Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction

Joppe W. Bos
Marcelo E. Kaihara
Thorsten Kleinjung
Arjen K. Lenstra

Laboratory for Cryptologic Algorithms,
École Polytechnique Fédérale de Lausanne,
Station 14, CH-1015 Lausanne, Switzerland

Peter L. Montgomery

One Microsoft Way,
Microsoft Research,
Redmond, WA 98052, USA

Abstract We describe a Cell processor implementation of Pollard's rho method to solve discrete logarithms in groups of elliptic curves over prime fields. The implementation was used on a cluster of PlayStation 3 game consoles to set a new record. We present in detail the underlying single instruction multiple data modular arithmetic.

Keywords: Elliptic curve discrete logarithm; Pollard's rho method; Cell processor; SIMD; negation map

1 Introduction

The security of elliptic curve cryptography (ECC) [36, 40] is based on the difficulty of the *elliptic curve discrete logarithm problem* (ECDLP). The two most common versions use *prime fields* and *binary extension fields*, respectively. Currently, the best methods to solve large scale ECDLPs are variants of Pollard's rho method [44].

We describe an implementation of Pollard's rho method to solve prime field ECDLPs on the Cell processor, the processor that is the heart of the Sony PlayStation 3 (PS3) game console. The underlying modular arithmetic is targeted at single instruction multiple data (SIMD) platforms and is mostly branch-free. It can take advantage of prime moduli of a special form using efficient "sloppy reduction." We used the implementation to set a new prime field ECDLP record for a 112-bit prime of the proper special form. The calculation was performed on EPFL's cluster of about 215 PS3s.

The previous prime field ECDLP record, reported in 2002, involved a 109-bit prime [17]. The following may explain the apparent lack of interest to set new ECDLP records. The expected cost to solve a particular ECDLP on any combination of platforms can be extrapolated from a relatively short calculation, given implementations of Pollard's rho method. Cryptographically relevant ECDLPs turn out to be firmly out of reach, despite occasional improvements of Pollard's rho method. Given easy estimation of overall cost and infeasibility of cryptographically relevant problems, not much is gained by solving an ECDLP, in particular of a cryptographically irrelevant size. This is unlike integer factorization where the only convincing way to show the feasibility and estimate the cost of a record-breaking calculation is completing it (cf. the orders of magnitude difference between the actual cost reported in [33] and the estimate in [45]).

Besides investigating algorithmic improvements, it is relevant to study if and how new processor architectures affect the cost of solving ECDLPs. This concerns any device that allows implementation of Pollard's rho method and occasional communication of its results. Per device, it suffices to provide an implementation, to measure its long term yield, and to extrapolate. An example of such a study focuses on binary extension field ECDLPs [2]. It comprises various types of desktop processors, the Cell processor, graphics cards, field programmable gate arrays, and application specific integrated circuits. It is even possible to harvest cycles on cellphones [52], but it remains to be seen if that affects ECDLP feasibility.

We present a parallelized implementation of Pollard's rho method on a cluster of PS3 game consoles, devices that are relatively inexpensive given their processing power. The parallelization exists on five distinct levels: each PS3 runs independently of all others, on each

PS3's Cell processor six cores work independently of each other, and each of these cores simultaneously runs 50 times two interleaved 4-fold SIMD processes. The top two levels are merely 'embarrassingly parallel', the first at the physical PS3 level, the other provided by the Cell processor's multi-core design. The 50 simultaneous copies serve to amortize a high cost modular inversion, interleaving is done to improve throughput, and 4-way SIMD exploits the core's arithmetic instruction set.

The first projects on EPFL's PS3 cluster concerned cryptographic hash collisions [49]. To ascertain the cluster's reliability and stability for projects requiring long integer arithmetic, a rough version of Pollard's rho method for prime field ECDLP was run for a few weeks. Because it turned out to work satisfactorily and because no other project was ready to be deployed, it was left running. As this soon led to the completion of a non-negligible fraction of the total expected work for the ECDLP at hand, it was decided to further optimize the code and, some misgivings notwithstanding, to attempt to solve it. Although our choice of improvements that could be carried through was limited by the early design decisions (as we did not want to start afresh), the overall expected runtime was reduced by more than 60% in the course of the calculation. It may be further reduced by adopting a variety of changes in the initial design [6].

Apart from the prime field ECDLP record, we consider if the method from [19] can be applied to ECDLP as well, and we present an efficient 4-way SIMD binary modular inversion and a fast branch-free sloppy reduction and normalization modulo primes of the form $\frac{2^{32\ell}\pm m}{c}$, for relatively small $\ell, m, c \in \mathbf{Z}_{>0}$. These methods are designed for cryptanalytic applications in a SIMD environment. The sloppy reduction may not be suitable for cryptographic applications, because it can produce an incorrect result. When solving ECDLPs, however, it suffices if calculations are most of the time correct: as expected based on our heuristics, sloppy reduction never produced an incorrect result. Many of these methods can be used on SIMD platforms other than the Cell processor, such as graphics cards. Finally, this paper led to the study of the negation map reported in [13].

This paper is organized as follows. Section 2 presents the general ECDLP, with Section 2.1 defining the prime field ECDLP that is solved in this article. Section 3 describes the state of the art of Pollard's rho method and our design choices, and includes some of our findings from [13]. The Cell implementation is described in Section 4: sloppy reduction in Section 4.4, the new ECDLP record in Section 4.6, and a discussion of potential implications for larger prime fields in Section 4.7.

Related work. A Cell processor implementation of Pollard's rho method to solve a particular type of binary extension field ECDLPs (namely, for Koblitz curves [37]) is presented in [15] and is part of the larger implementation project [2] cited above. As far as we are aware ours is the first paper describing a Cell processor implementation of Pollard's rho method to solve prime field ECDLPs, though parts of it were prepublished as [12]. Our modular arithmetic uses methods from [9]. Other efficient methods for modular arithmetic on the Cell appeared in [21, 20, 11]. Applications of such arithmetic to run the elliptic curve factorization method [39] on the Cell have been explored in [7] for small numbers in the number field sieve [38] cofactorization step, and in [14] to factor (large) composites of the form $2^M - 1$.

2 The prime field ECDLP

Let $\mathbf{F}_p$ denote a finite field of prime cardinality $p > 3$. Any $a, b \in \mathbf{F}_p$ with $4a^3 + 27b^2 \neq 0$ define an elliptic curve $E_{a,b}$ over $\mathbf{F}_p$. The *group of points* $E_{a,b}(\mathbf{F}_p)$ *of* $E_{a,b}$ *over* $\mathbf{F}_p$ is defined as the *zero point* $\mathfrak{o}$ along with the set of pairs $(x, y) \in \mathbf{F}_p \times \mathbf{F}_p$ that satisfy the *short Weierstrass equation* $y^2 = x^3 + ax + b$, with the following additively written group law. For $\mathfrak{c} \in E_{a,b}(\mathbf{F}_p)$ define $\mathfrak{c} + \mathfrak{o} = \mathfrak{o} + \mathfrak{c} = \mathfrak{c}$. For non-zero $\mathfrak{c} = (x_1, y_1), \mathfrak{d} = (x_2, y_2) \in E_{a,b}(\mathbf{F}_p)$ define $\mathfrak{c} + \mathfrak{d} = \mathfrak{o}$ if $x_1 = x_2$ and $y_1 = -y_2$. Otherwise $\mathfrak{c} + \mathfrak{d} = (x, y)$ with $x = \lambda^2 - x_1 - x_2$ and $y = \lambda(x_1 - x) - y_1$, where $\lambda = \frac{3x_1^2 + a}{2y_1}$ if $x_1 = x_2$ (and thus $\mathfrak{c} = \mathfrak{d}$) and $\lambda = \frac{y_1 - y_2}{x_1 - x_2}$ otherwise. Thus, using these *affine Weierstrass coordinates* to represent group elements, *doubling* (i.e., $\mathfrak{c} = \mathfrak{d}$) is different from *regular addition* (i.e., $\mathfrak{c} \neq \mathfrak{d}$).

Let $\mathtt{A}$, $\mathtt{I}$, $\mathtt{M}$, and $\mathtt{S}$ denote the average cost of addition, inversion, multiplication, and squaring, respectively, in $\mathbf{F}_p$. The cost of regular addition in $E_{a,b}(\mathbf{F}_p)$ is $6\mathtt{A} + \mathtt{I} + 2\mathtt{M} + \mathtt{S}$, and doubling can be done in $8\mathtt{A} + \mathtt{I} + 2\mathtt{M} + 2\mathtt{S}$. For relevant sizes of p it is safe to assume that $\mathtt{I}$ is much larger than $\mathtt{M}$, i.e., at least $\mathtt{I} > 5\mathtt{M}$ when using software (in hardware the difference can be made smaller [30]). Our choice of curve parametrization and arithmetic is further explained and discussed below.

Let p, a, b and $\mathfrak{g} \in E_{a,b}(\mathbf{F}_p)$ of prime order q be given such that the index $[E_{a,b}(\mathbf{F}_p) : \langle\mathfrak{g}\rangle]$ is small. For $\mathfrak{h} \in \langle\mathfrak{g}\rangle$ the ECDLP is to find an integer m such that $m\mathfrak{g} = \mathfrak{h}$. For curves without special properties, solving ECDLP is believed to require an effort on the order of $\sqrt{q}$.

To find the least positive m with $m\mathfrak{g} = \mathfrak{h}$, *Shanks' baby step giant step* method [35, Exercise 5.25] builds a hash table containing $i\lceil\sqrt{q}\rceil\mathfrak{g}$ for $i = 0, 1, \ldots, \lceil\sqrt{q}\rceil$ and searches it for $\mathfrak{h} + j\mathfrak{g}$ for $j = 0, 1, 2, \ldots,$ until a match is found. This works in time and memory on the order of $\sqrt{q}$ which can both be reduced to $\sqrt{m}$. Pollard's rho method (Section 3) achieves expected runtime $O(\sqrt{q})$ and $O(\log q)$ memory or, if run in parallel, much less memory than Shanks' method: $O((\log q)^2)$ memory suffices [25, Exercise 16.23] when roughly $\sqrt{q}\log q$ out of q group elements are distinguished (Section 3.1).

2.1 The 112-bit prime field ECDLP

This article concentrates on curve "secp112r1" from [18]. Let $R = 2^{128}$ and $\tilde{p} = R - 3$, then $p = \frac{\tilde{p}}{11 \cdot 6949}$ is prime. The elliptic curve $E_{a,b}$ over $\mathbf{F}_p$ defined by $a = p - 3$ and

$$b = 2061118396808653202902996166388514$$

has a group $E_{a,b}(\mathbf{F}_p)$ of prime order $q = p + 1 + 4407293269000505$ and is generated by

$$\mathfrak{g} = (\,188281465057972534892223778713752,\\ 3419875491033170827167861896082688).$$

This curve and generator were created "verifiably at random" [18], implying that solving ECDLP in $\langle \mathfrak{g} \rangle = E_{a,b}(\mathbf{F}_p)$ should not be unexpectedly easy due to a built-in trapdoor. Because no corresponding challenge ECDLP was included in [18], we defined one ourselves in a "verifiably not pre-cooked" manner by taking $\mathfrak{h} = (x, y) \in \langle \mathfrak{g} \rangle$ for an unforgeable value of x. With $x = \lfloor (\pi - 3)10^{34} \rfloor$, this leads to the 112-bit prime ECDLP where

$$\mathfrak{h} = (\,1415926535897932384626433832795028,\\ 3846759606494706724286139623885544) \in E_{a,b}(\mathbf{F}_p),$$

is given and an m with $m\mathfrak{g} = \mathfrak{h}$ must be found.

