On the correct use of the negation map in the Pollard rho method

Daniel J. Bernstein[1], Tanja Lange[2], and Peter Schwabe[2]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org, peter@cryptojedi.org

Abstract. Bos, Kaihara, Kleinjung, Lenstra, and Montgomery recently showed that ECDLPs on the 112-bit secp112r1 curve can be solved in an expected time of 65 years on a PlayStation 3. This paper shows how to solve the same ECDLPs at almost twice the speed on the same hardware. The improvement comes primarily from a new variant of Pollard's rho method that fully exploits the negation map without branching, and secondarily from improved techniques for modular arithmetic.

Keywords: Elliptic curves, discrete-logarithm problem, negation map, branchless algorithms, SIMD.

1 Introduction

~~The purpose of this paper is to tell people something that is stated already in textbooks.~~ ← Suppressed introduction [18].

In July 2009, Bos, Kaihara, Kleinjung, Lenstra, and Montgomery announced breaking a discrete-logarithm problem on an elliptic curve over a 112-bit prime field using a cluster of 214 PlayStation 3 (PS3) game consoles. The initial announcement was [3]; more details on the same computation were published in [5], [4], and [6]. The overall attack costs were estimated to be about 60 PS3 years. This computation was, and still is, the largest ECDLP for which a successful solution has been publicly announced.

Our main result is that discrete-logarithm problems on the same curve (or any other curve of the form $y^2 = x^3 - 3x + b$ over the same field) can be solved at almost twice the speed on exactly the same hardware. We performed extensive computational experiments to verify our scalability and performance claims; details of these experiments appear in Section 6.

This work was supported by the National Science Foundation under grant ITR–0716498, by the European Commission under Contract ICT-2007-216499 CACE, and by the European Commission under Contract ICT-2007-216646 ECRYPT II. Computer time was provided by the Jülich and Barcelona supercomputer centers. Permanent ID of this document: dde51a91feeb8d746756566ac14323d1. Date: 2011.01.02.

This result combines several different optimizations. A large part of our work consists of faster algorithms for arithmetic modulo the prime $(2^{128}-3)/76439$ that defines this field; these algorithms use a different approach from the previous papers, and we describe this approach in detail. We also introduce several smaller refinements in rho computations, often co-designing our choice of iteration function with our lower-level optimizations. However, the largest part of our improvement, gaining a factor of almost $\sqrt{2}$, comes from careful use of the negation map.

The conventional wisdom is that the negation map has been known for ten years and trivially gains a factor of $\sqrt{2}$. However, [3] said "We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment." SIMD (single instruction, multiple data) is a critical feature of modern CPU designs, including the Cell processor used in the PS3.

Similarly, [5] said that the benefit of negation was outweighed by "the conditional branches required to check for fruitless cycles." The paper [6] observed that most of the negating rho algorithms stated in the literature were non-functional (i.e., had negligible chance of succeeding in the claimed amount of time); considered a huge array of 126 different combinations of negation options; and concluded that the best option did save time *for non-SIMD architectures*, but with a speedup far below $\sqrt{2}$. The paper continued to dispute the possibility of a negation speedup for SIMD architectures:

> If the Pollard rho method is parallelized in SIMD fashion, it is a challenge to achieve any speedup at all. ... Dealing with cycles entails administrative overhead and branching, which cause a non-negligible slowdown when running multiple walks in SIMD-parallel fashion. ... [This] is a major obstacle to the negation map in SIMD environments.

This paper resolves this dispute by explaining how to use the negation map without branching and without significant overhead handling cycles. We demonstrate a speedup very close to $\sqrt{2}$ on the PS3. We comment that the speedup on non-SIMD architectures would be even closer to $\sqrt{2}$.

2 Review of Pollard's rho method

This section gives an overview of Pollard's rho method. In this paper we will use this method to compute discrete logarithms on elliptic curves but the first subsections apply to any finite cyclic group $G = \langle P \rangle$ and $Q \in G$. Computing the discrete logarithm of Q to the base P means computing an integer k such that $Q = kP$. The integer k is unique modulo ℓ where ℓ is the order of P; we assume for simplicity that ℓ is an odd prime.

The generic rho method. Pollard's rho method [23] is a low-memory algorithm that finds a discrete logarithm by finding a collision in the map

$$(a, b) \mapsto aP + bQ \qquad \text{where} \qquad a, b \in \mathbb{Z}.$$

Finding a collision usually reveals the discrete logarithm k of Q to the base P: if $aP + bQ = a'P + b'Q$ and $b \not\equiv b' \pmod{\ell}$ then $k \equiv (a - a')/(b' - b) \pmod{\ell}$.

A generic way to find this collision is to iterate this function. Define maps a and b from $\langle P \rangle$ to $\mathbb{Z}$ and compute $W_{i+1} = f(W_i) = a(W_i)P + b(W_i)Q$, starting from some initial combination $W_0 = a_0P + b_0Q$. If any W_i and W_j collide then also $W_{i+1} = W_{j+1}$, $W_{i+2} = W_{j+2}$, etc. This means that the sequence enters a cycle. This can be detected efficiently using, e.g., Floyd's cycle-finding method.

If the functions $a(R)$ and $b(R)$ are random modulo ℓ then the iterations perform a random walk in $\langle P \rangle$. The random walk can be modeled as drawing objects with repetition from an urn containing ℓ elements. A collision corresponds to drawing the same element twice. A standard birthday-paradox calculation implies that a collision occurs after approximately $\sqrt{\pi\ell/2}$ iterations on average.

Use of group automorphisms. If the functions a and b are chosen such that $f(W_i) = f(-W_i)$ then the walk is actually defined on equivalence classes under $\pm$. There are only $\lceil \ell/2 \rceil$ different classes. This reduces the average number of iterations by a factor of almost exactly $\sqrt{2}$.

More generally, Pollard's rho method can be combined with any easily computed group automorphism σ of small order. One chooses a and b so that $f(W_i) = f(\sigma(W_i)) = f(\sigma^2(W_i)) = \cdots$. The walk is then defined on equivalence classes under the automorphisms, reducing the average number of iterations accordingly. However, for most elliptic curves the only easily computed group automorphism of small order is the negation map.

The parallel rho method. To spread the computations in Pollard's rho method across multiple computers one replaces the long walk by a collection of short walks, as proposed by van Oorschot and Wiener in [22]. Some fixed subset of $\langle P \rangle$ is declared to be the *set of distinguished points*. Whenever a walk reaches a distinguished point it reports this to a central server and the server stores the distinguished point along with the values for a and b. If two walks reach the same distinguished point the server notices a collision.

This parallelization requires the server to receive, store, and sort all distinguished points. Tradeoffs are possible. If distinguished points are chosen to be rare then a small number of very long walks will be performed, reducing the number of distinguished points sent to the server but increasing the delay before a collision is recognized. If distinguished points are frequent then many shorter walks will be performed.

Additive walks. The generic rho method described above requires two scalar multiplications for each iteration. One can merge the two scalar multiplications into a 2-scalar multiplication, and further merge the 2-scalar multiplications across several parallel iterations, to reduce the number of group additions required for each iteration; but it is much simpler to use an *additive walk*, requiring only one addition for each iteration.

An additive walk is defined as $W_{i+1} = f(W_i) = W_i + R_{h(W_i)}$. Here h maps from $\langle P \rangle$ to $\{0, 1, \ldots, r-1\}$, and $R_0, R_1, \ldots, R_{r-1}$ are precomputed as known combinations of P and Q: say $R_j = c_jP + d_jQ$ for each $j \in \{0, 1, \ldots, r-1\}$.

One then also knows that $W_i = a_iP + b_iQ$, where a and b are defined recursively as follows: $a_{i+1} = a_i + c_{h(W_i)}$ and $b_{i+1} = b_i + d_{h(W_i)}$.

Additive walks have disadvantages. The walks are noticeably nonrandom and need more iterations than the generic rho method to find a collision. This effect disappears as r grows, but if r is large then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. Additive walks also have trouble with automorphisms; see the discussion of fruitless cycles below.

Pollard's original proposal was to use $r = 3$. Instead of 3 different precomputed points, Pollard mixed 2 points with a doubling $W_{i+1} = 2W_i$; note that Pollard was computing discrete logarithms in multiplicative groups, where a doubling (i.e., a squaring) is faster than a general addition (i.e., a multiplication). Experiments by Teske in [27] showed that larger values of r, such as $r = 20$, are much closer to random walks. A general heuristic due to Brent and Pollard implies that nonrandomness slows down this type of walk by a factor $\sqrt{1 - 1/r}$; for further discussion of such heuristics see, e.g., [1].

Eliminating coefficients. In all of the discrete-logarithm computations described above, coefficients a_i, b_i are stored and used to compute the final discrete logarithm. In the parallel rho method these coefficients are communicated along with each distinguished point sent from a client to the server. Computing these coefficients in additive walks requires each client to implement arithmetic modulo ℓ or at least to allocate space for counters to keep track of how often each R_j is used.

An alternative approach, introduced recently as part of the ECC2K-130 attack [1], eliminates the coefficients a_i and b_i. Clients compute W_i without keeping track of a_i and b_i. When a client encounters a distinguished point W_i, it reports the point (or a hash of the point) along with a seed identifying the start of the walk, and then starts a new walk. This saves code in the clients, storage in the clients, communication between the clients and the server, and storage on the server. When the server encounters a collision, it recomputes the two walks involved in the collision, starting from the same seeds but now computing a_i and b_i. This is done only rarely, ideally just once.

Fruitless cycles. Fast negation on elliptic curves reduces the number of iterations in the generic rho method by a factor $\sqrt{2}$, as discussed above. However, a non-negating additive walk is faster than a negating generic walk: the extra speed of each iteration in the non-negating additive walk outweighs the smaller number of iterations in the negating generic walk.

One might think that a negating additive walk combines the advantages of a small number of iterations and a very fast iteration function. It is easy to make a canonical choice $|W_i|$ between W_i and $-W_i$; to define $h(W_i)$ as a function of $|W_i|$, so that $h(W_i) = h(-W_i)$; and to define $f(W_i) = |W_i| + R_{h(W_i)}$. Then f is a walk on equivalence classes under $\pm$, and computing f takes only one addition.

