Elliptic and Hyperelliptic Curves: a Practical Security Analysis

Joppe W. Bos[1], Craig Costello[1], and Andrea Miele[2]

[1] Microsoft Research, Redmond, WA, USA
[2] LACAL, EPFL, Lausanne, Switzerland

Abstract. Motivated by the advantages of using elliptic curves for discrete logarithm-based public-key cryptography, there is an active research area investigating the potential of using hyperelliptic curves of genus 2. For both types of curves, the best known algorithms to solve the discrete logarithm problem are generic attacks such as Pollard rho, for which it is well-known that the algorithm can be sped up when the target curve comes equipped with an efficiently computable automorphism. For the first time, we perform a systematic security assessment of elliptic curves and hyperelliptic curves of genus 2, by incorporating all of the known optimizations. We use our software framework to give concrete estimates on the number of core years required to solve the discrete logarithm problem on four curves that target the 128-bit security level: on the standardized NIST CurveP-256, on a popular curve from the Barreto-Naehrig family, and on their respective analogues in genus 2.

1 Introduction

In the last couple of decades, the use of elliptic curves, or *genus 1 curves*, has become a popular and standardized choice to instantiate public-key cryptography [25, 29]. The security of these cryptographic schemes relies on the difficulty of the elliptic curve discrete logarithm problem (ECDLP). Currently, the best known algorithms to solve this problem are the so-called "generic" attacks, such as the parallelized version [36] of the Pollard rho algorithm [32], which has been used to solve large instances of the ECDLP (cf. [22, 12, 8, 2]). It is well-known that this algorithm can be optimized by a constant factor when the target curve comes equipped with an efficiently computable group automorphism [38, 15]. For example, all elliptic curves can efficiently compute the inverse of a point and this negation map can be used to speed up the run-time by at most a factor $\sqrt{2}$. When the cardinality of the automorphism group is larger, such as for the elliptic curves proposed in [18, 17], a higher speedup is expected when solving the ECDLP.

Jacobians of hyperelliptic curves of genus 2 have also been considered for cryptographic applications [26] (also see [5, 27]). Just as with their elliptic curve counterpart, the best known algorithms to solve the discrete logarithm in such groups are the generic ones. The practical potential of genus 2 curves in public-key cryptography has recently been highlighted by the fast performance numbers presented in [7]. Since larger automorphism groups are possible in genus 2, higher-dimensional scalar decompositions are possible on certain families of curves [17]. This not only aids the cryptographer, but also the cryptanalyst: one can expect a larger speed-up when computing the (H)ECDLP on curves from these families [15].

In this paper we investigate the *practical* speed-up of Pollard rho when exploiting the automorphism group. We use the methods presented in [9, 6] for situations where only the negation map is available, and extend these techniques to curves with a larger group automorphism. As examples in the elliptic case, we use two curves that target the 128-bit security level: the NIST Curve P-256 [35] and a BN-curve [3] – the automorphism groups on these two curves are of size two and six respectively, which are the minimum and maximum possible sizes in

genus 1. To mimic these choices in the hyperelliptic case[3], we use two curves from [7], where the automorphism groups are of size two and ten – these are the minimum and maximum possible sizes in genus 2. We implemented efficient field and curve arithmetic that was optimized for each of these four curves, and derived the best parameters to make use of the automorphism optimization. To our knowledge, this is the first time the security of genus 1 and genus 2 curves has been systematically compared.

We obtain security estimates for these four curves using parameters and implementations that were devised to minimize the practical inconveniences arising from the group automorphism optimization. When taking the standardized NIST Curve P-256 as a baseline for the 128-bit security level, we show that curves with a larger automorphism group (of cardinality $m > 2$) indeed sacrifice some security. The constant-factor speedup, however, is lower in practice than the often cited $\sqrt{m}$. Nevertheless, using both theoretical and experimental analysis, we provide parameters which push the performance of the Pollard rho algorithm close to what can be achieved in practice.

The paper is organized as follows. We give preliminaries on the Pollard rho algorithm in Section 2. In Section 3 we discuss the main practical issues that arise from the automorphism optimization, and how to minimize their impact on performance. In Section 4 we give the fine-grained details and parameter choices for each of the curves under consideration, before discussing our implementations and results in Section 5. We conclude the paper in Section 6.

2 Preliminaries

General group elements. We use $\mathcal{J}_C$ to denote the Jacobian group of a curve C over a finite field $\mathbf{F}_q$, where $q > 3$ is prime. For our purposes, C and $\mathcal{J}_C$ can be identified when C is an elliptic curve, where our group elements are all points $(x, y) \in \mathbf{F}_q \times \mathbf{F}_q$ satisfying $C/\mathbf{F}_q : y^2 = x^3 + ax + b$, together with the identity element $\mathcal{O}$. In genus 2, our curves are assumed to be of the form $C/\mathbf{F}_q : y^2 = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$. In this case we write general elements of the Jacobian group (i.e. weight 2 divisors) in their *Mumford representation* as $(u(x), v(x)) = (x^2 + u_1x + u_0, v_1x + v_0) \in \mathbf{F}_q[x] \times \mathbf{F}_q[x]$, such that $u(x_1) = u(x_2) = 0$, $v(x_1) = y_1$ and $v(x_2) = y_2$, where (x_1, y_1) and (x_2, y_2) are two (not necessarily distinct) points in the set $C(\overline{\mathbf{F}}_q)$. The canonical embedding of C into $\mathcal{J}_C$ maps $(x_1, y_1) \in C(\mathbf{F}_q)$ to the divisor with Mumford representation $(x - x_1, y_1)$ – we call such divisors *degenerate*. Since $\#C \approx p$ and $\#\mathcal{J}_C \approx p^2$, the probability of randomly choosing a degenerate divisor from $\mathcal{J}_C$ is $O(\frac{1}{p})$; this is also the probability that the sum of two random elements in $\mathcal{J}_C$ is a degenerate divisor [31, Lemma 1]. Combining these probabilities with standard Pollard rho heuristics allows us to ignore the existence of degenerate divisors in practice – in all of the cases considered in this work, it is straightforward to see that an optimized random walk is more likely to solve the discrete logarithm problem than it is to walk into a degenerate divisor. Note that in the unlikely event one encounters a degenerate divisor, such that our general-case formulas compute divisors which are not on the Jacobian, this can be dealt with at almost no additional cost by performing a sanity check on all active walks, once in a while. Another solution is to perform such a sanity check on the distinguished elements only (see the description of the parallel Pollard rho algorithm below) and to discard such incorrect elements.