3 Pollard's rho method

3.1 Random walks and parallelization

If objects are selected at random and with replacement from q objects, the conditional probability at step $n+1$ of finding the first duplicate (or *collision*) is $\frac{n}{q}$ (if $n < q$). Via straightforward arguments (e.g. [34, Exercise 3.1.12]) this leads to $\sqrt{\frac{\pi q}{2}}$ for the expected number of steps (or *iterations*) until the first collision. This result is generally referred to as the *birthday paradox*. For arbitrary multipliers u, v, the object $u\mathfrak{g} + v\mathfrak{h} \in \langle \mathfrak{g} \rangle$. A collision corresponds to random integer multipliers $u, v, \bar{u}, \bar{v}$ such that $u\mathfrak{g} + v\mathfrak{h} = \bar{u}\mathfrak{g} + \bar{v}\mathfrak{h}$. Unless $\bar{v} \equiv v \bmod q$, the value $m = \frac{u - \bar{u}}{\bar{v} - v} \bmod q$ solves the discrete logarithm problem. The expected number of steps of this idealized version of Pollard's rho method [44] is $\sqrt{\frac{\pi q}{2}}$.

***r*-adding and *r* + *s*-mixed walks.** Pollard's rho method uses an approximation of a truly random walk in $\langle \mathfrak{g} \rangle$. An index function $l : \langle \mathfrak{g} \rangle \mapsto [0, r-1]$ is chosen, for some small integer r, such that the l-induced r-partition $\langle \mathfrak{g} \rangle = \cup_{i=0}^{r-1} \mathfrak{G}_i$, where $\mathfrak{G}_i = \{\mathfrak{x} : \mathfrak{x} \in \langle \mathfrak{g} \rangle, l(\mathfrak{x}) = i\}$, results in $\mathfrak{G}_i$'s of approximately the same cardinality. For random integer multipliers u_i, v_i, *addition constants* $\mathfrak{f}_i = u_i\mathfrak{g} + v_i\mathfrak{h} \in \langle \mathfrak{g} \rangle$ are pre-computed for $0 \le i < r$, and the starting point of the walk is selected as a random but known multiple of $\mathfrak{g}$. Given a point $\mathfrak{p}$ of the walk calculate $\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})} \in \langle \mathfrak{g} \rangle$ as the next point. This is called an *r-adding walk*. It is easy to keep track of the integer multipliers $u, v \in \{0, 1, \ldots, q-1\}$ such that $\mathfrak{p} = u\mathfrak{g} + v\mathfrak{h}$. Finding a duplicate can be done by *Floyd's cycle finding method* [34, Exercise 3.1.6] requiring only a constant number of group elements and integer multipliers: compute $(\mathfrak{p}_k, \mathfrak{p}_{2k})$ for $k = 1, 2, \ldots$ (where $\mathfrak{p}_k$ denotes the kth point of the walk) until a collision occurs, i.e., $\mathfrak{p}_k = \mathfrak{p}_{2k}$. A stack-based cycle detection method is described in [43].

As shown by the following heuristic analysis from [2, Appendix B], which refines the arguments from [16], the average number of steps for an r-adding walk is somewhat larger than $\sqrt{\frac{\pi q}{2}}$. Let $p_i = \frac{\#\mathfrak{G}_i}{q}$. A point in the walk is said to be of class i if its predecessor upon its first occurrence belongs to $\mathfrak{G}_i$. If the nth point belongs to $\mathfrak{G}_j$ (with probability p_j) and the $(n+1)$st point produces the first collision, the collision point cannot be of class j (this happens with probability p_j), since then the collision would already have occurred in the previous step. Therefore, the conditional probability that the first collision occurs at step $n+1$ is heuristically assumed to be

$$\frac{n}{q}\left(1 - \sum_{j=0}^{r-1} p_j^2\right).$$

With $q' = \frac{q}{1 - \sum_{j=0}^{r-1} p_j^2}$ this probability is $\frac{n}{q'}$, so that we get via the same arguments referred to above

$$\sqrt{\frac{\pi q'}{2}} = \sqrt{\frac{\pi q}{2(1 - \sum_{j=0}^{r-1} p_j^2)}} \qquad (1)$$

as a heuristic estimate for the average number of steps until the first collision.

Pollard, in [44], uses $r = 3$ with addition constants $\mathfrak{f}_0 = \mathfrak{h}$ and $\mathfrak{f}_2 = \mathfrak{g}$, but replaces the $i = 1$ case by the doubling $2\mathfrak{p}$. Although the successive points are not independent, further undermining the arguments in the above heuristics, it was shown in [32] that with high probability a collision occurs in $\Theta(\sqrt{q})$ steps, if the partition is given by a random oracle. Teske, in [50], shows that larger r-values, such as $r = 20$, with random addition constants lead to fewer iterations and better performance on average, in accordance with the heuristics and even if none of the choices does an explicit doubling (as Pollard's $i = 1$ case). Generalization of the arguments of [32] to Teske's variant is not immediate [10].

Inclusion of doublings leads to *r + s-mixed walks*: given a function $l : \langle \mathfrak{g} \rangle \mapsto [0, r+s-1]$ that induces an $r+s$-partition of $\langle \mathfrak{g} \rangle$, the next point equals $\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})}$ if $0 \le l(\mathfrak{p}) < r$, but $2\mathfrak{p}$ if $l(\mathfrak{p}) \ge r$. The original walk by Pollard is a $2+1$-mixed walk. The above heuristics apply to this case too, if we define the doublings as a single class hit with probability $p_D = \frac{\sum_{i=r}^{r+s-1} \#\mathfrak{G}_i}{q}$ (which should be $\approx \frac{s}{r+s}$). Experiments by Teske show that best performance is achieved when $\frac{1}{4} \le \frac{s}{r} \le \frac{1}{2}$ but that mixed walks are not significantly better than r-adding ones unless $r \le 3$. Our experiments support the heuristics suggesting that the optimal ratio is close to zero (see also Table 1).

Per step the occurrence probability of the event $\mathfrak{p} = \mathfrak{f}_i$ (and thus potentially an immediate solution to the discrete logarithm problem) is negligible compared to the probability of a birthday collision. So, if r-adding as opposed to $r+s$-mixed walks are used, the possibility that doublings will occur can safely be ignored, making it efficient to SIMD-parallelize r-adding walks. This is further commented on below and exploited in Section 4.

Some types of elliptic curves allow faster variants of r-adding walks. For instance, for so-called Koblitz curves over binary extension fields (which are not covered by our definition in Section 2), the Frobenius automorphism of the finite field can be used to define an efficient function ψ on the group of points of the elliptic curve. For instance, defining the successor of point $\mathfrak{p}$ as $\psi^i(\mathfrak{p}) + \mathfrak{p}$ allows its quick computation [26].

Parallelized random walks. Parallelization of Pollard's rho method does **not** consist of running random walks in parallel until one of them collides: on M processors the expected speedup would be only a factor of $\sqrt{M}$, so it would overall require $\sqrt{M}$ more processing power than a single processor. The proper way to parallelize Pollard's rho method [51] achieves an M-fold speedup on M processors, thus requiring the same overall processing power as a single process in $\frac{1}{M}$th of the time. Different processes must be able to efficiently recognize whether, probably at different points in time, their walks have hit upon the same group element. To achieve this, each process generates a single random walk, each from its own random starting point, but all using the same index function l and the same $\mathfrak{f}_i$'s. As soon as a walk hits upon a *distinguished point*, this point is reported to a central location, along with the corresponding integer multipliers u and v. If the latter would require too much central storage, information to regenerate the starting point should be provided so that, if needed, u and v can be recalculated. The walk may start afresh from a new random starting point, or it may continue. The idea is that as soon as two walks collide – without noticing it – they will keep taking the same steps (because they use the same l and the same $\mathfrak{f}_i$'s) and will thus both ultimately reach the same distinguished point. This will be noticed when the colliding distinguished point is reported to the central location. The discrete logarithm can then be computed from the two, hopefully distinct, pairs of integer multipliers that correspond to the same distinguished point.

A point is distinguished if it has an easily recognizable property that occurs with low enough probability to make it possible to store distinguished points on disk and to efficiently find collisions, but often enough for every walk to hit a distinguished point, eventually. We required the 24 lowest-order bits of a unique representation of the x-coordinate to be zero, so that it may be expected that walks hit a distinguished point once every 2^{24} steps and that storage requires less than a terabyte of disk space. Analysis of the distinguished point property is performed in [46] where the results from [51] are reaffirmed when $\sqrt{q} \ll \frac{q}{2^k} \ll q$; i.e. the distinguished point property should be chosen in such a way that at least one distinguished point is expected in each cycle.

Unique point representation. When using Pollard's rho method, group elements must be represented in a unique way to be able to decide to which partition they belong. When using the parallelized version, uniqueness is also useful to recognize if a point is distinguished. The fastest point representations that we are aware of that are applicable are the affine ones, such as the one in Section 2. It requires an inversion in $\mathbf{F}_p$ per group operation, i.e., per step of the walk. The resulting high inversion cost is amortized over many walks running in parallel, as described below.

Simultaneous inversion. In the parallelized version of Pollard's rho method, Montgomery's *simultaneous inversion* method from [42] can be used to share the inversion with any number of synchronous but independent walks. Let n be some number of independent walks (typically all running on the same processor), and let $z_i \in \mathbf{F}_p^*$ denote the element that needs to be inverted for the computation of λ in the ith walk (with λ as in Section 2). With $w_0 = 1$, first combine the z_i's by calculating $w_i = z_i w_{i-1} \in \mathbf{F}_p^*$ for $i = 1, 2, \ldots, n$, then calculate $\bar{w} = w_n^{-1}$, and finally unravel the results: for $i = n, n-1, \ldots, 1$ in succession calculate $z_i^{-1} = \bar{w} w_{i-1}$ and replace $\bar{w}$ by $z_i \bar{w} = w_{i-1}^{-1}$. Avoiding useless multiplications, the cost $n\mathtt{I}$ of n inversions can thus be replaced by $3(n-1)\mathtt{M} + \mathtt{I}$, which usually implies a considerable saving. For Pollard's rho method it leads to an amortized cost of about $6\mathtt{A} + \frac{1}{n}\mathtt{I} + 5\mathtt{M} + \mathtt{S}$ per step per walk. This makes affine Weierstrass coordinates the least costly point representation for this type of application, if n can be chosen sufficiently large.

The disadvantage is, however, that the group operations are non-uniform: i.e. the addition and doubling are different operations. For SIMD implementation of two or more walks, this means that a regular addition step in one walk cannot be executed simultaneously with a doubling step in another walk. For regular r-adding walks this is not a problem because, as argued above, doubling steps will most likely not occur. Also, excluding $r + s$-mixed walks in a SIMD environment is not a big issue since such walks are not advantageous anyhow (in SIMD, threads could be regrouped to separate regular addition from doubling steps, but this may lead to considerable overhead). More importantly, it makes it harder to profit from the negation map, an optimization discussed in Section 3.2, in a SIMD environment, so elliptic curve parametrizations that allow identical addition and doubling operations remain relevant. Note that the one from [24] (see [8] and a series of follow-up papers) cannot be used if $\#E_{a,b}(\mathbf{F}_p)$ is prime, as in our case. Such parametrizations, and others, are preferable to affine Weierstrass coordinates in applications that heavily use scalar multiplication, such as ECC and the elliptic curve method for integer factorization [39].

3.2 Using automorphisms.

Following [53], define an equivalence relation $\sim$ on $\langle \mathfrak{g} \rangle$ by $\mathfrak{p} \sim -\mathfrak{p}$ for $\mathfrak{p} \in \langle \mathfrak{g} \rangle$. Instead of searching $\langle \mathfrak{g} \rangle$ of size q, search $\langle \mathfrak{g} \rangle / \sim$ of size about $\frac{q}{2}$, where the equivalence class containing $\mathfrak{p}$ and $-\mathfrak{p}$ is represented by, for instance, the element with y-coordinate of least absolute value. Thus, using this *negation map* one would expect to save a factor of $\sqrt{2}$ in the number of iterations, at the cost of finding the representative after each step. The latter is fast since $-(x, y) = (x, -y)$ for $(x, y) \in \langle \mathfrak{g} \rangle$. Obviously, if $-\mathfrak{p}$

Table 1 Number of steps required by Pollard's rho method in random elliptic curve groups of 32-bit prime order q over fields of random 32-bit prime cardinality p, divided by $\sqrt{\pi q/2}$ or by $\sqrt{\pi q/4}$ (without or with the negation map). Lowest and highest averages are over 10 measurements. Each measurement calculates the average number of steps taken until a collision occurs, over 100 000 collision searches where for each search a prime p and an elliptic curve over $\mathbf{F}_p$ are randomly selected until the order q of the group of points is prime. Overall average is the average of the 10 averages (thus, the average over one million searches). Expression (1) and (2) columns are the quotients as expected based on expressions (1) (with $p_i = \frac{1}{r}$ for $0 \leq i < r$) and (2) (with $p_i = \frac{1}{r+s}$ for $0 \leq i < r$ and $p_D = \frac{s}{r+s}$), respectively. Those expressions are for $q \to \infty$ and indeed for larger (smaller) q they give a better (worse) fit.