The problem is that a negating additive walk does not behave randomly: it quickly enters short fruitless cycles. For example, if $|W_{i+1}| = -W_{i+1}$ and $h(W_{i+1}) = h(W_i)$ then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + R_{h(W_i)} = -(|W_i| + R_{h(W_i)}) + R_{h(W_i)} = -|W_i|$ so $|W_{i+2}| = |W_i|$. One expects this to occur with probability $1/(2r)$ at each step, and if it does occur then the walk enters a 2-cycle. Subsequent iterations will not proceed to a distinguished point; subsequent computations will be wasted.

It is also possible to have fruitless cycles of larger lengths, although these are less frequent. Heuristics are given in [12]. Choosing a larger r reduces the chance of entering fruitless cycles, but for large-scale computations the cycles will occur and will occur frequently.

There are many different attempts in the literature to work around this problem. The first proposals were introduced independently by Harley and Singer in [16], Wiener and Zuccherato in [29], and Gallant, Lambert, and Vanstone in [14]; further analyses appeared in [12] and much more recently in [6]. See Appendix A for a detailed review of these proposals. Some of the proposals are successful in eliminating fruitless cycles, but all of these proposals involve frequent conditional operations and, as stated in [6], perform poorly in a SIMD environment.

3 How to use negation in Pollard's rho method

This section presents an efficient branchless negating rho algorithm to compute elliptic-curve discrete logarithms. For simplicity we restrict to curves of the form $y^2 = x^3 - 3x + b$ over large prime fields $\mathbb{F}_p$.

This algorithm uses, on average, $(1 + o(1))\sqrt{\pi\ell/4}$ iterations for a group of prime order ℓ, assuming standard heuristics; here $o(1)$ means something that converges to 0 as $\ell \to \infty$. Each iteration uses 5 multiplications mod p, 1 squaring mod p, and an asymptotically negligible amount of extra work.

We emphasize that we use a *branchless* sequence of iterations, always performing the same operations in the same order. This is of theoretical interest: any bounded-time algorithm can be made branchless by standard conversions, but these conversions usually lose efficiency. This is also of practical interest: the algorithm is well suited for modern SIMD CPUs such as the Cell CPU in the PS3, as discussed in subsequent sections of the paper.

We also emphasize that the number of iterations is $(1 + o(1))\sqrt{\pi\ell/4}$, not $(1 + o(1))\sqrt{\pi\ell/2}$. We use the fast elliptic-curve negation map to save a factor of $\sqrt{2}$; we do this without branching, and we do it with asymptotically zero compromise in iteration speed.

Eliminating fruitless cycles. We begin with the simplest type of negating additive walk stated in the literature. The walk starts at the point $W_0 = |b_0 Q|$ where b_0 is chosen randomly, and then computes $W_1, W_2, \ldots$ by the rule $W_{i+1} = |W_i + R_{h(W_i)}|$. Here $|(x, y)|$ means (x, y) for $y \in \{0, 2, 4, \ldots, p-1\}$ or $(x, -y)$ for $y \in \{1, 3, 5, \ldots, p-2\}$; the hash function h maps points to elements of $\{0, 1, 2, \ldots, r-1\}$; and the points $R_0, R_1, \ldots, R_{r-1}$ are known multiples of P.

We modify this walk by *occasionally* checking for fruitless cycles of length 2. Specifically, for a sparse pattern of indices i discussed below, we change the definition of W_i as follows. After computing W_{i-1}, we check whether $W_{i-1} =$

W_{i-3}. In the common case that $W_{i-1} \neq W_{i-3}$, we define $W_i = W_{i-1}$. In the unusual case that $W_{i-1} = W_{i-3}$, we define $W_i = |2 \min\{W_{i-1}, W_{i-2}\}|$, where min means lexicographic minimum and 2 means doubling.

We further modify this walk by occasionally, with even lower frequency, checking for fruitless cycles of length 4. Specifically, for an even more sparse pattern of indices i discussed below, we redefine W_i as W_{i-1} if $W_{i-1} \neq W_{i-5}$, and we redefine W_i as $|2 \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}\}|$ if $W_{i-1} = W_{i-5}$.

We continue with analogous modifications for fruitless cycles of lengths 6, 8, etc., up to the smallest even length that exceeds $(\log \ell)/(\log r)$.

Eliminating branches. The sequence of iterations described above might seem to include branches: a branch to replace y by $-y$ if y is odd, for example, and a branch to conditionally compute $W_i = |2 \min\{W_{i-1}, W_{i-2}\}|$. However, one can easily simulate all of these branches by a straight-line program with negligible loss of efficiency, as described in the following paragraphs.

First, $|(x, y)|$ is the same as $(x, (1 - 2\epsilon)y)$ where $\epsilon = y \bmod 2$. The implicit reduction modulo p here is not an asymptotic bottleneck: it takes linear time (even without branching), while all known multiplication algorithms take superlinear time. We prefer to make the implicit reduction explicit, computing $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$; the addition and subtraction take linear time.

Second, we amortize min computations such as $\min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}\}$ across all relevant iterations: after computing W_{i-3} we initialize a running minimum $W_{\min}$ as $\min\{W_{i-4}, W_{i-3}\}$, then replace it with $\min\{W_{\min}, W_{i-2}\}$ after computing W_{i-2}, then replace it with $\min\{W_{\min}, W_{i-1}\}$ after computing W_{i-1}. These computations are performed for only a small fraction of all indices i, so the loss of efficiency is negligible. See below for a more detailed cost analysis.

Third, we compute doublings such as $D_i = 2 \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}\} = 2W_{\min}$ for all of the selected indices, whether or not the doublings will actually be used. We then compute W_i without branches by selecting between W_{i-1} and $|D_i|$, the same way that $|(x, y)|$ selects between (x, y) and $(x, -y)$. The selection bit is the output of a branch-free comparison between W_{i-1} and W_{i-5}, or in general between W_{i-1} and W_{i-1-c} for detecting fruitless cycles of length c.

Note that each of these selections and comparisons takes linear time per iteration, and is therefore asymptotically negligible compared to a multiplication.

Eliminating inversions. The bottleneck in each iteration is now exactly one elliptic-curve operation: usually an elliptic-curve addition, but an elliptic-curve doubling for occasional iterations.

The standard formulas for elliptic-curve addition in affine coordinates are as follows. Let $P = (x_1, y_1), R = (x_2, y_2)$ with $x_1 \neq x_2$. Then $P + R = (x_3, y_3)$ where $\lambda = (y_1 - y_2)/(x_1 - x_2)$, $x_3 = \lambda^2 - x_1 - x_2$, and $y_3 = \lambda(x_1 - x_3) - y_1$. These formulas use 1 inversion, 2 multiplications, 1 squaring, and 6 subtractions. The formulas for a doubling, where $R = P$, are very similar; only the computation of $\lambda = 3(x_1^2 - 1)/(2y_1)$ is different. We ignore, without further comment, the extraordinarily unlikely event that $R = -P$.

Inversions are the most expensive operations in finite fields. Standard practice in rho computations is to perform m independent walks in parallel, and to use

Montgomery's trick [21], which computes a batch of m inversions using just 1 inversion and $3(m-1)$ multiplications.

We choose a branchless inversion method, specifically computing the $(p-2)$nd power using $O(\lg p)$ multiplications. We then choose m to grow asymptotically more quickly than $\lg p$: in other words, $\lg p \in o(m)$. A batch of m elliptic-curve additions then costs $5m-3$ multiplications, m squarings, $6m$ subtractions, and 1 inversion. The subtractions and inversion are negligible, so each elliptic-curve addition costs 5 multiplications and 1 squaring.

A batch of m elliptic-curve doublings is slightly more expensive (costing m extra squarings), but occurs for only a small fraction of iterations, as discussed below. Each iteration therefore uses 5 multiplications and 1 squaring.

Analysis and optimization. Fruitless cycles of length 2 appear with probability approximately $1/(2r)$. These cycles persist after they appear, wasting subsequent iterations (in the sense that new points and new collision opportunities do not occur), until we check for them. If we check every w iterations then we expect a cycle to appear with probability approximately $w/(2r)$, and for it to waste approximately $w/2$ iterations on average if it does appear.

This does not mean that w should be chosen as small as possible. If a cycle has *not* appeared then checking for it wastes an iteration. The overall loss is approximately $1 + w^2/(4r)$ iterations out of w. To minimize the quotient $1/w + w/(4r)$ we take $w \approx 2\sqrt{r}$.

More generally, fruitless cycles of small length $2c$ appear with probability approximately proportional to $1/r^c$, so the optimal checking frequency is approximately proportional to $1/r^{c/2}$. The loss here rapidly disappears as c increases.

To summarize, fruitless cycles slow down this algorithm by a factor $1 + \Theta(1/\sqrt{r})$. This negation overhead $\Theta(1/\sqrt{r})$ is on a larger scale than the overhead $\Theta(1/r)$ from the nonrandomness of r-adding walks, but both overheads become asymptotically negligible if r is chosen so that $r \to \infty$ as $p \to \infty$.

As an illustration of these optimizations, our PS3 software takes $r = 2048$, checks for 2-cycles every 48 iterations, and checks for larger cycles much less frequently. To simplify the software we unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations. If we had instead taken $r = 512$ then we would have checked for 2-cycles every 24 iterations. In general, the $\Theta(1/\sqrt{r})$ asymptotic means that the negation overhead approximately doubles when the table size is reduced by a factor of 4.

Storage reduction. The storage overhead for detecting and escaping a fruitless cycle consists of storing $W_{\min}$ and W_{i-1-c}. For the latter it is enough to store one of the coordinates.

We further reduce storage by avoiding having all iterations check for cycles at the same time. For example, with a batch of 224 iterations running in parallel, we have just 14 iterations checking for 2-cycles and consuming extra space. All iterations perform 2 addition steps, and then these 14 iterations perform a masked doubling while the remaining 210 iterations perform another addition. We then rotate the batch so that the next 14 iterations check for 2-cycles.