[3] The fact that the BN curve is pairing-friendly, while our chosen genus 2 "analogue" is not, does not make a difference in the context of our Pollard rho analysis. We wanted curves with maximal automorphism group sizes, and we deemed *the* BN curve to be the most interesting such choice in genus 1.

The Pollard rho algorithm. The Pollard rho algorithm [32] can be used to compute discrete logarithms in arbitrary groups, but here we give a description that is specific to our context of Jacobian groups. Suppose we are given $P \in \mathcal{J}_C$ that generates a group of large prime order n: given some $Q \in \langle P \rangle$, the (hyper-)elliptic curve discrete logarithm problem (H)ECDLP is to find $k \in \mathbf{Z}/n\mathbf{Z}$ such that $Q = [k]P$. At the highest level, the idea is to compute pseudo-random elements of the form $P_i = [a_i]P + [b_i]Q$ for known non-zero $a_i, b_i \in \mathbf{Z}/n\mathbf{Z}$, such that if a collision $P_i = P_j$ is found with $b_i \neq b_j$, then taking $k := (a_j - a_i)/(b_i - b_j) \in \mathbf{Z}/n\mathbf{Z}$ is a solution to the (H)ECDLP. The *birthday paradox* implies that we can expect to find such a collision after computing around $\sqrt{\frac{\pi n}{2}}$ group elements P_i, provided they are chosen independently and uniformly at random [23]. In practice we use the so-called *r-adding walk*, which starts with r precomputed group elements $S_j = [c_j]P + [d_j]Q$ for non-zero $c_j, d_j \in \mathbf{Z}/n\mathbf{Z}$ and $0 \leq j < r$. On input of a group element P_i, we use a *partition function* $\ell : \langle P \rangle \to \{0, 1, \ldots, r-1\}$ to define an *iteration function* $f : \langle P \rangle \to \langle P \rangle$, which computes the next element as $P_{i+1} = f(P_i) = P_i + S_{\ell(P_i)}$. Put simply, the iteration function chooses one of the r precomputed elements to add to P_i in order to step to P_{i+1}. On top of the minor costs of evaluating ℓ and updating the $a_i, b_i \in \mathbf{Z}/n\mathbf{Z}$, each such step comes at the cost of a single Jacobian group operation. Keeping every group element encountered in the walk imposes exponential (and therefore infeasible) storage requirements, which is why the parallel Pollard rho algorithm [36] stores only a small fraction of the elements we come across: the so-called *distinguished points*. Storage of $O(\log n)$ group elements suffices when roughly $\sqrt{n} \log n$ out of n group elements are distinguished [16, Exercise 14.2.17]. In practice one can use a simple check to determine whether the group element P_i is classed as distinguished, in which case it is reported to a central location, along with the corresponding a_i and b_i. Only these distinguished 'points' need to be cross-checked against one another for collisions; when two walks coincide at a non-distinguished point and this collision goes undetected, the deterministic iteration function guarantees that these walks continue along the same path until they arrive at the same distinguished point.

Affine additions with amortized inversions. As mentioned above, each step of a random walk requires the addition of two distinct Jacobian group elements. In the context of scalar multiplications, additions on the Jacobian are usually performed in projective space, where all inversions are avoided until the very end, at which point the result is normalized via a single inversion. In the context of Pollard rho however, it is preferred to work in affine space for two main reasons. Firstly, we need a way to suitably define and efficiently check a distinguished point criterion on *every* group element that is computed; clearly there is no computationally efficient way to do this if unique affine elements are represented using a projective representation. Secondly, optimized versions of Pollard rho run many concurrent random walks to take advantage of Montgomery's simultaneous inversion method [30]. If enough concurrent walks are used, then the amortized cost of each individual field inversion becomes roughly 3 field multiplications – this makes affine Weierstrass coordinates the fastest known coordinate system to work with for cryptanalysis. On elliptic curves, such amortized point additions require 5 $\mathbf{F}_q$ multiplications, 1 $\mathbf{F}_q$ squaring and 6 $\mathbf{F}_q$ additions; on genus 2 curves, these additions cost 20 $\mathbf{F}_q$ multiplications, 4 $\mathbf{F}_q$ squarings and 48 $\mathbf{F}_q$ additions [14] – see Table 1 in Section 4.