	Without negation map				With negation map			
	Averages			Expression	Averages			Expression
	lowest	overall	highest	(1)	lowest	overall	highest	(2)
8-adding	1.080	1.083	1.086	1.069	1.034	1.038	1.041	1.033
16-adding	1.034	1.036	1.039	1.033	1.013	1.016	1.019	1.016
32-adding	1.012	1.015	1.020	1.016	1.007	1.008	1.010	1.008
$16+4$-mixed	1.042	1.044	1.047	1.043	1.035	1.038	1.040	1.031
$16+8$-mixed	1.074	1.077	1.081	1.078	1.074	1.076	1.078	1.069

instead of $\mathfrak{p}$ is the representative, the integer multipliers u, v with $\mathfrak{p} = u\mathfrak{g} + v\mathfrak{h}$ must be replaced by $-u, -v$.

Adapting the earlier r-adding walk heuristics, it follows that for r-adding (or $r+s$-mixed) walks the speedup by a factor of $\sqrt{2}$ that is generally reported in the literature is slightly too pessimistic. Let the definitions of p_i, p_D, and of class i be as in Section 3.1. Assume that the nth point belongs to $\mathfrak{G}_j$ and that the $(n+1)$st point produces the first collision while hitting the representative $\mathfrak{p}$, either directly or after negation. If this step is a doubling then the same heuristics as in Section 3.1 applies. This happens with probability p_D^2. Otherwise, we only exclude the case that as a result of just the addition the two predecessors hit the same point ($\mathfrak{p}$ or $-\mathfrak{p}$). This happens with probability $\frac{p_j^2}{2}$. Therefore, the conditional probability that the first collision occurs at step $n+1$ is heuristically assumed to be

$$\frac{2n}{q}\left(1 - p_D^2 - \sum_{j=0}^{r-1}\frac{p_j^2}{2}\right).$$

As above we get

$$\sqrt{\frac{\pi q}{4(1 - p_D^2 - \frac{1}{2}\sum_{j=0}^{r-1} p_j^2)}} \qquad (2)$$

for the heuristically expected number of steps until the first collision. For the same parameter values this is more than a factor of $\sqrt{2}$ smaller than Expression (1).

Practical application of the negation map is complicated by *fruitless cycles*, as pointed out in [26, 53]. This is further discussed in Section 3.3. The group $\langle \mathfrak{g} \rangle$ may admit other trivially computable maps. For instance, for Koblitz curves the Frobenius automorphism of a degree-t binary extension field leads to a further $\sqrt{t}$-fold speedup [53, 26, 23]. This does not apply to the case considered in this article.

Small scale experimental verification. For 32-bit primes q we checked the accuracy of the predictions based on expressions (1) and (2) and list the results in Table 1. With all averages larger than 1, both r-adding and $r+s$-mixed walks on average perform worse than truly random walks. For most walks with the negation map the averages are lower than their negation-less counterparts, indicating that the reduction factor in the expected number of steps is indeed larger than $\sqrt{2}$. This does not imply a speedup by the same factor, because to obtain the figures costly fruitless cycle detection methods had to be used. It can be seen that $r+s$-mixed walks are disadvantageous if $s > \frac{r}{4}$. See [13, Table 1] for a similar table for 31-bit primes q.

3.3 Fruitless cycles

If the negation map is used, the successor of point $\mathfrak{p}$ equals $-\mathfrak{p} - \mathfrak{f}_i$ with probability $\frac{1}{2}$, for some i. Because $l(-\mathfrak{p} - \mathfrak{f}_i) = i$ with probability approximately $\frac{1}{r}$, it follows that $\mathfrak{p}$ equals its second successor with probability approximately $\frac{1}{2r}$. Thus, when combining r-adding walks with the negation map all walks can be expected to get trapped in a *fruitless 2-cycle*. Below we summarize how such and other cycles may be dealt with.

Reducing 2-cycles. To reduce occurrence of 2-cycles as above, [53] proposes an iteration function that looks ahead and tries to take the first i of $\ell(\mathfrak{p}), \ell(\mathfrak{p})+1, \ldots, \ell(\mathfrak{p})+r-1$ with $i \bmod r \neq \ell(\sim(\mathfrak{p}+\mathfrak{f}_i))$ (with indices i in $\mathfrak{f}_i$ taken modulo r): with $f : \langle \mathfrak{g} \rangle \to \langle \mathfrak{g} \rangle$ defined as

$$f(\mathfrak{p}) = \begin{cases} E(\mathfrak{p}) & \text{if } j = l(\sim(\mathfrak{p}+\mathfrak{f}_j)) \text{ for } 0 \leq j < r \\ \sim(\mathfrak{p}+\mathfrak{f}_i) & \begin{cases}\text{with } i \geq l(\mathfrak{p}) \text{ minimal such that} \\ l(\sim(\mathfrak{p}+\mathfrak{f}_i)) \neq i \bmod r,\end{cases}\end{cases}$$

define $\mathfrak{p}$'s successor as $f(\mathfrak{p})$. The function $E : \langle \mathfrak{g} \rangle \to \langle \mathfrak{g} \rangle$ may restart the walk at a new random point, which is expected to happen once every r^r steps, thus at negligible cost. Overall, the expected cost is increased by a factor of $\sum_{i=0}^{r} \frac{1}{r^i}$, which lies between $1 + \frac{1}{r}$ and $1 + \frac{1}{r-1}$.

Reducing 2- and 4-cycles. Although f reduces 2-cycles, it introduces new but less frequent 2-cycles, and does not address 4-cycles, which themselves start from a random point with probability at least $\frac{r-1}{4r^3}$ when f is used [13]. Some of these new 2-cycles and most 4-cycles are avoided by the iteration function g, which yet again introduces new (and again less frequent) short cycles:

$$g(\mathfrak{p}) = \begin{cases} E(\mathfrak{p}) & \begin{cases} \text{if } j \in \{l(\mathfrak{q}), l(\sim(\mathfrak{q} + \mathfrak{f}_{l(\mathfrak{q})}))\} \\ \text{or } l(\mathfrak{q}) = l(\sim(\mathfrak{q} + \mathfrak{f}_{l(\mathfrak{q})})) \text{ where} \\ \mathfrak{q} =\sim(\mathfrak{p} + \mathfrak{f}_j), \text{ for } 0 \leq j < r, \end{cases} \\ \mathfrak{q} =\sim(\mathfrak{p} + \mathfrak{f}_i) & \begin{cases} \text{with } i \geq l(\mathfrak{p}) \text{ minimal such that} \\ i \bmod r \neq l(\mathfrak{q}) \neq l(\sim(\mathfrak{q} + \mathfrak{f}_{l(\mathfrak{q})})) \\ \text{and } i \bmod r \neq l(\sim(\mathfrak{q} + \mathfrak{f}_{l(\mathfrak{q})})). \end{cases} \end{cases}$$

When using g the overall slowdown factor compared to the standard iteration function is at least $\frac{r+4}{r}$, with E called once every $(\frac{r}{2})^r$ steps on average.

Cycle detection, escape, and recurrence. Despite these countermeasures short cycles still occur, and longer cycles occur as well, so surviving cycles need to be dealt with. A common method, from [26], is to occasionally record a fixed length sequence of points, to compare the next point to all recorded ones, and to escape from a detected cycle by adding to a well-defined escape point $\mathfrak{p}$ on the cycle either $\mathfrak{f}_{l(\mathfrak{p})+c}$ for a small $c \in \mathbf{Z}_{>1}$, or a fixed other addition constant $\mathfrak{f}'$, or $\mathfrak{f}''_{l(\mathfrak{p})}$ from a set of r other addition constants.

However, after detecting and escaping from a cycle, a walk may recur to the same cycle, as was first described in [13]. Table 2 (slightly extending [13, Table 2]) lists lower bounds for the probability that such *recurring cycles* occur, along with various other relevant probabilities and data.

Using the negation map, or not. When the parameters were set for our initial rough Cell processor implementation of Pollard's rho method, recurring cycles and their occurrence probabilities as listed in Table 2 were still unknown. All we had observed was that tests combining the negation map with a variety of cycle reduction, detection, and escape attempts, and using r-values that were acceptable to us, resulted in unacceptably low long term distinguished point yields – as we later found out due to abundant recurring fruitless cycles. Here it should be pointed out that the little memory available was used to process simultaneously as many walks as possible, in order to better amortize the high cost of the original modular inversion [29]. This implied that we only looked at relative small r-values and settled for $r = 16$, a reasonable choice given the findings in [50]. Thus, based on our early findings and still unaware of various relevant issues, we chose not to use the negation map to run our PS3 cluster experiments with Pollard's rho method for the 112-bit prime field ECDLP.

By the time the inversion cost was reduced (by an order of magnitude, using the improved modular inversion presented in Section 4.5) and we decided to continue and to try to solve the ECDLP, it was not expected to be profitable to change the original r-value. Another argument was that methods to deal with cycles require more complicated code than the branch-free step function that is perfectly suited to the Cell processor's SIMD environment, and that applies if the negation map is not used.

Table 2, which we derived later while trying to better understand fruitless cycles, shows that large r-values greatly reduce the probability of mishaps. According to [6], using the negation map with a large r-value leads to the expected factor of $\sqrt{2}$ speedup on the cache-less cores of the Cell processor (Section 4.1), while not using cycle reduction but just occasional cycle detection and escape, and a clever choice of addition constants.

Also on processors *with* caches the negation map can profitably be used. Although large r-values are ruled out due to cache misses that lead to severe slowdowns [13, Fig. 1], and any r that is small enough to be cache-friendly unavoidably causes cycles (Table 2), they can effectively be dealt with using an occasional doubling. This is based on a heuristic argument from [13] that a cycle containing at least one doubling is most likely not fruitless. Thus, to avoid the problem of recurring cycles, it suffices to define the successor of the cycle escape point $\mathfrak{p}$ as $\sim(2\mathfrak{p})$. Furthermore, if the probabilities to enter 2- or 4-cycles in the last two columns of Table 2 are too high given the infrequency of cycle detection, $f(\mathfrak{p})$ and $g(\mathfrak{p})$ can be replaced by the doubling based variants

$$\bar{f}(\mathfrak{p}) = \begin{cases} \sim(\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})}) & \text{if } l(\mathfrak{p}) \neq l(\sim(\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})})), \\ \sim(2\mathfrak{p}) & \text{otherwise} \end{cases}$$

and

$$\bar{g}(\mathfrak{p}) = \begin{cases} \mathfrak{q} =\sim(\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})}) & \begin{cases} \text{if } l(\mathfrak{q}) \neq l(\mathfrak{p}) \neq l(\sim(\mathfrak{q} + \mathfrak{f}_{l(\mathfrak{q})})) \\ \text{and } l(\sim(\mathfrak{q} + \mathfrak{f}_{l(\mathfrak{q})})) \neq l(\mathfrak{q}), \end{cases} \\ \sim(2\mathfrak{p}) & \text{otherwise,} \end{cases}$$

respectively [13], thereby (heuristically) avoiding short fruitless cycles.