4 Low-cost arithmetic in $\mathbb{Z}/(2^{128}-3)$

Elements of the prime field $\mathbb{F}_p$, where $p = (2^{128}-3)/76439$, can be represented redundantly as elements of the ring $\mathbb{Z}/(2^{128}-3)$. Instead of reducing sums and products modulo p one reduces them modulo $2^{128}-3$. This representation requires about 15% more space per field element, but the sparsity of $2^{128}-3$ makes reductions much faster.

The prime p was chosen with this small sparse multiple precisely to allow this speedup for cryptographic operations on the secp112r1 curve; see [10, page 3, bottom, and page 6, bottom]. The same type of redundant representation can of course also be used by the cryptanalyst attacking the ECDLP on secp112r1, as in [3], [5], [4], and this paper. The critical problem is then to perform fast arithmetic modulo $2^{128}-3$.

This section explains how to efficiently decompose multiplications and squarings modulo $2^{128}-3$ into operations on 16-bit integers and 32-bit integers. Here a *b-bit integer* is an integer in the interval $[-2^{b-1}, 2^{b-1}-1]$. The next section applies this decomposition to the Cell, obtaining faster arithmetic than [3] et al.

Model of computation. This section uses a simplified model of computation that counts $16 \times 16 \to 32$ multiplications and certain other operations. Specifically, algorithms in this section are branchless sequences of the following operations:

- multiplication: $a, b \mapsto ab$ where a, b are 16-bit integers and ab is a 32-bit integer;
- multiply-add: $a, b, c \mapsto ab + c$ where a, b are 16-bit integers and $c, ab + c$ are 32-bit integers;
- 16-bit addition: $a, b \mapsto a + b$ where $a, b, a + b$ are 16-bit integers;
- 32-bit addition: $a, b \mapsto a + b$ where $a, b, a + b$ are 32-bit integers;
- 32-bit subtraction: $a, b \mapsto a - b$ where $a, b, a - b$ are 32-bit integers;
- 32-bit right shift by 12 bits: $a \mapsto \lfloor a/2^{12} \rfloor$ where a is a 32-bit integer;
- 32-bit right shift by 13 bits: $a \mapsto \lfloor a/2^{13} \rfloor$ where a is a 32-bit integer;
- 32-bit mask clearing 12 bits: $a \mapsto 2^{12}\lfloor a/2^{12} \rfloor$ where a is a 32-bit integer; and
- 32-bit mask clearing 13 bits: $a \mapsto 2^{13}\lfloor a/2^{13} \rfloor$ where a is a 32-bit integer.

We assign cost 1 to each of these operations, except that we assign cost 0.5 to the 16-bit addition operation. The next section explains how this cost model is related to PS3 speed.

Note that these operations are not defined for all pairs of inputs. For example, 32-bit addition is not permitted to add 2^{30} to 2^{30}, because 2^{31} is too large to be a 32-bit integer. One could of course define an extended 32-bit addition operation that handles all cases, working modulo 2^{32} to handle overflows, but there are no overflows in the algorithms in this section. One could also define shifts (and masks) for distances other than 12 and 13, but the only distances used in this section are 12 and 13.

Representing integers modulo $2^{128}-3$. We represent an element f of the ring $\mathbb{Z}/(2^{128}-3)$ as a sequence of 10 coefficients $(f_0, \ldots, f_9)$ such that $f =$

$\sum_{0\le i\le 9} f_i 2^{\lceil i\cdot 12.8\rceil}$; i.e.,

$$f = f_0 + f_1 2^{13} + f_2 2^{26} + f_3 2^{39} + f_4 2^{52} + f_5 2^{64} + f_6 2^{77} + f_7 2^{90} + f_8 2^{103} + f_9 2^{116}.$$

Note that the exponents $0, 13, 26, 39, 52, 64, 77, 90, 103, 116$ are not exactly evenly spaced. Our non-integer radix $2^{12.8}$ follows the use of radix $2^{25.5}$ by Bernstein in [2], and follows ideas from Costigan and Schwabe in [11] on making best use of SIMD instructions.

We call a coefficient f_i *reduced* if $|f_i| \le 1.01 \cdot 2^{12}$. We call $(f_0, \ldots, f_9)$ *reduced* if all coefficients are reduced.

Polynomial multiplication and polynomial reduction. It is easy to check that if $f = \sum_{0\le i\le 9} f_i 2^{\lceil i\cdot 12.8\rceil}$ and $g = \sum_{0\le i\le 9} g_i 2^{\lceil i\cdot 12.8\rceil}$ then the product fg equals $\sum_{0\le i\le 9} r_i 2^{\lceil i\cdot 12.8\rceil}$ in $\mathbb{Z}/(2^{128}-3)$ where

$$\begin{aligned}
r_0 &= f_0g_0+3(2f_9g_1+2f_8g_2+2f_7g_3+2f_6g_4+f_5g_5+2f_4g_6+2f_3g_7+2f_2g_8+2f_1g_9),\\
r_1 &= f_1g_0+f_0g_1+3(2f_9g_2+2f_8g_3+2f_7g_4+f_6g_5+f_5g_6+2f_4g_7+2f_3g_8+2f_2g_9),\\
r_2 &= f_2g_0+f_1g_1+f_0g_2+3(2f_9g_3+2f_8g_4+f_7g_5+f_6g_6+f_5g_7+2f_4g_8+2f_3g_9),\\
r_3 &= f_3g_0+f_2g_1+f_1g_2+f_0g_3+3(2f_9g_4+f_8g_5+f_7g_6+f_6g_7+f_5g_8+2f_4g_9),\\
r_4 &= f_4g_0+f_3g_1+f_2g_2+f_1g_3+f_0g_4+3(f_9g_5+f_8g_6+f_7g_7+f_6g_8+f_5g_9),\\
r_5 &= f_5g_0+2f_4g_1+2f_3g_2+2f_2g_3+2f_1g_4+f_0g_5+3(2f_9g_6+2f_8g_7+2f_7g_8+2f_6g_9),\\
r_6 &= f_6g_0+f_5g_1+2f_4g_2+2f_3g_3+2f_2g_4+f_1g_5+f_0g_6+3(2f_9g_7+2f_8g_8+2f_7g_9),\\
r_7 &= f_7g_0+f_6g_1+f_5g_2+2f_4g_3+2f_3g_4+f_2g_5+f_1g_6+f_0g_7+3(2f_9g_8+2f_8g_9),\\
r_8 &= f_8g_0+f_7g_1+f_6g_2+f_5g_3+2f_4g_4+f_3g_5+f_2g_6+f_1g_7+f_0g_8+3(2f_9g_9),\\
r_9 &= f_9g_0+f_8g_1+f_7g_2+f_6g_3+f_5g_4+f_4g_5+f_3g_6+f_2g_7+f_1g_8+f_0g_9.
\end{aligned}$$

The factors of 2 arise from the uneven exponent spacing mentioned above: for example, the product of $f_1 2^{13}$ and $g_4 2^{52}$ is $2f_1g_4 2^{64}$, contributing $2f_1g_4$ to $r_5 2^{64}$. The factors of 3 arise from reducing 2^{128} in $\mathbb{Z}/(2^{128}-3)$.

If $(f_0, \ldots, f_9)$ and $(g_0, \ldots, g_9)$ are reduced then a sum of any 10 products of the form f_ig_j can be computed with cost 10: specifically, 1 multiplication followed by 9 multiply-add operations in any convenient order. The sums $r_0, r_1, \ldots, r_9$ are slightly more expensive because of the extra factors of 2 and 3. We precompute $3g_1, 3g_2, \ldots, 3g_9$ and $2f_1, 2f_2, 2f_3, 2f_4, 2f_6, 2f_7, 2f_8, 2f_9$ (skipping $2f_5$); recall that a 16-bit addition costs only 0.5, so these 26 additions cost only 13. Each r_i is then a sum of 10 products of known 16-bit quantities, costing 100, for a total cost of 113.

Each r_i, and each intermediate result in this multiplication algorithm, is bounded in absolute value by $10 \cdot 3 \cdot 2 \cdot (1.01 \cdot 2^{12})^2 < 0.96 \cdot 2^{30}$. The same algorithm also works if $(f_0, \ldots, f_9)$ is a sum or difference of two reduced vectors while $(g_0, \ldots, g_9)$ is reduced: then each result is bounded in absolute value by $0.96 \cdot 2^{31}$, still safely below 2^{31}.

Coefficient reduction. The product coefficients $r_0, r_1, \ldots, r_9$ constructed above are usually not reduced. Some extra work is required to compute a reduced product suitable for use as input to subsequent multiplications.

For $i \in \{0, 1, 2, 3, 5, 6, 7, 8\}$ we reduce r_i by *carrying* from r_i to r_{i+1}. This means changing (r_i, r_{i+1}) into $(r_i - 2^{13}c, r_{i+1} + c)$ where $c = \lfloor (r_i + 2^{12})/2^{13} \rfloor$. This leaves the sum $r_i 2^{\lceil i \cdot 12.8 \rceil} + r_{i+1} 2^{\lceil (i+1) \cdot 12.8 \rceil}$ unaffected, and increases the maximum possible r_{i+1} only slightly, while guaranteeing that the new r_i is between -2^{12} and 2^{12}. This costs 5: an addition, a right shift, a mask, another addition, and a subtraction.

We similarly reduce r_4 by carrying from r_4 to r_5. This has a slightly different definition, to accommodate the uneven spacing of exponents: it means changing (r_4, r_5) into $(r_4 - 2^{12}c, r_5 + c)$ where $c = \lfloor (r_4 + 2^{11})/2^{12} \rfloor$. This guarantees that the new r_4 is between -2^{11} and 2^{11}.

The most expensive step is to reduce r_9 by carrying from r_9 to r_0. This means changing (r_9, r_0) into $(r_9 - 2^{12}c, r_0 + 3c)$ where $c = \lfloor (r_9 + 2^{11})/2^{12} \rfloor$. This leaves $r_0 + r_9 2^{116}$ unaffected modulo $2^{128} - 3$, while guaranteeing that the new r_9 is between -2^{11} and 2^{11}. This costs 7 rather than 5: the computation of $3c$ uses two extra additions.