Exploiting automorphisms. The Pollard rho algorithm can be sped up by a constant factor if the presence of automorphisms on C is exploited [38, 15]. Let m denote the cardinality of the automorphism group, $\mathrm{Aut}(C)$, which we assume is cyclic with generator ψ; in genus 2, ψ extends in the natural way to $\mathcal{J}_C$ under the canonical embedding described above. For all

$R, R' \in \langle P \rangle$, define an equivalence relation $\sim$ on $\langle P \rangle$ by $R \sim R'$ if and only if $R = \psi^i(\hat{R})$ and $R' = \psi^j(\hat{R})$, for some $\hat{R} \in \langle P \rangle$ and $0 \leq i, j < m$. Note that there are around n/m such equivalence classes in $\langle P \rangle$, and that $m \geq 2$ since $\mathrm{Aut}(C)$ contains (at least) the identity map id and the negation/involution map "$-$". We write $\tilde{R}$ for the unique *representative* of the class containing R, i.e. $\tilde{R_1} = \tilde{R_2}$ if and only if $R_1 \sim R_2$. An efficient way of choosing such representatives is imperative to an optimized implementation of the Pollard rho algorithm, so we give the fine-grained details for each of the curves under consideration in Section 4. The important point is that each time the iteration function computes a new group element P_{i+1} via an addition, it now immediately computes *the* representative element $\tilde{P}_{i+1}$, thereby accounting for m elements at a time. This effectively reduces the size of the set on which we walk by a factor of m, which theoretically reduces the expected time to a collision by a constant factor $\sqrt{m}$. In practice however, computing these representatives incurs an overhead which reduces the actual speedup obtained; one of the contributions of this work is to optimize parameter selection in a variety of scenarios to see how close we can get to this theoretical $\sqrt{m}$ improvement.

3 Handling Fruitless Cycles

It is well known that certain practical issues are encountered when exploiting the automorphism optimization [38, 18, 15, 9, 6]. Walks will end up in *fruitless cycles* – endless small loops where many fruitless collisions are found over-and-over again (the collisions are fruitless because they have the same a_i and b_i). At a high level, these collisions occur because the automorphism ψ, which generates $\mathrm{Aut}(C)$, has a minimal polynomial of small degree; for all scenarios in this paper, ψ satisfies $\sum_{i=0}^{d} e_i \psi^i = 0$ for $e_i \in \mathbf{Z}$ and where $d \leq 5$. Since each step in a walk involves the addition of an element from a relatively small fixed table, it is possible that the same table element (or a very small subset of them) is added multiple times in succession, and that these contributions to the walk are annihilated by unfortunate linear combinations of powers of ψ (which sum to zero). The most simple and frequently occurring example is when the negation map sends us into a fruitless 2-cycle: the partition function will choose the same table element twice in a row (i.e $\ell(P_i) = \ell(P_{i+1}) = \ell(P_i + S_{\ell(P_i)})$) with probability $1/r$, and the representative $\tilde{P}_{i+1}$ of the equivalence class $\{P_{i+1}, -P_{i+1}\}$ will be $\tilde{P}_{i+1} = -P_{i+1} = -(P_i + S_{\ell(P_i)})$ with probability $1/2$, meaning that $\tilde{P}_{i+2} = \tilde{P}_i$ with probability $1/(2r)$. This is analyzed in more detail for different cycle lengths and values of $m = \#\mathrm{Aut}(C)$ in [15].

In this section we summarize the current literature and discuss how to reduce the occurrence of fruitless cycles, how to detect when they occur, and subsequently how to deal with a walk that is stuck in such a cycle.

3.1 Cycle Reduction

In [38], a 'look-ahead' technique is described to reduce the event of 2-cycles. This method starts by computing a candidate point $\hat{P}$ for P_{i+1} as usual, i.e. computing $\hat{P} = P_i + S_{\ell(P_i)}$; if $\ell(\hat{P}) \neq \ell(P_i)$, then we set $P_{i+1} = \hat{P}$ and continue, otherwise we discard the point $\hat{P}$ and compute another candidate point by adding the next lookup table element $S_{\ell(P_i)+1 \bmod r}$ to P_i. Note that the probability that r lookup elements result in invalid candidates is extremely low, i.e. r^{-r}. As analyzed in [9], using this look-ahead technique lowers the probability to enter a 2-cycle from $\frac{1}{2r}$ to $\frac{1}{2r^3} + \mathcal{O}(\frac{1}{r^4})$. This technique can be generalized to longer cycles as

well [38, 9]. Note that if a point gets discarded, it means that we have computed the group operation but did not take a step forward in our pseudo-random walk. We refer to this event as a *fruitless step due to cycle reduction*. In this work we use a 2-cycle reduction technique that slightly modifies the above approach, as we detail in Section 3.3.

3.2 Escaping Fruitless Cycles

Even if the probability of a fruitless cycle is lowered using the look-ahead strategy in Section 3.1, the walks will still eventually enter a fruitless cycle, which clearly must be dealt with. The first step towards a remedy is to detect that a walk is trapped; the next step is to then escape the fruitless cycle in a deterministic way, such that if any other walk encounters the same cycle, they both end up exiting using the exact same point. The idea described in [18] is to occasionally store a sequence of points and to check for repetitions by comparing new points to these stored points. If a cycle has been detected, then one can escape by applying a modified iteration function to a representative of the cycle – in [18], the point with smallest x- or y-coordinate is proposed to be the representative. In [9] it is observed that many modified iteration functions used to escape the cycle are insufficient, and can result in the walk recurring to the same fruitless cycle soon after it "escapes". As observed in [15, 9], one example of how to properly escape cycles is to *double* the representative of the fruitless cycle – our implementations use this approach.

