF(2^m) using an optimal normal basis of type II. In *IEEE Symposium on Computer Arithmetic - ARITH-16*, pages 196–202, 2003.

[LM08] Ahmad Lavasani and Reza Mohammadi. Implementing a feasible attack against ECC2K-130 Certicom challenge, 2008. Poster abstract from ANTS-8.

[MdDBQ07] Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, pages 378–393. Springer, 2007.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[NS06a] Martin Novotný and Jan Schmidt. General digit width normal basis multipliers with circular and linear structure. In *FPL*, pages 1–4, 2006.

[NS06b] Martin Novotný and Jan Schmidt. Two Architectures of a General Digit-Serial Normal Basis Multiplier. In *Proceedings of 9th Euromicro Conference on Digital System Design*, pages 550–553, 2006.

[Pol78] John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.

[Ser98] Gadiel Seroussi. Compact representation of elliptic curve points over GF(2^n). Technical report, Research Contribution to IEEE P1363, 1998.

[SG08] Robert Szerwinski and Tim Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES 2008*, volume 5154 of *LNCS*, pages 79–99, 2008.

[Sho07] Jamshid Shokrollahi. *Efficient implementation of elliptic curve cryptography on FPGAs*. 2007. Dissertation. `http://hss.ulb.uni-bonn.de/diss_online/math_nat_fak/2007/shokrollahi_jamshid/0960.pdf`.

[Son06] Sony Corporation. *Cell Broadband Engine Architecture, Version 1.01*, 2006. `http://cell.scei.co.jp/pdf/CBE_Architecture_v101.pdf`.

[Tes01] Edlyn Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.

[VD08] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Austin, TX, USA, November 2008.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[vzGN05] Joachim von zur Gathen and Michael Nöcker. Polynomial and normal bases for finite fields. *J. Cryptology*, 18(4):337–355, 2005.

[vzGSS07] Joachim von zur Gathen, Amin Shokrollahi, and Jamshid Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In Claude Carlet and Berk Sunar, editors, *WAIFI*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68. Springer, 2007.

[WZ98] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected Areas in Cryptography*, volume 1556 of *LNCS*, pages 190–200, 1998.

Appendix A: Parameters of the ECC2K-130 challenge

The challenge curve is the Koblitz curve $y^2 + xy = x^3 + 1$ over $\mathbf{F}_{2^{131}}$. The group order is $|E(\mathbf{F}_{2^{131}})| = 4\ell$, where ℓ is the 129-bit prime

$$\ell = 680564733841876926932320129493409985129.$$

The challenge is given with respect to a polynomial-basis representation of $\mathbf{F}_{2^{131}} \cong \mathbf{F}_2[z]/(F)$ with $F(z) = z^{131} + z^{13} + z^2 + z + 1$. Field elements are naturally given as bit strings with respect to this basis; the Certicom challenge represents them as hexadecimal numbers, padded with 0's on the left, by grouping four bits into one hexadecimal number. The base point P and the challenge point Q have coordinates

```
P_x = 05 1C99BFA6 F18DE467 C80C23B9 8C7994AA,
P_y = 04 2EA2D112 ECEC71FC F7E000D7 EFC978BD,
Q_x = 06 C997F3E7 F2C66A4A 5D2FDA13 756A37B1,
Q_y = 04 A38D1182 9D32D347 BD0C0F58 4D546E9A.
```

The challenge is to find an integer k with $Q = [k]P$.

Appendix B: Analysis of the iteration function

This appendix analyzes the number of iterations required by our iteration function. It is easy to find literature on the number of iterations required for perfect random walks and also for adding walks where each direction has the same probability; see, e.g., Teske [Tes01]. Our

iteration function is defined via the Hamming weight of the normal-basis representation of the x coordinates; this weight is not uniformly distributed. Brent and Pollard in [BP81] gave some heuristics for analyzing the randomness in the iteration function of Pollard's rho method for factoring. Our analysis is a refinement of their heuristics.