We use the carry chain $r_0 \to r_1 \to r_2 \to r_3 \to r_4 \to r_5 \to r_6 \to r_7 \to r_8 \to r_9 \to r_0 \to r_1 \to r_2$: we first carry from r_0 to r_1, then from r_1 to r_2, etc., then from r_9 to r_0, then again from r_0 to r_1, then again from r_1 to r_2. Tracing the bounds on each r_i shows that the final r_2 is reduced, and therefore that $(r_0, r_1, \ldots, r_9)$ is reduced.

This carry chain costs 62. The complete multiplication algorithm, taking reduced representations of f and g as input and producing a reduced representation of fg as output, costs 175.

Squaring. Squaring in $\mathbb{Z}/(2^{128} - 3)$ is very similar to multiplication except that several intermediate results are reused: for example, $f_1 g_0 + f_0 g_1$ becomes just $2 f_0 f_1$. We begin with a cost-10 computation of $2f_0, \ldots, 2f_9$; $3f_5, \ldots, 3f_9$; $6f_5, \ldots, 6f_9$. We then obtain $r_0, \ldots, r_9$ straightforwardly at cost 55, and apply the same carry chain as for multiplication. The complete squaring algorithm costs 127.

5 Fast iterations on the PlayStation 3

This section analyzes and optimizes the performance of rho iterations modulo $p = (2^{128} - 3)/76439$ on the PS3. This optimization makes some changes to the iteration function; we take advantage of the flexibility of co-designing our iteration function with our arithmetic algorithms.

The Cell SPEs. The CPU in the PS3 is the Cell Broadband Engine. The main computational power of the Cell is in 8 Synergistic Processor Elements (SPEs). These SPEs are arranged around a central 64-bit PowerPC core. The PS3 makes only 6 of these SPEs available for computations.

We report the performance that we achieve with 6 SPEs, leaving the central PowerPC core mostly idle. This does not make full use of the Cell—performing some iterations on the PowerPC core would noticeably reduce the overall computation time—but it simplifies comparisons: the speeds reported for the same ECDLP in [3], [5], and [4] also left the central PowerPC core mostly idle.

Our implementation runs independently on each SPE. Each SPE has 256 KB of fast local storage; this storage holds all code and data, including a batch of parallel walks and a table of precomputed points. Communication between the SPE and main memory is very small in this algorithm: a few words of data for every million iterations. Performance is therefore determined almost completely by how well we make use of the computational power of the SPE.

Arithmetic on the SPE. The following description summarizes only the SPE features that are relevant to our implementation. See [19] and [25] for more information about the SPE.

The SPE has a register file consisting of 128 128-bit vector registers. Typical arithmetic instructions are SIMD instructions operating on these registers as vectors of 4 independent 32-bit integers or vectors of 8 independent 16-bit integers. There is a multiplication instruction that multiplies 4 pairs of 16-bit integers in parallel, producing 4 32-bit integers. There is also a fused multiply-add instruction that adds the results into another vector of 4 32-bit integers.

Each SPE cycle carries out at most one of these arithmetic instructions: i.e., 4 32-bit operations or 8 16-bit operations. An algorithm that costs n in the model of Section 4 therefore uses at least $n/4$ cycles on the SPE. There are, however, several reasons that the SPE can take many more cycles than this:

- In-order execution. An arithmetic instruction must wait until 1 cycle after the previous arithmetic instruction in the program.
- Arithmetic latency. An instruction cannot begin until its results are ready. The result of an arithmetic instruction is not ready until several cycles later.
- Load latency. Loads are handled by a separate instruction pipeline but can still delay arithmetic instructions that use the load results.

There are also various function-call overheads, typically consuming 70 cycles per function call. One can eliminate these overheads by inlining and merging functions, but this also increases code size, putting pressure on the SPE's local storage.

Digitsliced multiplication on the SPE. We use 8-way vectorization of our iterations: we repeat our inputs, computations, and outputs 8 independent times. We store 8 independent elements of the ring $\mathbb{Z}/(2^{128}-3)$ in 10 128-bit vector registers $r_0, \ldots, r_9$, where coefficient i of ring element j is in 16-bit component j of register r_i. We convert each 16-bit operation into a 128-bit vector operation, and we convert each 32-bit operation into two 128-bit vector operations.

Scheduling instructions carefully then works around all arithmetic latency. The multiplication algorithm fits comfortably into the SPE registers: loads and stores are not a bottleneck. (Replacing 8-way vectorization with 16-way vectorization would remove the need for careful instruction scheduling but would put more pressure on registers and, more importantly, would cut in half the number of walks that fit easily into local storage without reshuffling.) The 32-bit results at the end of the algorithm are known to be reduced, so they fit into 16 bits; 10 extra instructions are required to shuffle the results into 10 vectors of 8 16-bit integers, but these extra instructions are handled by the SPE's load/store pipeline and are effectively free when they are interleaved with arithmetic instructions.

Overall our software for vectorized multiplication of 8 pairs of ring elements takes exactly 350 SPE arithmetic instructions, i.e., 43.75 arithmetic instructions per multiplication (1/4 of the cost 175 in the previous section). Our software for vectorized squaring of 8 ring elements uses exactly 254 SPE arithmetic instructions, i.e., 31.75 arithmetic instructions per squaring. Each of our iterations performs 5 multiplications and 1 squaring, in total consuming 250.5 arithmetic instructions, at least 250.5 cycles.

Inversion. To invert modulo $p = (2^{128}-3)/76439$ we simply compute a $(p-2)$nd power. Our addition chain for $p-2$ uses 107 squarings and 32 multiplications. Essentially the same speed would also be achieved by an addition chain for the larger but sparser exponent $2^{128} - 76443 = p - 2 + 76439(p-1)$, using 126 squarings and just 18 multiplications.

Of course, we actually perform 8 independent inversions in parallel, using $8 \cdot 107$ squarings and $8 \cdot 32$ multiplications modulo $2^{128}-3$. Each inversion uses $107 \cdot 31.75 + 32 \cdot 43.75 = 4797.25$ arithmetic instructions, consuming at least 4797.25 cycles.

Our 8-inversion function actually uses 43293 cycles, i.e., 5411.625 cycles per inversion. The gap is almost entirely explained by the overhead of 64 function calls, half to multiplication and half to an n-squaring function that computes $r \leftarrow r^{2^n}$ for variable n.

To reduce code size we rejected the possibility of more complicated Euclid-type inversion as used in [5]. In context (see below) inversion is already quite fast, only 6.6% of our final iteration cost.

Canonicalizing the y-coordinate. Redundant representations cause trouble for two parts of the algorithm stated in Section 3. First, because $y \in \mathbb{F}_p$ has multiple representations, checking whether $y \in \{0, 2, 4, \ldots, p-1\}$ is not a simple matter of inspecting the bottom bit of the representation of y. Second, because $x \in \mathbb{F}_p$ has multiple representations, finding the hash of x is not a simple matter of extracting bits from the representation of x.

We address both problems by canonicalizing y. We use the canonicalized version of y to decide whether to negate y. Rather than canonicalizing and hashing x, we extract some bits from the canonicalized version of y as a table index. Note that there can be as many as 3 points having the same y-coordinate, but hashing all of those points to the same table index does not merge walks. We also use the canonicalized version of y to define distinguished points.

The most obvious way to canonicalize y is to replace it with $y \bmod p$; but reductions modulo p are expensive. We instead compute $s = 76439y \bmod (2^{128} - 3)$. One can think of this s as a unique representative $y \bmod p$, but represented as $76439(y \bmod p)$. An alternative is to use Montgomery reduction to compute $y \cdot 2^{-16} \pmod p$ with precomputed $2^{-16} \pmod p$, as in [4].

To compute s we multiply y by the cofactor 76439 and then perform a slightly longer reduction chain than the one we use after multiplication. The polynomial-multiplication step here, producing unreduced coefficients $s_0, s_1, \ldots, s_9$, uses only

5 arithmetic instructions (cost 20) instead of 25 (cost 100), because 76439 is represented in just two reduced coefficients: $76439 = 2711 + 9 \cdot 2^{13}$. The usual precomputations of $3g_1$, $2f_1$, etc. also disappear: all we need are the constants $5422 = 2 \cdot 2711$ and $16266 = 6 \cdot 2711$.

To reduce the result we carry $s_6 \to s_7 \to s_8 \to s_9 \to s_0 \to s_1 \to s_2 \to s_3 \to s_4 \to s_5 \to s_6 \to s_7 \to s_8 \to s_9 \to s_0 \to s_1$. This uses 19.75 arithmetic instructions. The overall cost of canonicalization is 24.75 arithmetic instructions per iteration.

We do not claim that the resulting $(s_0, s_1, \ldots, s_9)$ is *always* completely determined by the image of y in $\mathbb{F}_p$. What matters is that the probability of non-uniqueness for random y's is so small that two colliding walks have negligible probability of diverging before they hit a distinguished point. We computer-verified this as explained in Section 6.

Subtraction. The easy part of subtracting two ring elements is performing 10 subtractions of 16-bit integers. The problem is that the output is usually not reduced. Carries would reduce the output but would make subtraction much more expensive.

We use two standard techniques to avoid carries after subtractions. First, we skip unnecessary reductions before multiplications; recall from Section 4 that the multiplication algorithm can safely multiply f by $g - g_2$, where f, g, g_2 are reduced. The elliptic-curve addition formulas involve three subtractions of this type: the denominator $x - x_2$, which is multiplied by a reduced product inside Montgomery's batch-inversion method (except for one product at the beginning of a batch); the numerator $y - y_2$, which is multiplied by the reduced reciprocal of $x - x_2$; and $x_2 - x_3$, which is multiplied by the reduced quotient λ.

Second, we combine multiply-reduce-subtract-reduce into multiply-subtract-reduce. In particular, to compute $\lambda^2 - x - x_2$, we first add x to x_2, and then subtract the sum from λ^2 *before* reducing the coefficients of λ^2. Similarly, we subtract y_2 from $\lambda(x_2 - x_3)$ before reducing the coefficients of $\lambda(x_2 - x_3)$. This makes the subtractions more expensive (32-bit instead of 16-bit), but still much less expensive than extra carries.