3.3 Handling Fruitless Cycles in Practice

In this subsection we compute a lower-bound on the number of fruitless steps we expect to perform in order to state an upper-bound on the (theoretical) speedup. For this analysis, we measure the cost of the additional (fruitless) computations we have to perform in order to deal with cycles. To analyze this cost, we use a function c which expresses the cost of certain operations in terms of the number of modular multiplications. We summarize which strategy we use in our implementation and outline how we select the various parameters, based on our analysis, to perform cycle reduction and cycle escaping.

In [9], different scenarios and varied parameters for both cycle reduction and cycle escaping techniques are implemented and compared. The recommendations are to use medium sized values of r (since larger values might decrease the performance by introducing cache-misses), to reduce the event of 2-cycles only (not any higher cycles), and to escape cycles by doubling the cycle's representative. This combination of choices was able to achieve a 1.29 times speedup over not using the negation map on architectures supporting the `x64` instruction set, while from a theoretical perspective a speedup of 1.38 should be possible (both speedups are slightly below $\sqrt{2}$). A follow-up paper [6] takes a different approach on the single instruction, multiple data (SIMD) Cell processor. Since multiple walks are processed by the same instructions, all of which must follow identical computational steps, the cycle reduction technique is completely omitted. Instead, the walk is modified to occasionally check for fruitless cycles – different cycle lengths are detected at different points in time, but if a cycle is detected, this is resolved by escaping from it by again doubling the cycle's representative.

We now analyze the maximum expected speedup in more detail. Assume we perform $w > 0$ steps, and that at every step we can enter a cycle with probability p, if we are not in a cycle already. Once we enter a cycle at step $0 < i \leq w$, all subsequent $w - i$ steps are fruitless.

Hence, after w steps we expect to have computed $W(w,p)$ fruitless steps where

$$W(w,p) = \sum_{i=0}^{w-1} p(1-p)^i(w-i) = \frac{(1-p)^{w+1} + p(w+1) - 1}{p}. \quad (1)$$

Using this simple analysis, one can compute the ratio between the number of fruitful steps and the number of total steps. For example, the implementation described in [6] uses $r = 2048$, checks for 2-cycles every 48 iterations, and checks for larger cycles much less frequently. Since 2-cycles occur with probability $\frac{1}{2r}$, the expected number of multiplications due to fruitful steps (per 48 iterations) is $c(f) \cdot (48 - W(48, \frac{1}{2 \cdot 2048}))$, where $c(f)$ is the cost to compute the iteration function expressed in multiplications, which in this setting is $c(f) = 6$. The total number of multiplications computed is then $48 \cdot c(f) + c(D)$, where the latter is the cost for point doubling in order to escape the 2-cycle, which is $c(D) = 7$ in the elliptic curve case. Ignoring the various implementation overheads, this analysis shows that a speedup of at most $0.97\sqrt{2}$ is expected when taking only 2-cycles into account.

In our implementations, we chose to follow an approach closer to that which is described in [9]. The reason is that we *do* want to use the cycle reduction technique to lower the probability for walks to enter 2-cycles (at the price of occasionally computing fruitless cycles due to cycle reduction). We remark that in a SIMD setting, such as that considered in [6], an approach without cycle reduction might be more efficient in practice. We note that using the 2-cycle reduction technique also reduces the event of 3-cycles, which can only occur if $3 \mid \#\mathrm{Aut}(C)$, for which the BN curve is the only such scenario in this paper. As shown in [15], 3-cycles occur only if we add representatives from the same partition three times in a row – this repetition is exactly what we aim to avoid using the 2-cycle reduction technique.

We check for cycles every α steps by recording the β points $\{\alpha, \alpha+1, \ldots, \alpha+\beta-1\}$, and checking if the $(\alpha+\beta)$th point occurs in the list of recorded points. If it does, then we select a fruitless cycle representative and use this point to double out of this fruitless cycle: this heuristically eliminates recurring cycles [9].

We modify the cycle reduction technique from [38, 9], as described in Section 3.1. In order to avoid, with probability r^{-r}, the scenario where all of the r lookup table elements give rise to an invalid next point, we simply add a point from another precomputed lookup table $\tilde{f}$ (which also contains r elements), as follows:

$$p_{i+1} = \begin{cases} p_i + f_{\ell(p_i)} \text{ if } \ell(p_i) \neq \ell(p_i + f_{\ell(p_i)}), \\ p_i + \tilde{f}_{\ell(p_i)} \text{ otherwise.} \end{cases}$$

Following the analysis from [9], this reduces the probability to enter a 2-cycle from $(mr)^{-1}$ to approximately $\frac{4}{(mr)^3}$. For practical values of r, this makes 4-cycles the most likely event to occur, with probability approximately $(mr)^{-2}$. Due to this cycle reduction technique, we expect that one out of r steps is fruitless (since the probability that $\ell(p_i) = \ell(p_i + f_{\ell(p_i)})$ is $\frac{1}{r}$). Hence, the fraction of all steps that are fruitful is $\frac{r-1}{r}$.

4 Target Curves and their Automorphism Groups

In this section we discuss our chosen target curves and the associated parameter choices and optimizations in the context of Pollard rho. The computational costs for divisor addition, computing the equivalence class representative, and updating the a_i and b_i values are summarized

in the worst and average case in Table 1. The average case costs are used in our analysis, but we include the worst case costs for settings (like parallel architectures) where all the walks must always perform the same (worst-case) computational steps.