Our iteration function is a multiplicative iteration function: every step maps an intermediate point P_i to $P_{i+1} = \sigma^j(P_i) \oplus P_i$ which by the equivalence of σ and s on points of order ℓ corresponds to the multiplication $[s^j + 1]P_i$. Our j is limited to $\{3, 4, \ldots, 10\}$ but to justify our choices we will analyze a more general function. Thus assume that r different scalars s_i are used to define the random walk and let p_i be the probability that scalar s_i is used.

Fix a point R, and let S and S' be two independent uniform random points. Consider the event that S and S' both map to R but $S \neq S'$. This event occurs if there are distinct i, k such that the following three conditions hold simultaneously: first, $R = [s_i]S = [s_k]S'$ with $i \neq k$; second, s_i is chosen for S; third, s_k is chosen for S'. These conditions have probability $1/\ell^2$, p_i, and p_k respectively. Summing over all (i, k) gives the overall probability $\left(\sum_{i\neq k} p_i p_k\right)/\ell^2 = \left(\sum_{i,k} p_i p_k - \sum_i p_i^2\right)/\ell^2 = \left(1 - \sum_i p_i^2\right)/\ell^2$. This means that the probability of an immediate collision from S and S' is $\left(1 - \sum_i p_i^2\right)/\ell$, where we added over the ℓ choices of R.

After t iterations there are $t(t-1)/2$ pairs of iteration outputs. If the inputs to the iteration function were independent uniformly distributed random points then the probability of success would be $1-\left(1 - \left(1 - \sum_i p_i^2\right)/\ell\right)^{t(t-1)/2}$ and the average number of iterations before a collision would be approximately $\sqrt{\pi\ell/(2(1 - \sum_i p_i^2))}$. The inputs to the iteration function in Pollard's rho method are not independent but the average number of iterations nevertheless appears to be approximately $\sqrt{\pi\ell/(2(1 - \sum_i p_i^2))}$ for almost all choices of scalars.

In the simple case that all the p_i are $1/r$, the difference from the optimal $\sqrt{\pi\ell/2}$ iterations is a factor of $1/\sqrt{1 - 1/r} \approx 1 + 1/(2r)$. Teske in [Tes01] presents extensive experimental evidence that Pollard's rho method follows the $1/\sqrt{1 - 1/r}$ behavior for adding walks where all the p_i are $1/r$.

For our ECC2K-130 attack, all Hamming weights of x-coordinates of points on E are even, and experiments show that the distribution of even-weighted words of length 131 is close to the distribution of x-coordinates of points. Using the Hamming weight to define the iteration function therefore inevitably introduces an extra factor to the running time of $1/\sqrt{1 - \sum_i \binom{131}{2i}^2/2^{260}} \approx 1.053211$, even if all 66 weights use different scalars s_i.

We designed our step function to use only 8 different values for the scalars. The values are determined by $\mathrm{HW}(x_{P_i})/2 \mod 8$; the distribution of $\sum_i \binom{131}{16i+2k}$ for $0 \leq k \leq 7$ gives probabilities

$$0.1443, 0.1359, 0.1212, 0.1086, 0.1057, 0.1141, 0.1288, 0.1414,$$

giving a total increase of the number of iterations by a factor of 1.069993.

We consider a 7% increase in the number of iterations justified by the simpler, faster iteration function (at most 20 squarings and one point addition to compute P_{i+1} in comparison to the costs of working with canonical representatives and adding walks) in particular since 5% is unavoidable when one uses the Hamming weight.

Appendix C: Central servers

To simplify administration we decided to collect all data at a single site, on a small pool of 8 central servers. The job of each client around the Internet is to compute pairs (s, h) and send those pairs to the central servers. Here s is a random seed, and h is the hash of the corresponding distinguished point.

Each pair (s, h) occupies 16 bytes. The servers each have 460 gigabytes of disk storage allocated to this project, enough room in total for $2^{37.8}$ pairs (s, h), i.e., for the outputs of 2^{63} iterations; our computation has more than a 99.999% chance of finding a collision within 2^{63} iterations. The *expected* volume of data is below one terabyte, and if we limited storage to a terabyte (by discarding old data) then we would not drastically slow down the computation, but at the moment there is no real competition for the disk space on these servers.