Overall these 6 subtractions use 10 arithmetic instructions: 5 instructions for 40 subtractions of 16-bit integers, and another 5 instructions for 20 subtractions of 32-bit integers.

Table lookups. Our iteration function uses a precomputed table of 2048 multiples of P. Each multiple uses 16 bytes for an x-coordinate and 16 bytes for a y-coordinate, for a total of 64 KB of local storage. If we were concerned with defining an iteration function to perform well on many platforms simultaneously then we would use a smaller table, say 16 KB, to avoid L1 cache misses on typical CPUs. However, the analysis in Section 3 shows that this would slightly increase the number of iterations. In this paper our goal is purely to maximize Cell performance, so we keep $r = 2048$.

Normally a precomputed x-coordinate in reduced form $(x_0, x_1, \ldots, x_9)$ would occupy 20 bytes. We instead represent each precomputed x-coordinate in reduced form $(x_0, x_1, \ldots, x_7, 0, 0)$, stored as a contiguous 16-byte vector $(x_0, x_1, \ldots, x_7)$.

Each y-coordinate is stored similarly. The final zeros mean that this representation is limited to approximately 2^{103} different integers; it cannot handle all elements of $\mathbb{Z}/(2^{128}-3)$, or even all elements of $\mathbb{F}_p$. We find 2048 representable multiples of P by generating approximately 400 million random multiples of P; this precomputation has negligible cost.

This table representation also allows lookups with very little arithmetic (but with many load/store instructions, which we carefully interleave with other computations), as explained in the following paragraph. We could further compress the table entries in several ways, but we are not aware of further compression techniques that avoid extra arithmetic. We comment that scaling the same type of precomputation to larger ECDLPs, squeezing the precomputed points as far as we can with a negligible precomputation, reduces the space for rho tables by asymptotically 25%.

The table entry for a point (x, y) has index $s_0 \bmod 2048$ where $(s_0, s_1, \ldots, s_9)$ is the canonicalization of y. We perform 8 table lookups in parallel as follows. We perform one arithmetic instruction to obtain 8 parallel table indices; this instruction is a mask of the vector $(2047, 2047, 2047, 2047, 2047, 2047, 2047, 2047)$ with a vector of canonicalizations. We shift the result left by 5 bits (since table entries have 32 bytes) to obtain 8 addresses in memory; this 128-bit shift is handled by the SPE's load/store pipeline. We then shuffle the result into 8 separate registers, perform 8 x-coordinate loads, and perform 8 y-coordinate loads. Finally we use 24 shuffle instructions to convert to digitsliced format.

Batching inversions. We use Montgomery's trick to batch the inversions in 224 independent iterations, replacing them by 669 multiplications and 1 inversion. This batching is on top of the 8-way parallelism of all of our arithmetic operations. Overall the SPE handles $1792 = 224 \cdot 8$ walks at once. At each moment we watch 1/16th of the walks for fruitless cycles.

Each walk uses 70 bytes of storage: 20 for x, 20 for y, 20 for s, 8 for the seed used to start the walk, and 2 for a table index. Each watched walk uses an extra 102 bytes of storage: 20 for the first s to detect a cycle, 60 for the smallest (s, x, y) to escape a cycle, 20 as extra storage needed for conditional doubling, and 2 for a flag indicating whether the walk needs a doubling to escape a cycle. Overall the walks use $1792(70 + 102/16) = 136864$ bytes of local storage.

Overall performance. The most important arithmetic instructions in each iteration are as follows:

- 5 multiplications: 218.75 arithmetic instructions (43.75 each);
- 1 squaring: 31.75 arithmetic instructions;
- 1 canonicalization: 24.75 arithmetic instructions;
- 6 subtractions: 10.00 arithmetic instructions;
- 1/224 inversion minus 3/224 multiplications: ≈ 20.83 arithmetic instructions.

These add up to 306.08 arithmetic instructions per iteration, implying a lower bound of 306.08 cycles per iteration for our software. Our software actually takes 362 cycles per iteration, about 18% more than this lower bound.

Under 4% of the cycles per iteration are spent on operations that can be blamed on negation: specifically, negating s and y, detecting fruitless cycles, and resolving fruitless cycles by doubling. The rest of the gap between 306 and 362 is explained by detection of distinguished points, loop control, and function-call overhead.

There is, outside the iterations, an extra cost for communicating the occasional distinguished points that do appear (and in setting up replacement points). We made no effort to optimize this cost, since it is multiplied by the distinguished-point probability. With the rather large distinguished-point probability used in our experiments, namely 2^{-20}, and with all 6 SPEs running in parallel and competing for communication resources, this cost effectively added 15 cycles to each iteration, slowing the computation down by about 4%. A smaller distinguished-point probability would reduce this penalty below 1%.

Comparison to previous work. Bos, Kaihara, Kleinjung, Lenstra, and Montgomery state in [4, Appendix A] that they use 456 SPE cycles per iteration for the same ECDLP, including 322 cycles for 6 multiplications, 30 cycles for 6 subtractions, 12 cycles for 1/400 inversions, 24 cycles for canonicalization (with Montgomery reduction), and 68 cycles for miscellaneous overhead. Bos, Kaihara, and Montgomery report 453 cycles per iteration in [5, Section 5], with 318 cycles instead of 322 for the multiplications, and 69 cycles instead of 68 for the miscellaneous overhead.

Each of our multiplications is faster than the multiplications in [4] and [5], by a factor of approximately 1.23. This speedup can be traced directly to our use of the non-integer radix $2^{12.8}$, while [4] et al. used the conventional radix 2^{16}. Most of our other operations are also faster than the operations in [4]. We pay a slight penalty for negation but overall gain the same factor of approximately 1.23 in the number of cycles per iteration. Our overall speedup in solving the ECDLP is much larger, because we use far fewer iterations, as discussed in Section 6.

6 Experimental results and evaluation

We do not have access to the cluster of 1284 SPEs used for many months by the authors of [3], [4], [5], and [6]. However, a few SPEs at the Jülich and Barcelona supercomputer centers were enough for us to perform some reasonably large discrete-logarithm experiments, demonstrating clearly that our code works and runs at the expected speed. This section presents the details of our experiments.

Scaling elliptic-curve challenges without changing the prime. Our software is dedicated to the prime $p = (2^{128} - 3)/76439$, and is designed to break the ECDLP on the curve secp112r1 over $\mathbb{F}_p$. However, the same software works without modification for points P, Q on any curve of the form $y^2 = x^3 - 3x + b$ over this $\mathbb{F}_p$.

By counting points on $y^2 = x^3 - 3x + b$ for various b we found group orders having many different prime divisors. For example, there are points

– of prime order $1195174242772417 \approx 1.0615 \cdot 2^{50}$ on $y^2 = x^3 - 3x + 238^2$;

- of prime order $36817627222637377 \approx 1.0219 \cdot 2^{55}$ on $y^2 = x^3 - 3x + 372^2$; and
- of prime order $1186848158152955759 \approx 1.0294 \cdot 2^{60}$ on $y^2 = x^3 - 3x + 240^2$.

For each of these prime-order groups we generated a challenge P, Q and then repeatedly solved the challenge, collecting statistics on the distribution of the time needed to solve the challenge.

Distinguished points per second. We used the same distinguished-point property for all of the challenges, inspecting 20 bits of s_4 and s_9. The probability of a point being distinguished is almost exactly 2^{-20}.

We predicted that we would need slightly more than 2^{20} iterations on average to find a distinguished point, for two reasons. The first reason is that a small percentage of the iterations are wasted by fruitless cycles, as discussed in Section 3. This percentage is under 1% and is independent of the group order ℓ.

The second reason is that a walk can enter a long cycle that does not contain a distinguished point. We predicted that this would occur with probability roughly $2^{40}/\ell$ for each seed, i.e., roughly once for every $\ell/2^{40}$ seeds: certainly not an issue for a single-shot experiment with $\ell \approx 2^{112}$, but a serious concern for a careful statistical analysis studying many seeds with $\ell \approx 2^{50}$.

To prevent our software from running forever in case of long cycles, we added a few lines of code to abort each walk after approximately $47 \cdot 2^{18}$ iterations. A walk that is not in a long cycle has probability only about 2^{-17} of surviving for so many iterations and of therefore being aborted. We could also have modified our software to extract discrete logarithms from the long cycles, but there would have been no cryptanalytic benefit from doing so, since long cycles disappear as ℓ grows.

We also reduced our batch size from 224 to 192 (watching 12 instead of 14) to make room in local storage for keeping track of various statistics. This made each iteration slightly slower, 366 cycles instead of 362 cycles. The extra cost of communicating distinguished points adds 15 cycles per iteration as discussed in the previous section, so we predicted that 6 SPEs running in parallel would produce $6 \cdot 3.192 \cdot 10^9/(381 \cdot 2^{20}) \approx 47.94$ distinguished points per second.

We ran 6 SPEs on the original curve secp112r1 and found 48134 distinguished points in 1000 seconds, with no aborted walks. We also ran various experiments on our 50-bit, 55-bit, and 60-bit challenge curves, and in each case found distinguished points at the expected rate. We found 1 aborted walk for every $2^{12.9}$ distinguished points for the 55-bit challenge curve $y^2 = x^3 - 3x + 372^2$ with $\ell \approx 1.0219 \cdot 2^{55}$.

The number of distinguished points needed for a discrete logarithm. We performed the following experiment for our 50-bit challenge. Take a seed, and find the corresponding distinguished point. Take the next seed, and find the corresponding distinguished point. Continue this process until finding two colliding distinguished points. Compute a discrete logarithm from this collision, and verify that it matches the secret scalar used to generate the challenge in the first place.