It is concluded in [13] that the negation map indeed leads to a speedup on regular servers and desktop PCs, assuming the proper modifications to the definition of the walk. These modifications, may not be compatible with the restrictions imposed by platforms where multiple walks are executed in SIMD fashion. If doubling and regular addition are different operations, the required occasional doublings cause SIMD threads to wait for each other, or they require costly thread-rearrangement (as mentioned earlier). When using the Cell processor and a small r-value, we have not been able to find affine variants or curve parametrizations allowing the same operation for addition and doubling (such as [24]) that are competitive – taking all issues into account such as negation map speedup and cycle overheads – with affine Weierstrass coordinates without the negation map.

3.4 Tag-tracing

Introduced in [19] to speed up r-adding walks, the idea of *tag-tracing* is that, given the low probability to hit a distinguished point, for most iterations a partial computation suffices. Given $\mathfrak{p}$ with $l(\mathfrak{p}) = i$ there is no need to fully calculate the next point $\mathfrak{q} = \mathfrak{p} + \mathfrak{f}_i$, unless it is a distinguished point, as long as there is enough information to compute $k = l(\mathfrak{q})$ in order to calculate $\mathfrak{q}$'s successor $\mathfrak{q} + \mathfrak{f}_k$. If a table containing the points $\mathfrak{f}_{ik} = \mathfrak{f}_i + \mathfrak{f}_k$ has

Table 2 Summary of effect of cycle reduction, detection, and escape methods. With the exception of the two bold entries, all figures are lower bounds.

	Successor of $\mathfrak{p}$:	$\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})}$	$f(\mathfrak{p})$	$g(\mathfrak{p})$
	Corresponding cycle reduction method:	none	2-cycle	4-cycle
Probability to enter	2-cycle	$\mathbf{\frac{1}{2r}}$	$\frac{1}{2r^3}$	$\frac{2(r-2)^2}{(r-1)r^4}$
	4-cycle	$\mathbf{\frac{r-1}{4r^3}}$	$\frac{r-1}{4r^3}$	$\frac{4(r-2)^4(r-1)}{r^{11}}$
	2ω-cycle for $\omega \in \mathbf{Z}_{>2}$, see [23]	$\Omega(r^{-\omega})$	$\Omega(r^{-\omega})$	$\Omega(r^{-\omega})$
Probability to recur to a cycle after escaping it from $\mathfrak{p}$ to	$\sim(\mathfrak{p} + \mathfrak{f}_{l(\mathfrak{p})+c})$	$\frac{1}{2r}$	$\frac{1}{2r^2}$	$\frac{(r-2)^2}{r^4}$
	$\sim(\mathfrak{p} + \mathfrak{f}')$	$\frac{1}{8r}$	$\frac{1}{8r^3}$	$\frac{(r-2)^2}{2r^5}$
	$\sim(\mathfrak{p} + \mathfrak{f}''_{l(\mathfrak{p})})$	$\frac{1}{8r^2}$	$\frac{1}{8r^4}$	$\frac{(r-2)^2}{2r^6}$
Slowdown factor of iteration function		n/a	$\frac{r+1}{r}$	$\frac{r+4}{r}$

been precomputed, it would then suffice to fully compute $\mathfrak{q}$'s successor as $\mathfrak{p} + \mathfrak{f}_{ik}$. Or, better, by taking the largest τ that allows storage of the table containing the

$$\sum_{k=1}^{\tau} \binom{r+k-1}{k} = \binom{r+\tau}{\tau} - 1$$

sums over at most τ elements from $\{\mathfrak{f}_i : 0 \leq i < r\}$, the same observation applies to $\mathfrak{p}$'s partially calculated first $\tau - 1$ successors, only fully calculating again its τth successor. The first partially calculated intermediate point that could be a distinguished point is fully calculated.

For discrete logarithms in multiplicative groups of finite fields, the group operation is modular multiplication. The partial calculation given in [19] suffices to recognize properly defined distinguished points and partition properties and leads to a tenfold speedup for 1024-bit prime fields. Generalization to ECDLP was left open.

ECDLP tag-tracing. The more complicated group operation in $\langle \mathfrak{g} \rangle$ makes it harder to apply the same idea to ECDLP. If only the x-coordinate is used for distinguishing and partition properties, calculation of the y-coordinate can be avoided, reducing the average cost per step by $\frac{\tau-1}{\tau}(2\mathtt{A} + \mathtt{M})$. Combined with simultaneous inversion, this leads to a speedup by a factor of approximately $\frac{6}{5}$ at best (i.e., for large τ), but this comes at various disadvantages that, depending on the circumstances, may invalidate the speedup entirely.

Although initialization cost of the table can be ignored, the cost of retrieving its entries will grow with τ due to memory access latencies. In practice this implies that τ will be of moderate size, thereby lowering the computational speedup that would ideally be achievable. Slight improvements can be obtained by not storing rarely accessed entries (taking an infrequently occurring more costly step instead): for instance, the table entry corresponding to $\mathfrak{f}_0 + \mathfrak{f}_1 + \mathfrak{f}_2$ will be accessed six times as often as the one for $3\mathfrak{f}_0$.

ECDLP tag-tracing is incompatible with the negation map, because the latter needs the y-coordinate that may not be computed while tag-tracing. One may conclude that usage of tag-tracing in most circumstances leads to a slow-down by a factor of $\frac{5}{6}\sqrt{2}$: only if r must be small (caches or very little memory) and occasional doubling is best avoided (SIMD) is it conceivable that the negation map is ineffective and that ECDLP tag-tracing (with small τ) gives a small speedup. We could have, but did not attempt to use ECDLP tag-tracing.

4 Pollard's rho method on the PS3

To solve the ECDLP from Section 2.1 with Pollard's rho method, each core of the Cell processes four walks simultaneously in 4-way SIMD fashion, two of those SIMD-processes are interleaved, and as many as possible of these interleaved processes are batched, to amortize the inversion cost in the best possible way (Section 3.1). Although the description below focuses on the 4-way SIMD parallelization of the Cell processor's cores, many ideas apply to wider SIMD environments as well, such as graphics cards.

Section 4.1 summarizes the design of the Cell processor [27]. The 4-way SIMD long integer representation, tailored to the Cell's instructions (described in Section 4.2), is presented in Section 4.3. The interleaved 4-way SIMD arithmetic modulo the specific p (Section 2.1) is described in Section 4.4. To gain speed, results may be incorrect; it is argued that it may be expected that bad cases do not occur (though an example is given). Section 4.5 describes a 4-way SIMD implementation of binary modular inversion. Timings and the solution to the ECDLP are given in Section 4.6. Section 4.7 gives estimates for ECDLP over other finite fields.

4.1 The Cell processor

The Cell processor, the main processor of the PS3 and thus mainly targeted at the gaming market, is a powerful general purpose processor. On the first generation PS3s it can be accessed using Sony's hypervisor, making the PS3 a relatively inexpensive source of processing power.

The architecture of the Cell processor is quite different from that of regular server or desktop processors. The Cell has a *Power Processing Element* (PPE), a dual-threaded Power architecture-based 64-bit processor with

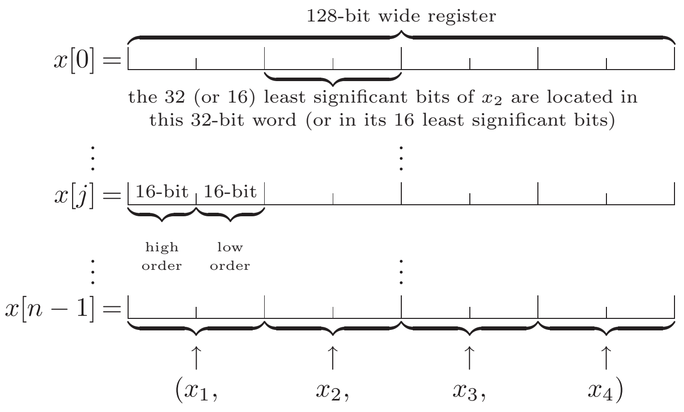
128-bit wide register
$x[0] =$
the 32 (or 16) least significant bits of x_2 are located in this 32-bit word (or in its 16 least significant bits)
$x[j] =$
16-bit
16-bit
high order
low order
$x[n-1] =$
$(x_1,$
$x_2,$
$x_3,$
$x_4)$

Figure 1 A four-tuple (x_1, x_2, x_3, x_4) of $32n$-bit or $16n$-bit integers represented by 128-bit registers $x[0], x[1], \ldots, x[n-1]$.

access to a 128-bit AltiVec/VMX SIMD unit. Its main processing power, however, comes from eight *Synergistic Processing Units* (SPUs). When running Linux, six SPUs can be used: one is disabled, and one is reserved by the hypervisor. It is conceivable that this last one becomes accessible too [28].

Each SPU runs independently from the others at 3.2GHz, using its own 256 kilobyte of fast local memory (the *Local Store*) for instructions and data. It is cache-less and has 128 registers of 128 bits each, allowing SIMD operations on sixteen 8-bit, eight 16-bit, or four 32-bit integers. An SPU has no $32 \times 32 \to 64$-bit or $64 \times 64 \to 128$-bit integer multiplier, but has several 4-way SIMD $16 \times 16 \to 32$-bit integer multipliers including three multiply-and-add instructions. They are described in Section 4.2 along with the other instructions used in the later sections. There is an odd and an even pipeline: per clock cycle an SPU can dispatch one odd and one even instruction. Because the SPU lacks smart branch prediction, branching is best avoided (as usual in SIMD).

4.2 Integer and bit arithmetic on the SPU

For a positive integer k, an *(unsigned) k-bit integer* is a non-negative integer less than 2^k, and a *signed k-bit integer* is an integer z satisfying $-2^{k-1} \leq z < 2^{k-1}$.

We interpret each 128-bit SPU register v as a four-tuple of 32-bit values (v_1, v_2, v_3, v_4), where v_i is the ith *word* of v which may be interpreted as signed or unsigned 32-bit integer. Below, a, b, c, d are 128-bit registers and all operations are for $i = 1, 2, 3, 4$ simultaneously.

The call $d = \texttt{spu\_add}(a, b)$ does 4-way SIMD 32-bit integer addition, calculating $d_i = (a_i + b_i) \bmod 2^{32}$. Other instructions generate the corresponding carries ($c = \texttt{spu\_genc}(a, b)$: $c_i = \lfloor (a_i + b_i)/2^{32} \rfloor$), include existing carries in additions ($d = \texttt{spu\_addx}(a, b, c)$: $d_i = (a_i + b_i + c_i) \bmod 2^{32}$), or generate the carries of the latter additions ($e = \texttt{spu\_gencx}(a, b, c)$: $e_i = \lfloor (a_i + b_i + c_i)/2^{32} \rfloor$). The corresponding integer subtraction instructions are `spu_sub`, `spu_genb`, `spu_subx`, and `spu_genbx`, where the '`b`' in '`genb`' indicates *borrow*: no borrow occurs if the borrow-bit is set to 1 (one), and a borrow occurs if the borrow-bit is set to 0 (zero). These are all even pipeline instructions that take two cycles.

The call $c = \texttt{spu\_mulo}(a, b)$ does 4-way SIMD $16 \times 16 \to 32$-bit unsigned integer multiplication, calculating $c_i = (a_i \bmod 2^{16}) \cdot (b_i \bmod 2^{16})$. There are two signed and one unsigned 4-way SIMD $(16 \times 16) + 32 \to 32$-bit multiply-and-add instructions. One of the signed ones calculates $c_i = (a_i \cdot b_i + d_i) \bmod 2^{32}$, where a_i and b_i are interpreted as signed 16-bit integers (i.e., their 16 most significant bits are ignored), and d_i and c_i are signed 32-bit integers.

The other two multiply-and-add instructions (the other signed one, and the unsigned one) work instead on the 16 most significant bits of a_i and b_i, ignoring the $2 \times 4 \times 16$ least significant bits. The unsigned instruction is used for modular multiplication: the call $c = \texttt{spu\_mhhadd}(a, b, d)$ calculates $c_i = (\lfloor a_i/2^{16} \rfloor \cdot \lfloor b_i/2^{16} \rfloor + d_i) \bmod 2^{32}$, where d_i and c_i are unsigned 32-bit integers. All these multiplications are even pipeline, one of them can be dispatched per cycle, taking seven cycles.