We choose to target two curves in genus 1 and two curves in genus 2. All four of these curves have a prime order between 254 and 256 bits. The two elliptic curves have $m = 2$ and $m = 6$, which are the respective minimum and maximum values of $m = \#\mathrm{Aut}(C)$ in genus 1; likewise, the two hyperelliptic curves have $m = 2$ and $m = 10$, which are the respective minimum and maximum values of $m = \#\mathrm{Aut}(C)$ in genus 2.

In each case we also outline our parameter choices for handling fruitless cycles. We follow the analysis and notation as outlined in Section 3.3, with a primary goal that less than one percent of the steps we compute are fruitless. In order to sufficiently reduce the probability of cycles to occur, we always take $r \geq 1024$. Furthermore, in order to detect much longer (and much less likely) cycles, we take $\beta = 32$, so that we can detect and deal with cycles up to length 32. More precisely, given a probability p to enter a cycle at every step, and a value for α (we check for cycles every α steps), we estimate the fraction of all computation that is *fruitful* using Eq. (1), as

$$\frac{c(f) \cdot (\alpha - W(\alpha, p))}{\alpha \cdot c(f) + c(D)} \cdot \frac{r-1}{r}, \tag{2}$$

where the first fraction is due to the cycle detection and escaping (we assume that we always compute a doubling to escape), and the second fraction incorporates the fruitless steps due to the cycle reduction technique.

In this section we assume that the cost of a modular multiplication and modular squaring are equivalent: if required, the analysis can be trivially adjusted to reflect any other cost ratio. Although we give the costs of updating the a_i and b_i, we omit these from our analysis – the correct a_i and b_i can be recovered when needed, when each path starts at a random point derived from a random seed, as described in [2].

4.1 Target Curves in Genus 1

NIST CurveP-256. Let $q = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, and define $E/\mathbf{F}_q\colon y^2 = x^3 - 3x + b$, with $b =$`0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B`. This curve has a 256-bit prime order n and is defined in the NIST standard [35]. In this case $\mathrm{Aut}(E) = \{id, -\}$, meaning that $(x, y) \sim (x, -y)$, so we take the representative of each class to be the point with the odd y-coordinate (when $0 \leq y < q$). In the worst case, the cost of computing this representative is a negation in $\mathbf{F}_q$, and updating the corresponding (a_i, b_i) pair costs two negations in $\mathbf{Z}/n\mathbf{Z}$. On average though, these costs are halved, since we have already computed (and detected) the representative half of the time.

In order to derive parameters for the cycle detection, we use $p = (2r)^{-2}$ as the probability to enter a 4-cycle, which (due to the cycle-reduction technique) is higher than the probability to enter a 2-cycle – see Section 3.3. The elliptic curve group operation costs are taken as $c(f) = c(A) = 6$ and $c(D) = 7$. Using the parameters $r = 1024$, $\alpha = 7 \cdot 10^4$ and $\beta = 32$, we expect that around one percent of the computed steps are fruitless: Eq.(2) evaluates to 0.9907.

BN254. Let q be the 254-bit prime obtained when $u = -(2^{62} + 2^{55} + 1)$ is plugged into $q(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$. The Barreto-Naehrig (BN) curve [3] $E/\mathbf{F}_q\colon y^2 = x^3 + 2$ has a 254-bit prime order n, and has been used in several of the "speed-record" papers for

pairing computations that target the 128-bit security level (e.g. [1, 21]). Since $q \equiv 1 \bmod 3$, there exists $\zeta \neq 1 \in \mathbf{F}_q$ such that $\zeta^3 = 1$, meaning that $E(\mathbf{F}_q)$ has additional automorphisms, e.g. $\phi : E \to E, (x, y) \mapsto (\zeta x, y)$. In fact, $\mathrm{Aut}(E) = \{id, -, \phi, -\phi, \phi^2, -\phi^2\}$, so that the points (x, y), $(x, -y)$, $(\zeta x, y)$, $(\zeta x, -y)$, $(\zeta^2 x, y)$ and $(\zeta^2 x, -y)$ are all equivalent under $\sim$. We take the representative of each equivalence class to be the point whose x-coordinate has least absolute value *and* whose y-coordinate is odd. In the worst case, computing this representative costs one multiplication, two negations and one addition in $\mathbf{F}_q$, and updating the corresponding (a_i, b_i) pair costs two multiplications in $\mathbf{Z}/n\mathbf{Z}$; we exploit $\zeta^2 x = -(\zeta + 1)x$ to compute the x-coordinate of $\phi^2(P)$ from the x-coordinates of $\phi(P)$ and P without any further multiplications. On average however, we only need the negation to get the odd y-coordinate half of the time; to update the (a_i, b_i), we compute the two $\mathbf{Z}/n\mathbf{Z}$ multiplications two thirds of the time, while in the remaining one third of the cases, we average a single $\mathbf{Z}/n\mathbf{Z}$ addition.

In order to derive parameters for the cycle detection, we use $p = (6r)^{-2}$ as the adjusted probability to enter a 4-cycle (taking the group automorphism into account). In this case the elliptic curve group operation costs are taken as $c(f) = c(A) = 7$ and $c(D) = 8$, where both costs incorporate the additional multiplication to compute the representative. Using $r = 1024$ and $\beta = 32$, we find that a corresponding α value (for which we expect that around one percent of the computed steps is fruitless) as $\alpha = 6 \cdot 10^5$, which is almost an order of magnitude larger than in the NIST CurveP-256 setting: in this case, evaluating Eq. (2) gives 0.9911.