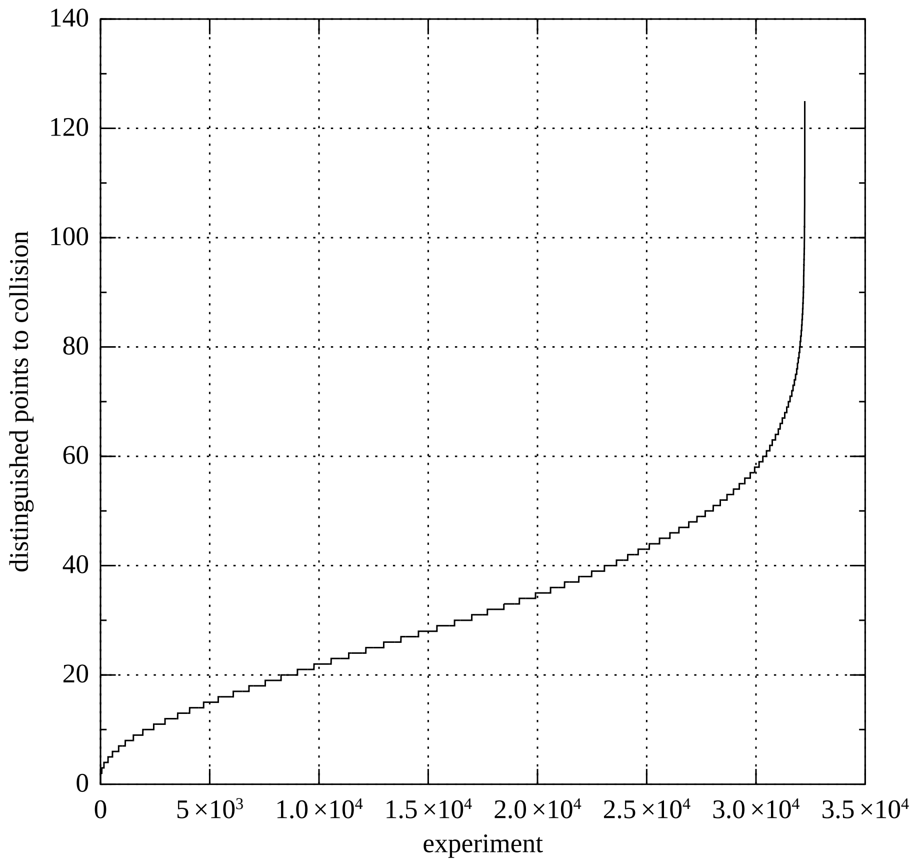

Fig. 6.1. Distribution of the number of distinguished points required to find a collision in 32237 independent experiments in a subgroup of size $\approx 1.0615 \cdot 2^{50}$ of the curve $y^2 = x^3 - 3x + 238^2$. The experiments were retroactively reordered so that experiment 1 used the smallest number d_1 of distinguished points, experiment 2 used d_2 distinguished points with $d_2 \geq d_1$, etc. A point (i, d_i) on this graph means that experiment i, in this retroactive ordering, used d_i distinguished points.

Our software handles many seeds at once and produces distinguished points out of order, so we sorted the outputs back into the original order of seeds. Skipping this step would have introduced a bias into our experiment, favoring distinguished points that use relatively few iterations.

We then performed this experiment again, starting from the first seed that was not used in the first experiment. We continued in the same way through 32237 experiments, using disjoint seeds for each experiment. We did not verify the discrete logarithms for every experiment, but we verified it for a large random sample of experiments, and encountered no failures. The number of distinguished points used in the experiments was on average $31.526 \approx 1.0789\sqrt{\pi\ell/4}/2^{20}$, with standard deviation $0.558\sqrt{\pi\ell/4}/2^{20}$. The median was $29 \approx 1.0565\sqrt{\ell \log 2}/2^{20}$. The complete distribution is shown in Figure 6.1.

We then ran 257241 experiments for our 55-bit challenge. The number of distinguished points was on average $163.37 \approx 1.0074\sqrt{\pi\ell/4}/2^{20}$, with standard

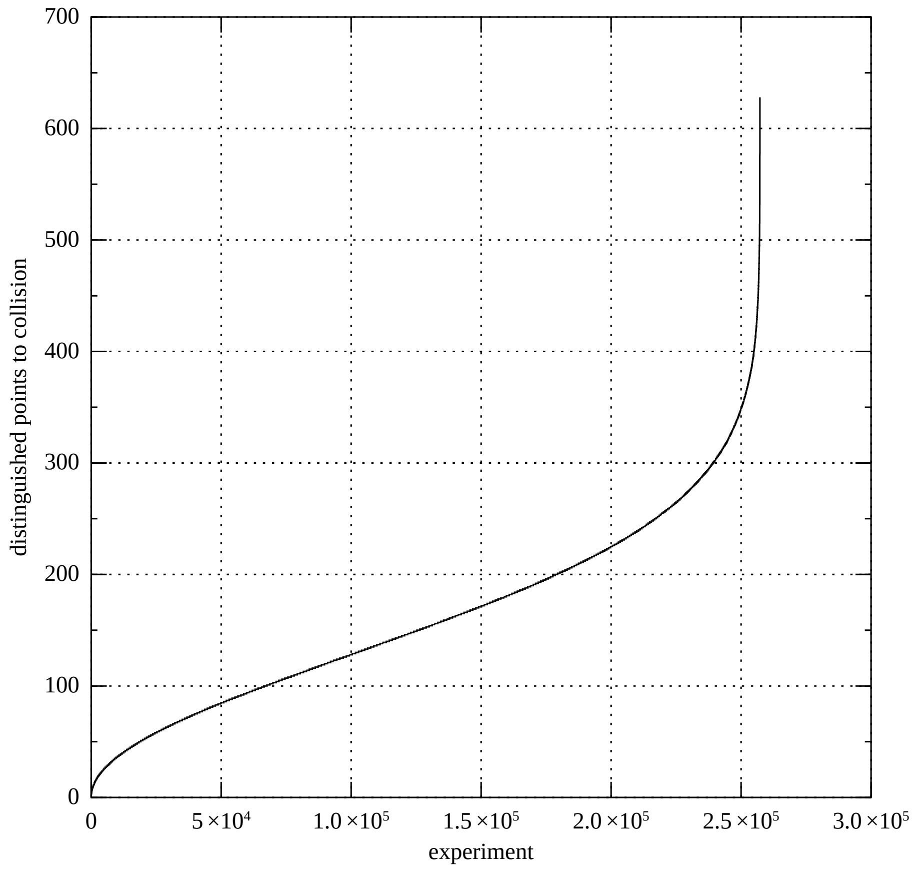

Fig. 6.2. Distribution of the number of distinguished points required to find a collision in 257241 independent experiments in a subgroup of size $\approx 1.0219 \cdot 2^{55}$ of the curve $y^2 = x^3 - 3x + 372^2$. See Figure 6.1 for an explanation of how the distribution is graphed.

deviation $0.527\sqrt{\pi\ell/4}/2^{20}$. The median was $152 \approx 0.9977\sqrt{\ell \log 2}/2^{20}$. The complete distribution is shown in Figure 6.2.

We similarly ran 33791 experiments for our 60-bit challenge. The number of distinguished points was on average $920.36 \approx 0.9996\sqrt{\pi\ell/4}/2^{20}$, with standard deviation $0.525\sqrt{\pi\ell/4}/2^{20}$. The median was $864 \approx 0.9989\sqrt{\ell \log 2}/2^{20}$. The complete distribution is shown in Figure 6.3.

Performance extrapolations. On the basis of these experiments we confidently predict that our software would solve the secp112r1 ECDLP in, on average, 37.3 years on a PS3, using $2^{35.71}$ distinguished points, requiring under 1 terabyte of storage. Here $2^{35.71}$ is calculated as $\sqrt{\pi\ell/4}/2^{20}$ with $\ell \approx p \approx 2^{128}/76439$, and 37.3 is calculated as $\sqrt{\pi\ell/4}/(2^{20} \cdot 47.94 \cdot 86400 \cdot 365.25)$.

The software runs in parallel on many PS3s without trouble, and will easily scale beyond the size of the cluster used in [5]. The computation time is inversely proportional to the number of machines, except for a few minutes at the end of the computation (by all machines while the final collision walks towards a

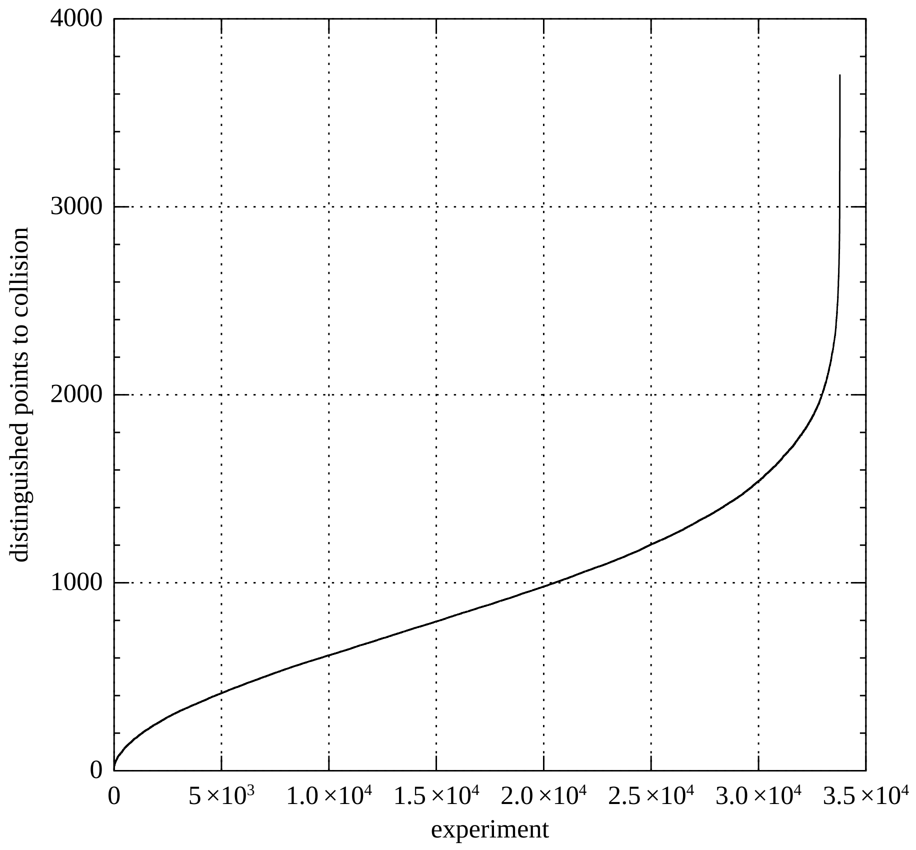

Fig. 6.3. Distribution of the number of distinguished points required to find a collision in 33791 independent experiments in a subgroup of size $\approx 1.0294 \cdot 2^{60}$ of the curve $y^2 = x^3 - 3x + 240^2$. See Figure 6.1 for an explanation of how the distribution is graphed.

distinguished point, and by a central machine recomputing the walks involved in the collision).