The call $c = \texttt{spu\_and}(a, b)$ calculates the 128-bit value $a \wedge b$, i.e., the bitwise and of its inputs. The word-wise comparison call $c = \texttt{spu\_cmpeq}(a, b)$ results in $c_i = 2^{32} - 1$ (i.e., all one bits across c's ith word) if a_i and b_i have the same value and $c_i = 0$ (i.e., all zero bits) otherwise. Both are even pipeline with a two cycle latency.

The or-across instruction call $\texttt{spu\_orx}(a)$ returns the 32-bit value $a_1 \vee a_2 \vee a_3 \vee a_4$, i.e., the bitwise inclusive or across the words of a. Using $d = \texttt{spu\_shuffle}(a, b, c)$ any 16 entries of a 32-byte table (a and b) can be looked up simultaneously: the pattern in c shuffles 16 of the 32 bytes of a and b to the output d, in such a way that the jth byte of c determines the jth byte of d, as a copy of a byte of a or b or as one of the constants {`0x00, 0xFF, 0x80`}. It allows duplicate copies. Both are odd four cycle latency instructions.

The positioning of bits in the top-half-words as in `spu_mhhadd` requires byte-rearranging shifts and shuffles. These are odd pipeline instructions that can be dispatched almost for free if they are interleaved with the even pipeline arithmetic ones. The split instruction call $(b, c) = \texttt{spu\_split}(a)$ re-arranges bytes: $b_i =$

$\lfloor a_i/2^{16} \rfloor$ and $c_i = a_i \bmod 2^{16} \in \{0, 1, 2, \dots, 2^{16}-1\}$, i.e., b_i gets a_i's top-half shifted right over two bytes and c_i gets a_i's bottom-half. This can be implemented in a variety of ways using a combination of two SPU instructions: using two even pipeline instructions, or two odd ones, or one of each. The opposite effect is achieved by the merge instruction call $a = \texttt{spu\_merge}(b, c)$: $a_i = 2^{16}b_i + c_i$, implemented using a single shuffle instruction. For $0 \le k < 32$, the shift instruction call $b = \texttt{spu\_sl}(a, k)$ left-shifts a_i over k bits: $b_i = a_i 2^k \bmod 2^{32} \in \{0, 1, 2, \dots, 2^{32}-1\}$.

4.3 Representation of long integers on the SPU

To solve the ECDLP from Section 2.1, integer arithmetic is required modulo the 112-bit number $p = \frac{2^{128}-3}{11 \cdot 6949}$. With 128-bit registers, a register could represent a 112-bit number modulo p. But that simple-minded approach is not easily compatible with the SPU's instruction set.

For applications that allow high degrees of parallelization, such as Pollard's rho method, a 90-degree interpretative turn of the words is a better fit for the SPU's instruction set: instead of representing a 112-bit integer using a single 128-bit register, a **four-tuple** of long integers (such as 112-bit ones) is laid out across the four-tuples of words of a sequence of 128-bit registers, thereby allowing the corresponding words of the four long integers, i.e., the words that belong to the same 128-bit register, to be processed simultaneously in SIMD fashion. Fig. 1 illustrates two ways to map four-tuples of long integers to a sequence of 128-bit registers: one that uses all $4 \times 32 = 128$ bits of each register, and one where only $4 \times 16 = 64$ of the 128 bits per register are significant. This approach allows 4-way SIMD processing of four-tuples of identically sized long integers of any size.

Both methods represent four-tuples of long integers by *word slicing* a number of 128-bit registers. The choice of representation depends on the operation to be carried out, as further discussed in the next section. As above, each 128-bit register v is interpreted as a four-tuple (v_1, v_2, v_3, v_4) of 32-bit words. Here these words are interpreted as unsigned 32-bit integers.

In the first representation method, a sequence of ℓ 128-bit registers $x[0], x[1], \dots, x[\ell-1]$ is used to represent a four-tuple (x_1, x_2, x_3, x_4) of 32ℓ-bit integers in their radix 2^{32} representation:

$$x_i = \sum_{j=0}^{\ell-1} x[j]_i 2^{32j}$$

for $i = 1, 2, 3, 4$. Thus, the ith word $x[j]_i$ of the 128-bit register $x[j]$ equals the coefficient of 2^{32j} in the radix 2^{32} representation of the ith 32ℓ-bit integer x_i, for $j = 0, 1, \dots, \ell-1$ and $i = 1, 2, 3, 4$. This representation matches the SPU's 4-way SIMD integer additions and subtractions from Section 4.2, and is used for long integer addition, subtraction, and modular inversion.

In the second representation method, a sequence of m 128-bit registers $y[0], y[1], \dots, y[m-1]$ is used to represent a four-tuple (y_1, y_2, y_3, y_4) of $16m$-bit integers in their radix 2^{16} representation:

$$y_i = \sum_{j=0}^{m-1} (y[j]_i \bmod 2^{16}) 2^{16j}$$

for $i = 1, 2, 3, 4$ and where $0 \le y[j]_i \bmod 2^{16} < 2^{16}$. Thus, the two least significant bytes of the ith word $y[j]_i$ of the 128-bit register $y[j]$ contain the coefficient of 2^{16j} in the radix 2^{16} representation of the ith $16m$-bit integer y_i, for $j = 0, 1, \dots, m-1$ and $i = 1, 2, 3, 4$. When used with the shift instruction `spu_sl`, this representation matches the SPU's 4-way SIMD unsigned multiply-and-add instruction `spu_mhhadd` (Section 4.2). It is used for long integer multiplication.

Thus we use the 128-bit register width to hard-code 4-way SIMD processing of four-tuples of long integers. The values for ℓ (full-word radix 2^{32}) and m (bottom-half-word radix 2^{16}) depend on the modulus size. For 112-bit or 128-bit moduli (as both used below), $\ell = 4$ and $m = 8$ are used, $\ell = 5$ for moduli of 129 up to 160 bits, and $m = 10$ for moduli of 145 up to 160 bits.

4.4 4-way SIMD long integer SPU-arithmetic

With $R = 2^{128}$, reduction modulo the multiple $\tilde{p} = R - 3$ of the prime $p = \frac{\tilde{p}}{11 \cdot 6949}$ (Section 2.1) can be done using *sloppy reduction modulo* $\tilde{p}$, which is faster than reduction modulo p but which may produce an incorrect result, with a probability that is argued to be negligible. When working in $\mathbf{F}_p$ we use a redundant representation modulo $\tilde{p}$. Only when required for distinguishing and partition properties (Section 3.1) we switch to a unique value modulo p using a quick Montgomery-like step [41]. All methods in this section allow any number of SIMD threads. See [3, 4, 5, 22, 47, 48], for instance, for previous work involving primes of a special form. We are not aware of earlier publication of modular arithmetic similar to sloppy reduction or an analysis thereof.

Sloppy reduction modulo $\tilde{p}$. For $z \in \mathbf{Z}$ with $0 \le z < R^2$ and $z = z_0 + R z_1$ for $z_0, z_1 \in \mathbf{Z}, 0 \le z_0, z_1 < R$, define

$$\mathfrak{R}(z) = z_0 + 3z_1.$$

From $\tilde{p} = R - 3$ it follows that $\mathfrak{R}(z) \equiv z \bmod \tilde{p}$ and $\mathfrak{R}(z) \equiv z \bmod p$. With $\mathfrak{R}(z) = y = y_0 + y_1 R$ for $y_0, y_1 \in \mathbf{Z}, 0 \le y_0, y_1 < R$, it follows from $\mathfrak{R}(z) = z_0 + 3z_1 \le 4R - 4$ that $y_1 \le 3$. If $y_1 = 3$, then $y_0 + y_1 R = y_0 + 3R \le 4R - 4$ and thus $y_0 \le R - 4$. Using $y_0 \le R - 1$ when $y_1 \le 2$, it follows that $\mathfrak{R}(y) = y_0 + 3y_1 \le R + 5$.

Define $\mathfrak{S}(z) = \mathfrak{R}(\mathfrak{R}(z))$. Then $\mathfrak{S}(z) < R + 6$ and $\mathfrak{S}(z) \equiv z \bmod \tilde{p}$ (and thus $\mathfrak{S}(z) \equiv z \bmod p$). If all values in the range of $\mathfrak{S}$ occur with approximately the same probability, then $\mathfrak{S}(z) \ge R$ with probability close to $\frac{6}{R+6}$, which is small. Thus, the truncated value $\mathfrak{S}(z) \bmod R \in \{0, 1, \dots, R-1\}$ is most likely equivalent to z modulo $\tilde{p}$. For relevant z-values, i.e, products of two 128-bit integers, it is argued below that $\mathfrak{S}(z) \ge R$

Algorithm 1 Sloppy reduction modulo $\tilde{p}$ of a four-tuple of 256-bit integers.

```
Input: { a four-tuple $(c_1, c_2, c_3, c_4)$ of 256-bit integers in radix $2^{16}$
       { represented by sixteen 128-bit registers $c[0], c[1], \ldots, c[15]$.
Output: { a four-tuple $(t_1, t_2, t_3, t_4)$ of 128-bit integers $t_i = \mathfrak{S}(c_i) \bmod R$, for $i = 1, 2, 3, 4$, in radix $2^{16}$
        { represented by eight 128-bit registers $t[0], t[1], \ldots, t[7]$.
 1: Let $r$ be a register with $r_1 = r_2 = r_3 = r_4 = 3 \cdot 2^{16}$
 2: /* the 16 most significant bits of the words of $r$ all represent 3 */
 3: /* Compute the first application of $\mathfrak{R}$ */
 4: for $k = 0$ to 7 do
 5:     $t[k] =$ spu_mhhadd(spu_sl($c[k+8]$, 16), $r$, $c[k]$)
 6: for $k = 0$ to 6 do
 7:     $(s, t[k]) =$ spu_split($t[k]$)
 8:     $t[k+1] =$ spu_add($t[k+1]$, $s$)
 9: $(s, t[7]) =$ spu_split($t[7]$)
10: /* Compute the second application of $\mathfrak{R}$ */
11: $t[0] =$ spu_mhhadd(spu_sl($s$, 16), $r$, $t[0]$)
12: $(s, t[0]) =$ spu_split($t[0]$)
13: if spu_orx($s$) $\neq 0$ then
14:     $t[1] =$ spu_add($t[1]$, $s$)
15:     for $k = 1$ to 6 do
16:         $(s, t[k]) =$ spu_split($t[k]$)
17:         $t[k+1] =$ spu_add($t[k+1]$, $s$)
18: /* truncate modulo $R$ by ignoring that there may be an $i \in \{1, 2, 3, 4\}$ with $t[7]_i \geq 2^{16}$ */
19: return $t[0], t[1], \ldots, t[7]$
```

with probability only about $\frac{1}{R}$, so low that $\mathfrak{S}$ may indeed simply be truncated, rather than applying $\mathfrak{R}$ a third time (which would always be correct modulo $\tilde{p}$ and p). Sloppy reduction modulo $\tilde{p}$ of z is therefore defined as $\mathfrak{S}(z) \bmod R \in \{0, 1, \ldots, R-1\}$.

The SPU calculation of 4-way SIMD sloppy reduction modulo $\tilde{p}$ of a four-tuple of 256-bit integers in radix 2^{16} representation is done by the algorithm depicted in Alg. 1. Without the `if`-statement in line 13 (while keeping lines 14-17) it is branch-free (and slower).