4.2 Target Curves in Genus 2

Generic1271. Let $q = 2^{127} - 1$ and $C/\mathbf{F}_q \colon y^2 = x^5 + a_3x^2 + a_2x^2 + a_1x + a_0$ with

$$a_3 = \texttt{0x1A237F07B8BB79AEBA5011C3FA697D2D},\ a_2 = \texttt{0x63D7B6834F8A4F3DBDBD141CE55EA675},$$
$$a_1 = \texttt{0x44642D7B9E492BE2E3C4F8A36F0C4236},\ a_0 = \texttt{0x504351F67810EFACF06E3A6E5C532F0}.$$

This curve was recently used in [7] as a "generic" instance of a (degree 5) genus 2 curve, since it has no special structure and the order of its Jacobian is a 254-bit prime n. Here $\mathrm{Aut}(C) = \{id, -\}$, which extends to $\mathcal{J}_C$ to give that the divisors $(x^2 + u_1x + u_0, v_1x + v_0)$ and $(x^2 + u_1x + u_0, -v_1x - v_0)$ are equivalent under $\sim$. Thus, we take the representative of each class to be the divisor whose v_0-coordinate is odd. In the worst case, the cost of computing this representative is two negations in $\mathbf{F}_q$, and updating the corresponding (a_i, b_i) pair costs two negations in $\mathbf{Z}/n\mathbf{Z}$. On average these costs are again halved since we already have the correct representative half of the time.

In order to derive parameters for the cycle detection, we use exactly the same parameters as in the NIST CurveP-256 setting, since the automorphism groups are the same, and only the costs of the group operations differ: $c(f) = c(A) = 24$ and $c(D) = 28$ in this case. Using the same α results in the same expectation of less than one percent computed fruitless steps – Eq.(2) evaluates to 0.9907 when $\alpha = 7 \cdot 10^4$.

4GLV127-BK. Let $q = 2^{64} \cdot (2^{63} - 27443) + 1$. The Buhler-Koblitz [11] curve $C/\mathbf{F}_q \colon y^2 = x^5 + 17$ gives rise to a Jacobian whose group order is a 254-bit prime n. Since $q \equiv 1 \bmod 5$, there exists $\zeta \neq 1$ in $\mathbf{F}_q$ such that $\zeta^5 = 1$, which gives rise to additional automorphisms on C, e.g. $\phi : C \to C, (x, y) \mapsto (\zeta x, y)$. The map ϕ extends to weight-2 divisors as $\phi : \mathcal{J}_C \to \mathcal{J}_C, (x^2 + u_1x + u_0, v_1x + v_0) \mapsto (x^2 + \zeta u_1x + \zeta^2 u_0, \zeta^4 v_1x + v_0)$. Here $\mathrm{Aut}(C) = \{id, -, \phi, -\phi, \ldots, \phi^4, -\phi^4\}$, so we take the representative of each class to be the divisor whose

Table 1. Cost of the Pollard rho iteration for the selected genus g curves, $m = \#\mathrm{Aut}$ and q is the prime field characteristic. We denote modular multiplications, modular squarings and modular additions/subtractions with **M**, **S** and **a** respectively. When updating the a_i and b_i values we compute modulo the cardinality n instead of modulo q.

curve	g	m	cost of one step				
			divisor	compute representative		update a_i, b_i	
			addition	worst	average	worst	average
CurveP-256	1	2	$5\mathbf{M}+\mathbf{S}+6\mathbf{a}$	$1\mathbf{a}$	$\frac{1}{2}\mathbf{a}$	$2\mathbf{a}_n$	$1\mathbf{a}_n$
BN254	1	6	$5\mathbf{M}+\mathbf{S}+6\mathbf{a}$	$1\mathbf{M}+3\mathbf{a}$	$1\mathbf{M}+2\frac{1}{2}\mathbf{a}$	$2\mathbf{M}_n$	$1\frac{1}{3}\mathbf{M}_n+\frac{1}{3}\mathbf{a}_n$
Generic1271	2	2	$20\mathbf{M}+4\mathbf{S}+48\mathbf{a}$	$2\mathbf{a}$	$1\mathbf{a}$	$2\mathbf{a}_n$	$1\mathbf{a}_n$
4GLV127-BK	2	10	$20\mathbf{M}+4\mathbf{S}+48\mathbf{a}$	$6\mathbf{M}+1\mathbf{S}+5\mathbf{a}$	$5\frac{2}{5}\mathbf{M}+\frac{4}{5}\mathbf{S}+\frac{3}{5}\mathbf{a}$	$2\mathbf{M}_n$	$1\frac{3}{5}\mathbf{M}_n+\frac{1}{5}\mathbf{a}_n$