One can trivially reduce the storage and communication requirements by, e.g., a factor of 16 by changing the definition of distinguished points to use 24 bits instead of 20. This increases the final few minutes by a factor of 16, but it also saves almost 15 cycles of communication cost for each iteration, as discussed in the previous section, reducing the total time to just 35.6 years on a PS3.

Comparison to previous work. Our speed is directly comparable to, and almost twice as fast as, the speed previously reported by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery.

Specifically, [4, Appendix A.5] reports that the expected number of iterations "is $\sqrt{\pi q/2} \approx 8.4 \cdot 10^{16}$, where q is the prime group order", to solve a secp112r1 ECDLP; each iteration consumes 456 cycles, totalling "about 60 PS3 years". This iteration count (also appearing in [5, Section 5.3]) is slightly too optimistic: the additive walk in [3] uses $r = 16$, creating a noticeable nonrandomness penalty of approximately $1/\sqrt{1 - 1/16}$. The Cell runs at 3.192 GHz, so a better estimate is

$$\frac{456\sqrt{\pi\ell/2}}{6 \cdot 3.192 \cdot 10^9 \cdot 86400 \cdot 365.25 \cdot \sqrt{1-1/16}} \approx 65.16$$

PS3 years. We have shown how to solve the same ECDLP using just 35.6 PS3 years.

References

[1] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier Van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gürkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, Bo-Yin Yang, *Breaking ECC2K-130* (2010). URL: http://eprint.iacr.org/2009/541/. Citations in this document: §2, §2.

[2] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [30] (2006), 207–228. URL: http://cr.yp.to/papers.html#curve25519. Citations in this document: §4.

[3] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, Peter L. Montgomery, *PlayStation 3 computing breaks* 2^{60} *barrier; 112-bit prime ECDLP solved* (2009). URL: http://lacal.epfl.ch/112bit_prime. Citations in this document: §1, §1, §4, §4, §5, §6, §6.

[4] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, Peter L. Montgomery, *On the security of 1024-bit RSA and 160-bit elliptic curve cryptography: version 2.1* (2009). URL: http://eprint.iacr.org/2009/389/. Citations in this document: §1, §4, §5, §5, §5, §5, §5, §5, §6, §6.

[5] Joppe W. Bos, Marcelo E. Kaihara, Peter L. Montgomery, *Pollard rho on the PlayStation 3*, Workshop record of SHARCS'09 (2009), 35–50. URL: http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf. Citations in this document: §1, §1, §4, §5, §5, §5, §5, §6, §6, §6.

[6] Joppe W. Bos, Thorsten Kleinjung, Arjen K. Lenstra, *On the use of the negation map in the Pollard rho method*, in ANTS 2010 [15] (2010), 66–82. Citations in this document: §1, §1, §2, §2, §6, §A, §A, §A, §A, §A, §A, §A.

[7] Certicom Research, *Certicom ECC challenge* (1998). URL: http://web.archive.org/web/19981207032405/www.certicom.com/chal/download/chall_wd.doc. Citations in this document: §A.

[8] Certicom Research, *Speeding up the parallelized Pollard lambda attack on binary anomalous curves (Koblitz curves)*, 2 December 1998 archive (1998). URL: http://web.archive.org/web/19981202004435/www.certicom.com/chal/note.htm. Citations in this document: §A.

[9] Certicom Research, *Exercise lists and challenge lists* (1998). URL: http://web.archive.org/web/19981203005914/www.certicom.com/chal/ch4.htm. Citations in this document: §A.

[10] Certicom Research, *SEC 2: Recommended elliptic curve domain parameters* (2000). URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=53B19EF2A189CAA598D073BAAAAD959B?doi=10.1.1.128.27&rep=rep1&type=pdf. Citations in this document: §4.

[11] Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine*, in Africacrypt 2009 [24] (2009), 368–385. URL: http://cryptojedi.org/users/peter/#celldh. Citations in this document: §4.

[12] Iwan M. Duursma, Pierrick Gaudry, François Morain, *Speeding up the discrete log computation on curves with automorphisms*, in Asiacrypt 1999 [20] (1999), 103–121. Citations in this document: §2, §2, §A, §A, §A, §A.

[13] Adrian Escott, *Implementing a parallel Pollard rho attack on ECC* (1998). URL: http://www.cacr.math.uwaterloo.ca/conferences/1998/ecc98/escott.ps. Citations in this document: §A, §A, §A.

[14] Robert P. Gallant, Robert J. Lambert, Scott A. Vanstone, *Improving the parallelized Pollard lambda search on anomalous binary curves.*, Mathematics of Computation **69** (2000), 1699–1705. Citations in this document: §2, §A, §A, §A, §A, §A, §A, §A.

[15] Guillaume Hanrot, François Morain, Emmanuel Thomé (editors), *Algorithmic number theory, 9th international symposium, ANTS-IX, Nancy, France, July 19–23, 2010, proceedings*, Lecture Notes in Computer Science, 6197, Springer, 2010. See [6].

[16] Robert J. Harley, *Solution to Certicom's ECC2K-95 problem (email message)* (1998). URL: http://cristal.inria.fr/~harley/ecdl5/ECC2K-95.submission.text. Citations in this document: §2, §A, §A, §A.

[17] Robert J. Harley, *Big calculations (email message)* (1999). URL: http://www.xent.com/FoRK-archive/sept99/0000.html. Citations in this document: §A.

[18] E. T. A. Hoffmann, *Lebens-Ansichten des Katers Murr nebst fragmentarischer Biographie des Kapellmeisters Johannes Kreisler in zufälligen Makulaturblättern*, Dümmler, 1820–1822. URL: http://gutenberg.spiegel.de/etahoff/murr1/book.xml. Citations in this document: §1.

[19] IBM DeveloperWorks, *Cell Broadband Engine Programming Handbook, Including the PowerXCell 8i Processor (version 1.11)*, 2008. URL: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D. Citations in this document: §5.

[20] Kwok-Yan Lam, Eiji Okamoto, Chaoping 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*, Lecture Notes in Computer Science, 1716, Springer, 1999. See [12].

[21] Peter L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), 243–264. ISSN 0025–5718. MR 88e:11130. URL: http://links.jstor.org/sici?sici=0025-5718(198701)48:177<243:STPAEC>2.0.CO;2-3. Citations in this document: §3.

[22] Paul C. van Oorschot, Michael Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28. ISSN 0933–2790. URL: http://members.rogers.com/paulv/papers/pubs.html. Citations in this document: §2.

[23] John M. Pollard, *Monte Carlo methods for index computation mod p*, Mathematics of Computation **32** (1978), 918–924. ISSN 0025–5718. MR 58:10684. Citations in this document: §2.

[24] Bart Preneel (editor), *Progress in cryptology — AFRICACRYPT 2009, second international conference on cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009, proceedings*, Lecture Notes in Computer Science, 5580, Springer, 2009. See [11].

[25] Sony Corporation, *Cell Broadband Engine architecture, Version 1.01*, 2006. URL: http://cell.scei.co.jp/pdf/CBE_Architecture_v101.pdf. Citations in this document: §5.

[26] Stafford Tavares, Henk Meijer (editors), *Selected areas of cryptography 1998*, Lecture Notes in Computer Science, 1556, Springer, 1999. See [29].

[27] Edlyn Teske, *On random walks for Pollard's rho method*, Mathematics of Computation **70** (2001), 809–825. Citations in this document: §2.

[28] Michael J. Wiener, Robert J. Zuccherato, *Faster attacks on elliptic curve cryptosystems* (1998); see also newer version [29]. URL: http://grouper.ieee.org/groups/1363/Research/contributions/attackEC.ps. Citations in this document: §A, §A, §A.

[29] Michael J. Wiener, Robert J. Zuccherato, *Faster attacks on elliptic curve cryptosystems*, in SAC 1998 [26] (1999), 190–200; see also older version [28]. Citations in this document: §2, §A, §A, §A, §A.

[30] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography — 9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. See [2].

A How not to use negation in Pollard's rho method

The literature on the negation map contains many concrete proposals for using the negation map in Pollard's rho method, combining several ideas for avoiding, detecting, and escaping fruitless cycles. This section reviews and compares the proposals.

Pre-history. Solutions to Certicom's first five ECDLP challenges, ECCp-79, ECC2-79, ECCp-89, ECC2-89, and ECCp-97, were completed by Harley et al. on 6 December 1997, 16 December 1997, 12 January 1998, 7 February 1998, and 16 March 1998 respectively. The first publication of a negation method, as far as we know, was after all five of these computations were completed.

Multiplicative walks. One way to exploit negation while avoiding fruitless cycles is to use a multiplicative walk: hash W_i to a small set of scalars such as $\{2, 3, 5, 7, 11, 13, 17, 19\}$, and multiply the resulting scalar by W_i to obtain W_{i+1}. This walk is defined on equivalence classes modulo negation, and has negligible probability of entering fruitless cycles (if the set of scalars has no short multiplicative relations), so it reduces the average number of iterations from $(1 + o(1))\sqrt{\pi\ell/2}$ to $(1 + o(1))\sqrt{\pi\ell/4}$.

This negating multiplicative walk is still not competitive with non-negating additive walks. The problem is that a typical step, such as multiplying a curve point by 7, is much more expensive than a single curve addition, outweighing the negation speedup.