Incorrectness probability of sloppy reduction modulo $\tilde{p}$ of products. Let $0 \leq x, y < R$ and let $xy = a + b\tilde{p}$ for integers a, b with $0 \leq a < \tilde{p}$ and $0 \leq b \leq R + 1$. Define c as the smallest integer such that $0 \leq cR + a - 3b < R$. It then follows from $xy = cR + a - 3b + (b - c)R$ that $\mathfrak{R}(xy) = cR + a - 3c$. If $a - 3c \geq 0$, then $\mathfrak{S}(xy) = a < \tilde{p}$ so that sloppy reduction modulo $\tilde{p}$ produces the correct result. If $a - 3c < 0$, then $\mathfrak{R}(xy) = (c-1)R + R + a - 3c$. With $cR < R - a + 3b \leq R + 3(R+1)$ so that $c \leq 4$ and thus $0 \leq R + a - 3c < R$, it follows that $\mathfrak{S}(xy) = R + a - 3c + 3c - 3 = R + a - 3$. Because also $\mathfrak{S}(xy) < R + 6$, the cases where $\mathfrak{S}(xy) \geq R$ (and sloppy reduction modulo $\tilde{p}$ is incorrect) are $3 \leq a \leq 8$.

Because $\mathfrak{S}(xy) \in \{R, R+1, \ldots, R+5\}$ for pairs (x, y) for which sloppy reduction modulo $\tilde{p}$ of xy is incorrect, it follows that $\mathfrak{S}(xy)$ is coprime to $\tilde{p}$, implying that x and y are co-prime to $\tilde{p}$. But if for such a pair it is the case that $\gcd(x, \tilde{p}) = 1$, then $\gcd(y, \tilde{p}) = 1$ as well.

Writing $a = i + 3k$, where $i \in \{0, 1, 2\}$ and $k \in \{1, 2\}$, it follows from $a - 3c < 0$ that $c \geq k + 1$. Since c is minimal such that $0 \leq cR + a - 3b < R$, it follows that $kR + a - 3b < 0$ and thus $b > \frac{a + kR}{3}$. With $xy = a + b\tilde{p}$ and $a \geq 3k$ this implies $xy > 3k + \frac{(3k + kR)\tilde{p}}{3} = 3k + \frac{k(R+3)(R-3)}{3} = \frac{kR^2}{3}$. Thus $x, y > \frac{kR}{3}$, since $0 \leq x, y < R$. The number of pairs (x, y) with $x, y > \frac{kR}{3}$ and $xy > \frac{kR^2}{3}$ is approximated as

$$\frac{(3-k)R^2}{3} - \int_{\frac{kR}{3}}^{R} \frac{kR^2}{3x} dx = \frac{(3-k)R^2}{3} - \frac{kR^2}{3} \log\left(\frac{3}{k}\right).$$

For $3k \leq a < 3(k+1)$ the probability that $xy \equiv a \bmod \tilde{p}$ for a pair (x, y) may be approximated as $\frac{3}{R} \cdot \frac{\phi(\tilde{p})}{\tilde{p}}$ (where ϕ denotes Euler's totient function). This leads to

$$\left(\frac{\phi(\tilde{p})}{\tilde{p}}\right) \cdot R \cdot \sum_{k=1,2} \left(3 - k - k \log\left(\frac{3}{k}\right)\right)$$

as a heuristic approximation for the total number of pairs (x, y) where sloppy reduction modulo $\tilde{p}$ of the product xy produces an incorrect result. Because $\frac{\phi(\tilde{p})}{\tilde{p}} \approx 0.90896$, the sum equals $3 - \log\left(\frac{27}{4}\right) \approx 1.09046$, and $0.90896 \cdot 1.09046 \approx 0.99118$, we find a heuristic upper bound of $\frac{1}{R}$ for the probability that sloppy reduction modulo $\tilde{p}$ of xy is incorrect, assuming that x and y are drawn at random.

Incorrectness probability for other moduli. Sloppy reduction may be advantageous for other primes of the form $\frac{2^{32\ell} \pm m}{c}$ for relatively small $\ell, m, c \in \mathbf{Z}_{>0}$. For $\ell = 6, 8$, $m = 38$, $c = 2$ [5, 9], and the functions $\mathfrak{R}'(z_0 + z_1 2^{32\ell}) = z_0 + mz_1$ and $\mathfrak{S}'(z) = \mathfrak{R}'(\mathfrak{R}'(z))$, sloppy reduction modulo either of the two primes $2^{32\ell-1} - \frac{m}{2}$ is defined as $\mathfrak{S}' \bmod 2^{32\ell}$, i.e., truncation of $\mathfrak{S}'$ to 32ℓ bits (this works for $\ell = \frac{1}{2}$ and $\ell = 1$ too). A heuristic upper bound of $\frac{343}{2^{32\ell}}$ for the probability that sloppy reduction

Algorithm 2 Radix 2^{16} schoolbook multiplication of two four-tuples of $16m$-bit integers.

```
Input:  { two four-tuples $(a_1, a_2, a_3, a_4), (b_1, b_2, b_3, b_4)$ of $16m$-bit integers in radix $2^{16}$
        { represented by $2m$ 128-bit registers $a[0], a[1], \ldots, a[m-1], b[0], b[1], \ldots, b[m-1]$.
Output: { a four-tuple $(c_1, c_2, c_3, c_4)$ of $32m$-bit integers $c_i = a_i \cdot b_i$, for $i = 1, 2, 3, 4$, in radix $2^{16}$
        { represented by $2m$ 128-bit registers $c[0], c[1], \ldots, c[2m-1]$.
 1: for $k = 0$ to $m - 1$ do
 2:     $c[m + k] = 0$
 3:     $a[k] = \texttt{spu\_sl}(a[k], 16)$
 4:     $b[k] = \texttt{spu\_sl}(b[k], 16)$
 5: for $j = 0$ to $m - 1$ do
 6:     $(e[0], c[j]) = \texttt{spu\_split}(\texttt{spu\_mhhadd}(a[0], b[j], c[m]))$
 7:     for $k = 1$ to $m - 1$ do
 8:         $(e[k], c[m + k - 1]) = \texttt{spu\_split}(\texttt{spu\_add}(\texttt{spu\_mhhadd}(a[k], b[j], c[m + k]), e[k - 1]))$
 9:         /* $a[k]_i \cdot b[j]_i + c[m + k]_i + e[k - 1]_i \le (2^{16} - 1)^2 + 2^{16} - 1 + 2^{16} - 1 = 2^{32} - 1$ for $i = 1, 2, 3, 4$ */
10:     $c[2m - 1] = e[m - 1]$
11: return $c[0], c[1], \ldots, c[2m - 1]$
```

Algorithm 3 Division by 2^{16} modulo p of a four-tuple of 128-bit integers.

```
Input:  { a four-tuple $(x_1, x_2, x_3, x_4)$ of 128-bit integers in radix $2^{16}$
        { represented by eight 128-bit registers $x[0], x[1], \ldots, x[7]$.
Output: { a four-tuple $(y_1, y_2, y_3, y_4)$ of 128-bit integers $y_i \equiv x_i 2^{-16} \bmod p$, for $i = 1, 2, 3, 4$, in radix $2^{16}$
        { represented by eight 128-bit registers $y[0], y[1], \ldots, y[7]$.
 1: Let $p[0], p[1], \ldots, p[6]$ be 128-bit registers representing $p_1 = p_2 = p_3 = p_4 = p$ in radix $2^{16}$
 2: /* Put $p$'s bits in the 16 most significant locations, if that has not been done beforehand */
 3: for $k = 0$ to 6 do
 4:     $p[k] = \texttt{spu\_sl}(p[k], 16)$
 5: $\nu = \texttt{spu\_sl}(\texttt{spu\_mulo}(x[0], r), 16)$ where $r$ is a register with $r_1 = r_2 = r_3 = r_4 = 47325$
 6: $(y[0], d) = \texttt{spu\_split}(\texttt{spu\_mhhadd}(p[0], \nu, x[0]))$ /* $d$ is zero */
 7: for $k = 1$ to 6 do
 8:     $(y[k], y[k - 1]) = \texttt{spu\_split}(\texttt{spu\_add}(\texttt{spu\_mhhadd}(p[k], \nu, y[k - 1]), x[k]))$
 9: $(y[7], y[6]) = \texttt{spu\_split}(\texttt{spu\_add}(x[7], y[6]))$
10: return $y[0], y[1], \ldots, y[7]$
```

modulo $2^{32\ell} - m$ of xy is incorrect, for random non-negative $x, y < 2^{32\ell}$, follows as above. It uses

$$\sum_{k=1}^{m-1} \left(m - k - k \log\left(\frac{m}{k}\right) \right) \approx 342.552$$

and an argument involving $c = 2$ that is somewhat more contrived than the $\phi(\tilde{p})$-argument above: for odd a both x and y must be odd and integration is over the odd x's only, for even a each odd x leads to a single even y whereas each even x leads to two y's. Thus, the summation hides the observation that $\frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} + \frac{1}{2} \cdot 2\right) = 1$.

Sloppy multiplication modulo $\tilde{p}$. Alg. 2 depicts the algorithm for the SPU calculation of 4-way SIMD schoolbook multiplication of two four-tuples of $16m$-bit integers in radix 2^{16} representation [9]. The only subtlety in Alg. 2 is that none of the two 4-way SIMD additions in line 8 (`spu_add` and as part of `spu_mhhadd`) generates a carry. Alg. 1 and Alg. 2 are compatible: using Alg. 2 with $m = 8$, its four-tuple output can be simultaneously reduced modulo $\tilde{p}$ using Alg. 1, and the latter's four-tuple output can again be used as one of the four-tuple inputs for Alg. 2. Sloppy multiplication modulo $\tilde{p}$ consists of a call to Alg. 2 with $m = 8$ followed by a call to Alg. 1.

All four outputs of Alg. 1 have a small probability not to be unique modulo $\tilde{p}$ (with only the residue classes 0, 1, and 2 modulo $\tilde{p}$ allowing two representations), but the outputs are not unique modulo p. Unique representations modulo p are obtained as indicated below. As analyzed above, each output has a small probability to be incorrect: for instance, when $2 \bmod \tilde{p}$ is represented as $2 + \tilde{p} = R - 1$ and squared, the value $\mathfrak{S}((R-1)^2) = R + 1$ is truncated to the incorrect result 1.

Unique representation modulo p. Given a four-tuple (x_1, x_2, x_3, x_4) of integers modulo $\tilde{p}$ in $\{0, 1, \ldots, R - 1\}$, a unique representation modulo p is required for each x_i at the end of each step of Pollard's rho method. Least non-negative remainders modulo p require computation of $x_i \bmod p \in \{0, 1, \ldots, p - 1\}$ for $i = 1, 2, 3, 4$. A faster way to obtain unique representations modulo p is to simultaneously calculate all $x_i 2^{-16} \bmod p \in \{0, 1, \ldots, p - 1\}$. This is not the same as $x_i \bmod p \in \{0, 1, \ldots, p - 1\}$, but that is not a problem as long as the distinguishing and partition properties are properly defined.

The computation of $x_i 2^{-16} \bmod p$ is done using a single Montgomery reduction [41] iteration in radix 2^{16}. Because $\frac{-1}{p} \equiv 47325 \bmod 2^{16}$, the value $\nu_i = \frac{-x_i}{p} \bmod 2^{16} = 47325 x_i \bmod 2^{16}$ satisfies $x_i + \nu_i p \equiv 0 \bmod 2^{16}$, so that $y_i = (x_i + \nu_i p)/2^{16} \equiv x_i 2^{-16} \bmod p$. A unique representation in $\{0, 1, \ldots, p-1\}$ of y_i modulo p is obtained by observing that

$$y_i \leq \frac{R - 1 + (2^{16} - 1)p}{2^{16}} < 3p,$$

so that either y_i, $y_i - p$, or $y_i - 2p$ is in $\{0, 1, \ldots, p-1\}$.

A 4-way SIMD algorithm to perform the calculation of (y_1, y_2, y_3, y_4) given a four-tuple (x_1, x_2, x_3, x_4) as above is depicted in Alg. 3 (which in practice should be replaced by a version that uses radix 2^{32} as opposed to radix 2^{16} for the additions to the x_i's). The unique representation is then obtained by two applications of the 4-way SIMD modular subtraction algorithm depicted in Alg. 4 with $\ell = 4$. Alg. 4 uses masks to avoid branching, and can simply be changed to have radix 2^{32} inputs or output. If it is used with $b_i = m_i = p$, then the resulting c_i equals the input a_i if $a_i < p$ but c_i equals $a_i - p$ if $a_i \geq p$, for $i = 1, 2, 3, 4$ simultaneously, as required.