u_1-coordinate has least absolute value *and* whose v_0-coordinate is odd. In the worst case, the cost of finding this representative is six multiplications, one squaring, three additions and two negations in $\mathbf{F}_q$; it takes three multiplications, three additions and a negation (this time we use $\zeta^4 = -(\zeta^3+\zeta^2+\zeta+1)$ to save a multiplication) to first determine the minimum value in $\{\zeta^i u_1\}$ for $0 \le i \le 4$, another two multiplications to compute the corresponding $\zeta^{2i}u_0$ and $\pm\zeta^{4i}v_1$, and finally one negation for the v_0-coordinate. To comply with the formulas in [14], we must also recompute the two extended coordinates u_1u_0 and u_1^2, which additionally incurs a multiplication and a squaring. Updating the (a_i, b_i) pair costs two multiplications in $\mathbf{Z}/n\mathbf{Z}$. On average though, we only need the three $\mathbf{F}_q$ multiplications and one $\mathbf{F}_q$ squaring for u_0, v_1, u_1u_0 and u_1^2 in eight of the ten cases (one of the ten needs only one $\mathbf{F}_q$ negation, the other case needs no computation), and we only need to negate v_0 in five of the ten cases. For updating (a_i, b_i) on average, we need two $\mathbf{Z}/n\mathbf{Z}$ multiplications in eight of the ten cases, two $\mathbf{Z}/n\mathbf{Z}$ negations in one of them, while the remaining case leaves (a_i, b_i) unchanged.

Taking the size of the automorphism group into account gives $p = (10r)^{-2}$ as the adjusted probability to enter a 4-cycle. Including the average number of additional multiplications to compute the representative of the equivalence class in the iteration function, the costs become $c(f) = 30\frac{1}{5}$ and $c(D) = 34\frac{1}{5}$. An α value for which we expect that around one percent of the computed steps is fruitless is $\alpha = 10^6$: this is over an order of magnitude larger compared to the Generic1271 setting: evaluating Eq.(2) gives 0.9943 in this case.

4.3 Other Curves of Interest

In this subsection we briefly mention the application of the Pollard rho algorithm to other popular curves that have appeared in the literature and that target the 128-bit security level.

Other genus 1 curves. Bernstein's Curve25519 [4] and Hisil's ecfp256e [24] both facilitate fast timings for scalar multiplications without the existence of additional morphisms, so besides the faster modular arithmetic that is possible over these pseudo-Mersenne primes, the application of Pollard rho to these two curves is identical to the case of CurveP-256.

There are other j-invariant zero curves (that are not pairing-friendly) which have been put forward for fast ECC using the Gallant-Lambert-Vanstone (GLV) technique [18]: the prime order curve $E/\mathbf{F}_q\colon y^2 = x^3 + 2$ with $q = 2^{256} - 11733$ was used by Longa and Sica [28], while the prime order curve $E/\mathbf{F}_q\colon y^2 = x^3 + 7$ with $q = 2^{256} - 4294968273$ is proposed in the SEC standards [13]. In both of these cases, the automorphism group is the same as that for BN254, so Pollard rho is optimized identically.

There exist numerous families of curves that come equipped with non-trivial morphisms which are useful in the context of scalar multiplications, but which are not useful in the context of Pollard rho. This is the case for curves that contain efficiently computable endomorphisms which are not automorphisms, like the families of $\mathbf{Q}$-curves recently proposed by Smith [33]. On the other hand, Galbraith-Lin-Scott (GLS) curves [17] do facilitate a constant-factor speedup in Pollard rho, since the GLS automorphism (which usually involves one multiplication by a fixed constant) is typically much faster than a group operation.

Other genus 2 curves. The authors of [7] recently used the Kummer surface found by Gaudry and Schost [20] to achieve fast scalar multiplications in genus 2. Interestingly, there is no known way to exploit the fast arithmetic on the Kummer surface in Pollard rho, since only pseudo-additions exist there. Discrete logarithm instances must therefore be mapped back to the full Jacobian group, where, besides the smaller prime subgroup resulting from the imposed cofactor of 16 on Kummer1271, the optimal application of Pollard rho is identical to the case of Generic1271.

In addition to BK curves of the form $y^2 = x^5 + b$, the performance of 4-dimensional scalar decompositions on curves of the form $C/\mathbf{F}_q\colon y^2 = x^5 + ax$ was also recently investigated [7]. Similar to the BK curves, the endomorphisms on these curves are very efficient in comparison to a group addition, so they facilitate significant speedups in Pollard rho. Here we have $m = 8$, so it would be interesting to see how close we can get to a $\sqrt{8}$ speedup in this case.

As is the case in the elliptic curve setting, there are several genus 2 families that possess maps which are useful to the cryptographer, but which offer no known benefit to the cryptanalyst – see [19] for some examples of endomorphisms which are not automorphisms. Thus, the application of Pollard rho to these families is identical to the case of Generic1271.

5 Performance Results

In order to systematically compare the security of the genus 1 and genus 2 curves in the previous section, we designed and implemented a software framework for 64-bit platforms supporting the `x64` instruction set. This modular design is capable of switching various features on or off: for example, using the automorphism optimization, employing different techniques for handling fruitless cycles, using different finite fields, or using different curve arithmetic. We implemented dedicated modular arithmetic for the special prime fields considered in this work (see Section 4); for each curve, we optimized the modular multiplication by hand in assembly, which resulted in a significant performance speedup compared to compiling our native C-code. All of the experimental results presented in this section have been obtained using an Intel Core i7-3520M (Ivy Bridge), running at 2893.484 MHz, and with the so-called *turbo boost* and *hyper-threading* features disabled.

We do not claim that the performance numbers reported in this section are the best possible. In a real attack, which focuses on a single curve target, the curve arithmetic and the arithmetic in the finite field should be optimized even further in assembly – we spent a moderate amount of time per curve to achieve good performance. We expect however, that the relative timings between the curves would remain roughly invariant under such further optimizations.