However, a Koblitz curve allows multiplications by $1 + \sigma$, $1 + \sigma^2$, etc. at the same speed as a single curve addition, where σ is the Frobenius endomorphism. Even better, the multiplicative walk is defined on equivalence classes modulo $\{\pm 1, \pm\sigma, \pm\sigma^2, \ldots\}$, producing an overall speedup close to $\sqrt{2d}$ for a Koblitz curve over $\mathbb{F}_{2^d}$.

The solution of ECC2K-95 by Harley et al., completed on 21 May 1998 and announced in [16], was the first successful application of automorphisms to an ECDLP challenge. The 2 April 1998 update of Harley's software says

```
Put in multiplicative pseudo-random iteration on orbits under the
involution [-1] and the Frobenius endomorphism [w] = [(-1+sqrt(-7))/2].
- Does 125 K iterations per second with "gcc -O4" + Linux + 500Mhz Alpha
(down from 205 K but number needed is less by factor sqrt(194)).
```

This software hashed W_i to some $j \in \{0, 4, 10, 11, 12\}$ and then multiplied W_i by $1 + \sigma^j$ to obtain W_{i+1}. This was joint work by Harley and Ari Singer in January 1998, according to [17].

In the meantime, an independent investigation by Gallant, Lambert, and Vanstone had prompted Certicom to suddenly reduce its published attack-time estimates for ECC2K-95 and other Koblitz curves by an order of magnitude. The reduction factor was $\sqrt{2 \cdot 97}$ for ECC2K-95, $\sqrt{2 \cdot 109}$ for ECC2K-108, $\sqrt{4 \cdot 131}$ for ECC2K-130, $\sqrt{4 \cdot 163}$ for ECC2K-163, $\sqrt{4 \cdot 239}$ for ECC2K-238, and $\sqrt{2 \cdot 359}$ for ECC2K-358; compare [9] to [7, Section 15].

An accompanying web page [8], dated 23 March 1998, claimed a $\sqrt{2d}$ speedup for Koblitz curves from normalizing x-coordinates of distinguished points modulo powers of σ, and a $\sqrt{2}$ speedup for general curves from normalizing x-coordinates of distinguished points modulo negation. However, as pointed out in [28, Section 6] two weeks later, these normalizations do not actually produce any speedup unless the walk is also modified to be compatible with the action of σ.

According to [16], the Certicom web page was subsequently updated to add a preprint of the paper [14], presenting a suitable multiplicative walk for Koblitz curves. However, as discussed above, multiplicative walks do not produce a $\sqrt{2}$ speedup for general curves. Harley stated in [16] that a multiplicative walk "can be used with any curve but only rarely gives a speedup"; Certicom did not apply a $\sqrt{2}$ reduction to its published attack-time estimates for most curves.

Additive walks with 2-cycle detection and hash modification. In [28] Wiener and Zuccherato independently claimed a speedup of $\sqrt{2}$ for arbitrary curves, and $\sqrt{2d}$ for Koblitz curves over $\mathbb{F}_{2^d}$, using an additive walk. See also the more recent version [29].

Wiener and Zuccherato pointed out that 2-cycles appear frequently in the simplest negating additive walk, $W_{i+1} = |W_i + R_{h(W_i)}|$. They suggested detecting and escaping these 2-cycles as follows. Check whether $h(W_{i+1})$ matches $h(W_i)$. If it does, then retroactively adjust h by incrementing $h(W_i)$, and go back to the computation of W_{i+1}. If r is not very small then this adjustment will not have to be repeated more than a few times for each i.

Duursma, Gaudry, and Morain showed in a followup analysis [12, Appendix A] that 2-cycles occur with probability $\Theta(1/r)$, that 4-cycles occur with probability $\Theta(1/r^2)$, etc. Wiener and Zuccherato do not consider, detect, or escape 4-cycles, so a cycle can be expected to occur within $\Theta(r^2)$ iterations: e.g., millions of iterations for $r = 2048$.

We comment that this is not necessarily a disaster. Setting a reasonable upper limit on walk length in parallel collision search was standard practice at the time of [28]; all walks that enter cycles, fruitless or otherwise, are aborted. If the distinguished-point frequency is much larger than $1/r^2$ then almost all of the Wiener–Zuccherato walks terminate with a distinguished point and the time wasted by aborted walks is very small. Presumably this is what happened in the successful experiment reported in [29, Section 5, last four lines].

However, being forced to use a distinguished-point frequency much larger than $1/r^2$ means that there will be many more than $\sqrt{\ell}/r^2$ distinguished points by the end of the computation. This poses increasingly severe communication problems and storage problems as ℓ grows, and those problems mean that for large ℓ the Wiener–Zuccherato method is unable to compete with non-negating walks.

Wiener and Zuccherato also suggested the possibility of "looking ahead two steps instead of just one" to further reduce the frequency of cycles. A detailed discussion of this possibility, without credit to [29], appeared much later in [6, page 75].

Frequent doublings and cycle detection. A few months later, in slides [13, pages 23–27] presented at ECC'98 in September 1998, Escott suggested explicitly checking for cycles of any length.

Escott detected cycles by comparing points, rather than comparing hash values. For example, Escott detected 2-cycles by comparing W_i to W_{i-2}, while Wiener and Zuccherato would have skipped the computation of W_i if $h(W_{i-1}) = h(W_{i-2})$. The Wiener–Zuccherato approach saves time when a 2-cycle is found, but loses time in the equally common case that $W_i - W_{i-1} = W_{i-1} - W_{i-2}$. On the other hand, one could sort and compare hash values together with signs to robustly check for 2-cycles, 4-cycles, etc.

Escott used a mixed walk that includes doublings, and checked for cycles only back to the most recent doubling, using the heuristic that "No small useless cycle includes a double." (The statement "A cycle with at least one doubling is most likely not fruitless" appeared much later in [6] without credit to Escott.) This still leaves many comparisons: for example, if the 16 operations corresponding to 16 hash values are 15 additions of precomputed points and 1 doubling, then there will typically be 10 or 20 points after the most recent doubling, and thus 10 or 20 comparisons to those points. Eliminating branches would obviously make this computation even more expensive.

Escott escaped a cycle by walking to a new point computed from the points in the cycle. Escott emphasized the importance of rotational symmetry, having the new point be independent of where the cycle was entered, so that "no collisions are lost." It is not clear exactly which new point was used for the successful experiments reported in [13, page 27]. Previous comments in [13] seem to suggest simply adding the points in the cycle, but this is highly nonrandom; for example, adding W to $-W - R_j$ produces $-R_j$, no matter what W is.

Delayed cycle detection. Another discussion of fruitless cycles appeared in the paper [14, Sections 5–6] by Gallant, Lambert, and Vanstone, submitted for publication in June 1998 and revised in October 1998.

To detect cycles in "labels" (e.g., cycles in x-coordinates), Gallant, Lambert, and Vanstone proposed to "intermittently save labels and detect repetitions by comparing new labels against these stored ones." They suggested that the "save interval" I be small, perhaps 10 or 20, since longer cycles are unlikely to occur. Saving the current label every I iterations, and comparing the new label to the saved label every iteration, detects all cycles of length at most I.

Note that the Gallant–Lambert–Vanstone method often needs more iterations than the Escott method to detect a cycle. For example, a length-2 cycle is detected after just 2 steps if it includes the saved label, but otherwise it waits for the next label, continuing to cycle for nearly I extra iterations. The compensating advantage is that each iteration is faster, involving only 1 comparison. The paper also suggests that one can store "different numbers of past values" at "different intervals" but does not elaborate.

Gallant, Lambert, and Vanstone recommended escaping a cycle by applying a "modified iteration" to the "lexicographically least label" in the cycle. The lexicographically least label has the obvious virtue of being easy to compute. The subsequent paper [6] says that the modified iteration in [14] consists of adding a precomputed point (from another pool of points, not the points used in unmodified iterations), and that this has a good chance of returning to the same cycle, rendering the method non-functional. However, [14] is actually much more vague than this; it does not specify a modified iteration. There exist modified iterations that work correctly, but [14] certainly does not make clear that the success of the method depends heavily on the choice of the iteration.

Statistical cycle detection and escape by doubling. Duursma, Gaudry, and Morain in [12, pages 109–110] proposed sorting the elements of a cycle, obtaining $C_1 \le C_2 \le \cdots \le C_t$, and then walking to $2C_1 + 5C_2 + \cdots + (t^t + 1)C_t$. Walking simply to $2C_1$ was proposed much later in [6], without credit to [12]. Neither method suffers from recurring cycles.

Combining the detection method from [14] with the escape method from [12] would have produced a reasonably efficient negating rho method. However, it still would not have produced a reasonably efficient *branchless* negating rho method; the possibility of having to escape a cycle at any moment causes trouble for SIMD architectures.

The same paper suggested detecting cycles by checking whether "the number of distinguished points" does not "evolve as prescribed by the theory." This suggestion, like the Wiener–Zuccherato algorithm, is not competitive for large ℓ: it cannot be efficient unless the distinguished-point frequency is large.

More possibilities. The much more recent paper [6] by Bos, Kleinjung, and Lenstra is the first paper on this topic that does *not* claim a $\sqrt{2}$ speedup. The paper reports a table of 126 different non-SIMD implementation results: specifically, 21 different combinations of cycle-detection methods, hash-modification methods, and cycle-escape methods for 6 different values of r.

The most efficient algorithm identified in [6] is the following: compare and modify hash values as in [29] to reduce the frequency of 2-cycles; "after α steps record a length β sequence of successive points and compare the next point to these β points" (miscredited to [14]) to detect any remaining cycle; escape any remaining cycle by doubling the smallest element of the cycle.

The comparison and potential modification of hash values at each iteration makes this algorithm perform quite poorly on SIMD architectures. Eliminating those modifications, and relying solely on the α-β cycle-detection mechanism, would also perform poorly: 2-cycles would arise frequently, so α would have to be small to avoid excessive delays in detecting 2-cycles, while β would have to be at least 4 to catch 4-cycles, requiring frequent comparisons and conditional operations along with several stored labels for each walk.

Given the history surveyed above, it was not unreasonable of [6] to dispute the speedup claims made in the earlier literature, and to question the possibility of any speedup on SIMD architectures.