Pipelining. To reduce bottlenecks in the even and the odd pipelines, the implementations of all algorithms presented here attempt to balance the two pipelines by shifting instructions between the two. Bottlenecks are also reduced by interleaving two 4-way SIMD processes, thereby considerably increasing overall throughput and reducing overall latency, sacrificing the (mostly irrelevant) latency per walk of Pollard's rho method.

Simultaneous inversion. With $r = 16$ as chosen in Section 3.1 it is possible to store the data for 50 sequential interleaved 4-way SIMD walks in the SPU's Local Store, synchronizing the walks at the point where the modular inverses are calculated. Per SPU we use the simultaneous inversion from Section 3.1 in a nested manner, not sharing inversions among multiple SPUs as the computational advantages would be outweighed by synchronization and communication overhead.

Let $z_{ijk} \in \mathbf{F}_p^*$ for $1 \leq k \leq 50$, $1 \leq j \leq 2$ and $1 \leq i \leq 4$ denote the 400 elements for which the inversions will be shared per SPU. Using 99 (partially interleaved) 4-way SIMD sloppy multiplications modulo $\tilde{p}$ the four-tuple $(\nu_1, \nu_2, \nu_3, \nu_4)$ of products $\nu_i = \prod_{k=1}^{50} \prod_{j=1,2} z_{ijk} \bmod \tilde{p}$ is calculated, for $i = 1, 2, 3, 4$ simultaneously, while keeping the partial products. The four inverses $\nu_i^{-1} \bmod p$ are then calculated using simultaneous inversion at the cost of $3 \times (4-1) = 9$ modular multiplications and one modular inversion (described in Section 4.5), possibly using a SIMD tree-based approach for the combination and unraveling. Finally, the individual inverses $z_{ijk}^{-1} \bmod p$ are calculated (in a representation modulo $\tilde{p}$) at the cost of twice 99 4-way SIMD sloppy multiplications modulo $\tilde{p}$, by unraveling in 4-way SIMD fashion.

4.5 SIMD modular inversion on the SPU

The calculation of the modular inverse of a positive integer b in a residue class of the odd modulus $a = p$ is outlined by the algorithm depicted in Alg. 5. It uses the binary version of the Euclidean algorithm from [31] to compute an almost Montgomery inverse $b^{-1} 2^k \bmod p$ for some integer k, because that allows fast implementation on the SPU. The factor $2^k \bmod p$ is removed by table look-up of the value $2^{-k} \bmod p$ (which equals $\frac{2^{1-k} \bmod p}{2}$ if $(2^{1-k} \bmod p) \in \{0, 1, 2, \ldots, p-1\}$ is even and $\frac{(2^{1-k} \bmod p) + p}{2}$ otherwise) followed by sloppy multiplication modulo $\tilde{p}$ from Section 4.4.

Let $d = \gcd(a, b)$. Let y be a solution of $by \equiv d \bmod a$. The algorithm has invariants

$$\begin{aligned} k_u, k_v &\geq 0, \quad u, v > 0, \\ u(2^{k_u + k_v} y) &\equiv rd \bmod a, \\ v(2^{k_u + k_v} y) &\equiv sd \bmod a, \\ \gcd(u, v) &= d, \\ us - vr &= a, \\ 2^{k_u} u \leq a, &\quad 2^{k_v} v \leq b, \\ r &\leq 0 < s. \end{aligned} \tag{3}$$

The values of u and v are bounded by a and b, respectively. The invariant $a = us - vr \geq s - r$ bounds r and s. For $\ell = 4$ both r and s fit in 128 bits. When the loop exits the subscript $k_u + k_v$ is bounded as follows:

$$2^{k_u + k_v} \leq (2^{k_u} u)(2^{k_v} v) \leq ab.$$

At that point $u = v = \gcd(u, v) = d$. If $v > 1$ then b is not coprime to a and the modular inverse computation fails. Otherwise $d = 1$ and the output $z = s \cdot (2^{-k_u - k_v})$ satisfies

$$\begin{aligned} z = zd &\equiv s \cdot (2^{-k_u - k_v}) d \\ &\equiv (v 2^{k_u + k_v} y) 2^{-k_u - k_v} \equiv vy = y \bmod a. \end{aligned}$$

At the start of every iteration at least one of u and v is odd, by (3). If t_u and t_v are picked as large as possible, then the new u and v will both be odd, so that after the subtraction and next iteration's shift $u + v$ will be reduced by at least a factor of 2.

The trailing zero bit count of a positive integer k is the population count of $\overline{k} \wedge (k - 1)$. Examining u and v simultaneously can therefore be done using the SPU's population count instruction; however, it acts only on 8-bit data, so the resulting t_u and t_v may not be maximal. This increases the number of iterations performed by Alg. 5 by about 1%: with maximal t_u and t_v the number of iterations would be close to 0.706 times the bitlength of a, as analyzed in [34]. Alg. 5 needs on average almost 80 iterations for inversion modulo p.

The four differences $u - v$, $r - s$, $v - u$, and $s - r$ are evaluated simultaneously. The loop is exited if neither $u - v$ nor $v - u$ needs a borrow. Otherwise, depending on the sign of $u - v$ a mask is created to build a fast

Algorithm 4 Modular subtraction of two four-tuples of 32ℓ-bit integers in radix 2^{16} representation.

```
Input:  a four-tuple $(m_1, m_2, m_3, m_4)$ of $32\ell$-bit integer moduli in radix $2^{32}$
        represented by $\ell$ 128-bit registers $m[0], m[1], \ldots, m[\ell-1]$
        (typically, but not necessarily, the four moduli are the same);
        two four-tuples $(a_1, a_2, a_3, a_4), (b_1, b_2, b_3, b_4)$ of $32\ell$-bit integers in radix $2^{16}$
        with $0 \leq a_i$ and $0 \leq b_i \leq m_i$ for $i = 1, 2, 3, 4$,
        represented by $4\ell$ 128-bit registers $a[0], a[1], \ldots, a[2\ell-1], b[0], b[1], \ldots, b[2\ell-1]$.
Output: a four-tuple $(c_1, c_2, c_3, c_4)$ of $32\ell$-bit integers in radix $2^{16}$ with $0 \leq c_i \equiv (a_i - b_i) \bmod m_i$
        for $i = 1, 2, 3, 4$, represented by $2\ell$ 128-bit registers $c[0], c[1], \ldots, c[2\ell-1]$ where,
        if $b_i = m_i = p$, then $c_i = a_i$ if $a_i < p$ but $c_i = a_i - p$ if $a_i \geq p$, for $i = 1, 2, 3, 4$.
1:  Let $\beta$ be a register with $\beta_1 = \beta_2 = \beta_3 = \beta_4 = 1$, for four borrows that are initially empty
2:  Let $\gamma$ be a register with $\gamma_1 = \gamma_2 = \gamma_3 = \gamma_4 = 0$, for four carries that are initially empty
3:  /* Convert $a$ and $b$ input registers to radix $2^{32}$ */
4:  for $k = 0$ to $\ell - 1$ do
5:      $u[k] =$ spu_merge($a[2k+1], a[2k]$)
6:      $v[k] =$ spu_merge($b[2k+1], b[2k]$)
7:  /* Do the subtraction */
8:  for $k = 0$ to $\ell - 1$ do
9:      $c[k] =$ spu_subx($u[k], v[k], \beta$)
10:     $\beta =$ spu_genbx($u[k], v[k], \beta$)
11: /* Set the masks for the negative $c_i$'s, i.e., the zero $\beta_i$'s */
12: $\mu =$ spu_cmpeq($\beta, 0$) /* where 0 consists of 128 zero bits */
13: /* if $\beta_i = 0$ (implying that $i$th mask $\mu_i$ is all ones), then add $m_i$ to $c_i$ */
14: for $k = 0$ to $\ell - 1$ do
15:     $\nu =$ spu_and($m_i, \mu$)
16:     $t[k] =$ spu_addx($c[k], \nu, \gamma$)
17:     $\gamma =$ spu_gencx($c[k], \nu, \gamma$)
18: /* Convert from radix $2^{32}$ to radix $2^{16}$ */
19: for $k = 0$ to $\ell - 1$ do
20:     $(c[2k+1], c[k]) =$ spu_split($t[k]$)
21: return $c[0], c[1], \ldots, c[2\ell-1]$
```

branch-free selector of the parts of (u, r, v, s) that must be updated. This, and the fact that we know that the inputs are co-prime, avoids the four branches from Alg. 5. The implementation does not take advantage of the decreasing sizes of u and v or of the initial small sizes of r and s, but treats them all as 32ℓ-bit integers. Nevertheless, it is quite efficient because only 4-way SIMD operations are carried out on the four-tuple (u, r, v, s). For $\ell = 4$ it is about 8.5 times faster than the implementation from [29].

4.6 Timings and solution of the ECDLP

With parameters as selected above, the clock cycle counts for the various operations are listed in Table 3. It lists both the number of clock cycles used by a single operation for eight walks in parallel (organized as two interleaved 4-way SIMD processes), but also the *artificial* number of clock cycles used per operation and iteration in the third and fifth column, respectively: artificial because a single sloppy multiplication modulo $\tilde{p}$ for one walk is not completed in 54 clock cycles, but $8 \times 54 \approx 430$ clock cycles suffice to do eight multiplications, one for each of eight walks.

Table 3 refers only to the cost of regular point addition, as iterations do not perform doublings: this saves code (and thus space) and makes the main inner-loop of the parallel walks branch-free at a negligible risk to drop off the curve (as argued in Section 3.1). The "Miscellaneous" category accounts for the retrieval of the $\mathfrak{f}_i$'s, data-shuffling, distinguished point checking, and all other overheads including occasional branching.

At 3.2GHz, an SPU performs about seven million iterations per second. With a 24-bit distinguishing property (of the unique representation of $x2^{-16} \bmod p \in \{0, 1, 2, \ldots, p-1\}$), a single PS3 (six SPUs) produced on average five distinguished points every two seconds, i.e., at most 160-bytes per second in uncompressed format. The ethernet connecting a server with the 215 PS3s could easily handle the required bandwidth.

Approximately 8.5×10^{16} elliptic curve additions were carried out to find that $m\mathfrak{g} = \mathfrak{h}$ for

$$m = 312521636014772477161767351856699.$$

This number of elliptic curve additions is close to the number $\sqrt{\frac{\pi q}{2}} \approx 8.36 \times 10^{16}$ of iterations expected based on the birthday paradox. It is also close to the number of iterations expected based on Eq. (1), namely $\sqrt{\frac{\pi q}{2(1-\frac{1}{16})}} \approx 8.64 \times 10^{16}$, which takes into account that we used a 16-adding walks. This effort translates into more than 10^{18} additions and multiplications modulo

Algorithm 5 Outline of a single modular inverse computation using 4-way SIMD arithmetic.

```
Input: { a, b, ℓ where a is odd, a, b > 0, and ℓ is the radix $2^{32}$ length of a;
       { assume availability of a large enough table of $2^{-k}$ mod a for $k = 0, 1, 2, 3, \ldots$ .
Output: "Not relatively prime," or a residue class $b^{-1}$ mod $a$.
1: Let $(u, r, v, s)$ be a four-tuple of $32\ell$-bit integers, represented in radix $2^{32}$ using $\ell$ 128-bit registers, with initial
   value $(a, 0, b, 1)$.
2: Let $(k_u, k_v)$ be a pair of 32-bit integers, represented using a 128-bit register, with initial value $(0, 0)$
3: while true do
4:    Find $t_u$ such that $2^{t_u}$ divides $u$ and $t_v$ such that $2^{t_v}$ divides $v$ (see text)
5:    Replace $(k_u, k_v)$ by $(k_u + t_u, k_v + t_v)$
6:    Use shifts to replace $(u, r, v, s)$ by $(u/2^{t_u}, r \cdot 2^{t_v}, v/2^{t_v}, s \cdot 2^{t_u})$
7:    if $u > v$ then
8:       Replace $(u, r, v, s)$ by $(u - v, r - s, v, s)$
9:    else if $v > u$ then
10:      Replace $(u, r, v, s)$ by $(u, r, v - u, s - r)$
11:   else if $v$ equals 1 then
12:      return $s \cdot 2^{-(k_u+k_v)} \bmod a$
13:   else
14:      return Not relatively prime
```

the 112-bit prime number p (or, most of the time, its 128-bit multiple $\tilde{p} = 2^{128} - 3$), and thus to well over 2^{60} operations on 32-bit or 64-bit integers. With our latest software the calculation would have taken less than four months. Because earlier versions were less efficient, the actual calculation took from January 13 to July 8, 2009. See also [6].