The client uses 3 bits of the hash h to decide which server will receive the pair (s, h). The server uses 10 more bits of h to spread data across 1024 buffers in RAM, each buffer consuming one quarter of a megabyte; whenever buffer i fills up, the server appends that quarter-megabyte of data to disk file i and clears the buffer. The $\leq$460 gigabytes of data arriving at the server are thus split into 1024 disk files, each occupying $\leq$460 megabytes. Note that, compared to the total length of the computation, the buffering adds negligible extra latency before data is written to disk. We could have written data to a much larger number of files, but this would have forced disk writes to be smaller and therefore less efficient; 460-megabyte files are small enough for the next stage of the computation.

This initial sorting by 13 bits of h means that any hash collision (any pair $(s, h), (s', h')$ with $s \neq s'$ and $h = h'$) is within one of the 8192 files on the 8 servers. Each server periodically checks for collisions by reading each file into RAM and sorting the file within RAM. The server does *not* take the time to write the sorted file back to disk; sorting within RAM is not a bottleneck.

For each seed involved in a hash collision, the server recomputes the distinguished points for that seed, keeping track of the number of occurrences of $\sigma^3 + 1$, $\sigma^4 + 1$, etc. so that it can express the distinguished point as a linear combination of P and Q using the correspondence of σ and s. The server does this for all seeds in parallel and, finally, checks for colliding distinguished points.

As an end-to-end test of the client/server software we successfully re-broke the ECC2K-95 challenge in just 19 hours on 10 2.4GHz Core 2 Quad CPUs. We scaled the quarter-megabyte buffers down to quarter-kilobyte buffers and ended up collecting 134 megabytes of pairs (s, h).

Appendix D: Client-server communication

Each client sends a stream of reports to each server through the following lightweight data-transfer protocol. The client buffers 64 reports for the server; collects the 64 reports into a 1024-byte block; adds an 8-byte nonce identifying the client site, the stream (for sites sending many streams in parallel), and the position of the block within the stream (0, 1, 2, etc.); and sends the resulting packet to the server through UDP. The server sends back an 8-byte acknowledgment identifying the client site, the stream, and the next block position.

UDP packets can be lost, transmitted out of order, etc. The client sporadically retransmits unacknowledged packets. The client also sends subsequent packets within a 32768-byte window, although this feature is of interest only for sites that collect large volumes of reports into a small number of streams. The server keeps 4 bytes of state for each stream, namely the next

expected block position; if the server receives a valid packet with a different block position, it sends back an acknowledgment identifying the expected block position. The client keeps track of the largest position in the acknowledgments it receives, and skips retransmission of packets before that position.

Each packet is encrypted, authenticated, verified, and decrypted using the "Networking and Cryptography library" from `http://nacl.cace-project.eu`. Client-to-server packets are encrypted from the client's secret key to the server's public key under the specified nonce, and are decrypted from the client's public key to the server's secret key. This cryptographic protection adds 16 bytes to each packet and costs a negligible amount of CPU time. We do not think that there are attackers spying upon, or trying to interfere with, our computation, but this cryptographic protection also protects our data against corruption from bad network links (note that UDP has only a 16-bit checksum), bad firewall software, etc.

We checked with a network sniffer that each 1024-byte block of reports consumes 1090 bytes for an IP packet to the server and 66 bytes for an IP packet from the server, not counting physical-layer overhead. Receiving a terabyte of data through this protocol in six months will consume under a megabit per second of actual network bandwidth, a tolerably small fraction of the bandwidth available at the site hosting the servers.

For comparison, sending a 1024-byte packet through the standard SSH protocol consumes 1138 bytes for an IP packet to the server and 66 bytes for an (unauthenticated) IP packet from the server, after a setup phase with dozens of IP packets consuming nearly 10000 bytes. Each SSH connection also requires more than a megabyte of server RAM (with the standard OpenSSH implementation under Linux), so it is not possible for thousands of clients to maintain separate long-term SSH connections. Buffering much larger blocks would reduce the setup overhead to an acceptable level but would also drastically increase latency. It would have been possible to organize clients into a distributed tree of long-term SSH connections, but it was easier to build new communication software.