Slightly more than five billion distinguished points were collected. All distinguished points received were correct, indicating that none of the 5×10^{17} sloppy reductions modulo $\tilde{p}$ was incorrect (each had probability argued to be less than $2^{-128} \approx 10^{-38.53}$ to be incorrect, Section 4.4), and that none of the walks dropped off the curve due to an overlooked doubling (which too would have happened with negligible probability, Section 3.1) – or that if such mishaps occurred they magically cancelled each others' effect (a possibility that can safely be ruled out).

4.7 ECDLP over other finite fields

Based on the above, the performance for other prime field ECDLP instances can be estimated. For binary extension fields different methods apply, both for the underlying field arithmetic and for Pollard's rho method; for such fields the reader is referred to [2].

If the prime does not have a special form, sloppy reduction cannot be used. For generic, odd moduli Montgomery arithmetic [41] can be used, because it is perfectly amenable to SIMD-implementation. It replaces modular multiplication by Montgomery multiplication, which consist of regular multiplication followed by (or interleaved with) Montgomery reduction. As a result special purpose sloppy reduction is replaced by generic Montgomery reduction. The latter is cost-wise roughly equivalent to ordinary schoolbook multiplication plus some overhead. An additional modular subtraction suffices to obtain a unique representation, if so required.

The average clock cycle count for two interleaved 4-way SIMD Montgomery multiplications for generic odd moduli of up to 128 bits is about 620, while the cost of obtaining all eight unique representations is reduced to 40 clock cycles [9]. As a result the total average clock cycle count per iteration is approximately 580, i.e., a slow-down by a factor of less than 1.3 compared to Table 3. The expected number of iterations is about 1.25 times the square root of the group size, and may thus be up to about $256 = 2^{(128-112)/2}$ times larger for 128-bit prime field ECDLPs compared to the 112-bit prime field ECDLP solved in Section 4.6. These figures do not take the improvements suggested in [6] into account.

For larger moduli overheads will have a relatively smaller effect on the cycle count, so that quadratic cost extrapolation for schoolbook multiplication will be too pessimistic. Thus, extrapolation of the figures from Table 3 to larger prime moduli must be done with care and should be validated by experiments. Nevertheless, naive extrapolation of the above 580 cycle count per iteration for 128-bit primes to $\left(\frac{160}{128}\right)^2 \cdot 580 \approx 900$ cycles per iteration for a 160-bit prime turns out to be reasonably accurate [9]. This takes into account the smaller number of sequential processes because field elements require more storage. The growth of the expected number of iterations by a factor of up to $65536 = 2^{(160-128)/2}$, however, is more dramatic and implies that 160-bit prime ECDLPs are currently out of reach.

For larger prime fields it may pay off to replace schoolbook multiplication by a Karatsuba-like method. But all efforts to bend the quadratic growth of the cost of the underlying arithmetic are made futile by the unavoidable growth of the number of iterations: the sheer number of iterations will for the moment preclude realistic attempts to solve ECDLPs over such fields.

Table 3 Average clock cycle count for the operations carried out during an iteration of Pollard's rho method on a single SPU that performs 50 sequential processes, each consisting of two interleaved 4-way SIMD iterations, for a total of 400 simultaneous walks per SPU.

Operation (sloppy modulus $\tilde{p} = 2^{128} - 3$, modulus $p = \frac{\tilde{p}}{11 \cdot 6949}$)	Average # cycles per two interleaved 4-SIMD operations	Average # cycles per operation	Operations per iteration	Average # cycles per iteration
Sloppy multiplication modulo $\tilde{p}$ (multiplication+reduction)	430 $(318 + 112)$	54 $(40 + 14)$	6	322
Modular subtraction	40 even, 24 odd: 40 total	5	6	30
Modular inversion	n/a	4941	$\frac{1}{400}$	12
Unique representation mod p	192	24	1	24
Miscellaneous	544	68	1	68
Total				456

5 Conclusion

We developed SIMD multiplication modulo primes of the form $\frac{2^{32\ell} \pm m}{c}$ for small $\ell, m, c \in \mathbf{Z}_{>0}$ that achieves a speedup of approximately 30% over more traditional methods. It uses a redundant representation modulo $2^{32\ell} \pm m$ and a truncation-based reduction method, whose probability to produce an incorrect result has been argued to be very small. The method is suitable for error-tolerant applications, such as cryptanalytic ones.

As an application, we have shown the cryptanalytic potential of a commonly available toy by using a cluster of PlayStation 3 game consoles to solve an elliptic curve discrete logarithm problem over a 112-bit prime field. The runtimes and their extrapolations provide upper bounds for the effort required to solve larger instances of the same problem using a larger network of game consoles. Such a network is in principle accessible using programs such as BOINC [1]. Although surreptitious application of such programs would not be difficult to arrange for any miscreant who desires to do so, the effort required to solve a "practically relevant" problem remains staggering.

It was shown that the tag-tracing method from [19] can in principle be applied in elliptic curve context as well, but that scenarios are limited where the proposed method could lead to a speedup. The work reported here triggered deeper investigations into the negation map and the resulting fruitless cycles, and led to the concept and analysis of recurring cycles, reported in more detail in [13].

Acknowledgements

We gratefully acknowledge comments by two reviewers. This work was supported by the Swiss National Science Foundation under grant numbers 200021-119776, 206021-117409 and 206021-128727 and by EPFL DIT.

References

[1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[2] D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. V. Herrewege, and B.-Y. Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. `http://eprint.iacr.org/2009/541`.

[3] J.-C. Bajard, N. Meloni, and T. Plantard. Efficient RNS bases for cryptography. In *IMACS'05 : World Congress: Scientific Computation Applied Mathematics and Simulation*, 2005.

[4] A. Bender and G. Castagnoli. On the implementation of elliptic curve cryptosystems. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 186–192. Springer, 1989.

[5] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *PKC 2006*, volume 3958 of *LNCS*, pages 207–228, 2006.

[6] D. J. Bernstein. Faster ECDL. Rump-session talk at CHES 2010, slides available from `http://cr.yp.to/talks/2010.08.19-2/slides.pdf`, 2010.

[7] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. 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`.

[8] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Asiacrypt 2007*, volume 4833 of *LNCS*, pages 29–50, 2007.

[9] J. W. Bos. High-performance modular multiplication on the Cell processor. In *WAIFI 2010*, volume 6087 of *LNCS*, pages 7–24, 2010.

[10] J. W. Bos, A. Dudeanu, and D. Jetchev. Optimal collision bounds in the Pollard rho algorithm for discrete logarithms using additive walks. Work in progress, 2010.

[11] J. W. Bos and M. E. Kaihara. Montgomery multiplication on the Cell. In *Parallel Processing and Applied Mathematics 2009*, volume 6067 of *LNCS*, pages 477–485. Springer, 2010.

[12] J. W. Bos, M. E. Kaihara, and P. 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`.

[13] J. W. Bos, T. Kleinjung, and A. K. Lenstra. On the use of the negation map in the Pollard rho method. In *ANTS-IX*, volume 6197 of *LNCS*, pages 67–83, 2010.

[14] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Pushing the limits of ECM. Cryptology ePrint Archive, Report 2010/338, 2010. `http://eprint.iacr.org/2010/338`.

[15] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In *Africacrypt 2010*, volume 6055 of *LNCS*, pages 225–242, 2010.

[16] R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36(154):627–630, 1981.

[17] Certicom. Press release: Certicom announces elliptic curve cryptosystem (ECC) challenge winner. `http://www.certicom.com/index.php/2002-press-releases/38-2002-press-releases/340-notre-dame-mathematician-solves-eccp-109-encryption-key-problem-issued-in-1997`, 2002.

[18] Certicom Research. Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters. Standard SEC2, Certicom, 2000.

[19] J. H. Cheon, J. Hong, and M. Kim. Speeding up the Pollard rho method on prime fields. In *Asiacrypt 2008*, volume 5350 of *LNCS*, pages 471–488, 2008.

[20] N. Costigan and P. Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In *Africacrypt 2009*, volume 5580 of *LNCS*, pages 368–385. Springer, 2009.

[21] N. Costigan and M. Scott. Accelerating SSL using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. Cryptology ePrint Archive, Report 2007/061, 2007. `http://eprint.iacr.org/2007/061`.

[22] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system, October 1992. U.S. patent number 5,159,632.

[23] I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In *Asiacrypt 1999*, volume 1716 of *LNCS*, pages 103–121, 1999.

[24] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, July 2007.

[25] S. Galbraith. Mathematics of public key cryptography (version 0.6). `http://www.isg.rhul.ac.uk/~sdg/crypto-book/crypto-book.html`, 2010.

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

[27] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA 2005*, pages 258–262, 2005.

[28] G. Hotz. Here's your silver plate. `http://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/`.

[29] IBM. Multi-precision math library. Example Library API Reference. Available at `https://www.ibm.com/developerworks/power/cell/documents.html`.

[30] M. E. Kaihara and N. Takagi. A hardware algorithm for modular multiplication/division. *IEEE Trans. Computers*, 54(1):12–21, 2005.

[31] B. S. Kaliski. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[32] J. H. Kim, R. Montenegro, Y. Peres, and P. Tetali. A birthday paradox for Markov chains, with an optimal bound for collision in the Pollard rho algorithm for discrete logarithm. *The Annals of Applied Probability*, 20(2):495–521, 2010.

[33] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *CRYPTO 2010*, volume 6223 of *LNCS*, pages 333–350, 2010.

[34] D. E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.

[35] D. E. Knuth. *Sorting and Searching*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1998.

[36] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[37] N. Koblitz. CM-curves with good cryptographic properties. In *Crypto 1991*, volume 576 of *LNCS*, pages 279–287, 1992.

[38] A. K. Lenstra and H. W. Lenstra, Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verslag, 1993.

[39] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[40] V. S. Miller. Use of elliptic curves in cryptography. In *Crypto 1985*, volume 218 of *LNCS*, pages 417–426, 1986.

[41] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

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

[43] G. Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135–140, 2004.

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

[45] RSA the security division of EMC. The RSA challenge numbers. formerly on `http://www.rsa.com/rsalabs/node.asp?id=2093`, now on `http://en.wikipedia.org/wiki/RSA_numbers`.

[46] E. Schulte-Geers. Collision search in a random mapping: some asymptotic results. Talk at ECC 2000, The Fourth Workshop on Elliptic Curve Cryptography, Essen, Germany, 2000, Slides available from `http://www.cacr.math.uwaterloo.ca/conferences/2000/ecc2000/slides.html`, 2000.

[47] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research, University of Waterloo, 1999.

[48] J. A. Solinas. Cryptographic identification and digital signature method using efficient elliptic curve, May 2005. U.S. patent number 6,898,284.

[49] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *CRYPTO*, volume 5677 of *LNCS*, pages 55–69, 2009.

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

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

[52] S. Wedeniwski. Piologie an exact arithmetic library in C++. Website: `http://www.zetagrid.net/zeta/sourcecode.html`, iPhone application: `http://itunes.apple.com/app/piologie/id387334278?mt=8`, 2010.

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