Table 2. Summary of the number of steps required when solving the DLP in a prime order subgroup n ($2^{N-1} < n < 2^N$) on the four (modified) curves we consider in this work. We computed 10 batches of 10^3 discrete logarithms and we display the minimum and maximum number of average steps out of these 10 batches, as well as the overall average. We used a 32-adding walk and a distinguished point property with $d = 8$, which we expect to occur once every 2^8 steps. The expected estimate is derived using Eq. (4).

curve	N	min	avg	max	expected
NIST CurveP-256	45	6 528 891	6 703 125	6 959 881	6 702 814
BN254	47	12 766 948	13 130 659	13 353 056	13 114 481
Generic1271	45	6 936 215	7 087 854	7 311 815	7 137 587
4GLV127-BK	45	5 339 249	5 489 583	5 668 256	5 489 249

5.1 Correctness

In order to make sure that our software framework works correctly and behaves as expected, we searched for curves defined over the same base fields as our target curves (as outlined in Section 4), but with smaller (around 45-bit) prime-order subgroups. We ran our implementations and enabled all the "statistic-gathering" options: this slows down the cost of a single step, but does not alter the behavior of the algorithm. We computed 10 batches of 10^3 Pollard rho computations for solving discrete logarithm instances in these subgroups, both with and without the use of the automorphism optimization.

Pollard rho without the group automorphism optimization. Assume we use an r-adding walk without the automorphism optimization (we take $m = 1$, where m is the cardinality of the group automorphism that is used). Experimental results from [34] suggest that using a larger r-value, such as $r \geq 16$, results in practical behavior that is closer to a truly random walk and gives a run-time that is close to the expected $\sqrt{\frac{\pi n}{2}}$. This is in agreement with the heuristic analysis from [2, Appendix B], which refines the arguments from [10], where it is shown that the average number of pseudo-random group elements required to find a collision (and solve the DLP) using an r-adding walk is

$$\sqrt{\frac{\pi n}{2m(1-\frac{1}{r})}}, \tag{3}$$

where n is the size of the prime order subgroup. We use the parallel (i.e. distinguished point) version of Pollard rho, such that approximately one out of every 2^d points is distinguished. When computing w walks concurrently, Eq. (3) can be adjusted to

$$\sqrt{\frac{\pi n}{2m(1-\frac{1}{r})}} + w \cdot 2^{d-1}. \tag{4}$$

This is because we need to perform an additional $w \cdot 2^{d-1}$ steps after two walks arrive at the same point: on average, 2^{d-1} steps are required to reach the next distinguished point, where both walks will be sent to the central database and the collision will be detected. For each scenario, Table 2 summarizes the average minimum, average and maximum steps of these 10 batches together with the theoretical number of steps we expect to take to solve the DLP. In all four cases, the average number of steps observed in practice matches the expected number of steps almost exactly: the difference is below one percent.

Table 3. A comparison of the expected (exp.) and real number of fruitless steps (**FS**) and fruitful steps when computing 10 batches of 10^3 discrete logarithms (as in Table 2) but using the group automorphism optimization. The genus-g curves have $m = \#\mathrm{Aut}(C)$ and we check for cycles up to length β every α steps.

	NIST CurveP-256	BN254	Generic1271	4GLV127-BK
(g, m, α, β)	$(1, 2, 7 \cdot 10^4, 32)$	$(1, 6, 6 \cdot 10^5, 32)$	$(2, 2, 7 \cdot 10^4, 32)$	$(2, 10, 10^6, 32)$
exp. # of fruitful steps (Eq.(4))	4 668 485	5 274 669	4 971 221	1 712 170
real # of fruitful steps (s)	4 643 787	5 271 219	5 010 354	1 723 756
exp. # of trapped **FS** (Eq. (5))	38 537	41 671	41 538	8185
real # of trapped **FS**	33 349	28 526	42 122	4835
exp. # of cycle reduction **FS**	4535	5148	4893	1683
real # of cycle reduction **FS**	4582	5173	4911	1687

Pollard rho with the group automorphism optimization. When using the group automorphism with $m = \#\mathrm{Aut}(C)$, we can encounter two types of fruitless steps: those due to the 2-cycle reduction technique and those which are performed when a walk is trapped in fruitless cycles. Due to the cycle reduction technique we use (see Section 3.3), the probability of 2-cycles and 3-cycles (if the latter can occur) have been reduced significantly. In fact, the probability to enter a 4-cycle becomes the most likely event by far, so we use the approximation $p = 1/(mr)^2$ (see Section 3.3) for the probability of entering any cycle. We check for cycles every α steps, where α depends on the curve (see Section 4), and we escape these cycles if necessary. If s is the expected number of steps required to solve the DLP, then the expected number of fruitless steps spent in fruitless cycles is

$$\frac{s}{\alpha} \cdot W(\alpha, (mr)^{-2}), \tag{5}$$

where W is as in Eq. (1).

Table 3 summarizes the results of running Pollard rho with the group automorphism optimization, where it is clear that the number of fruitful steps observed is very close to what we expect. Hence, we can expect to achieve a speedup if the practical cost of the iteration function is not increased too much. We note that the number of fruitless steps due to the 2-cycle reduction technique is also consistent with the prediction.

Interestingly, for the two curves with a larger automorphism group (i.e. with $m > 2$), the number of trapped fruitless cycles is lower than the expected value, which can be explained as follows. Since we expect fruitless cycles to occur much less frequently, the α parameter has been chosen significantly larger than for the curves with $m = 2$. In our benchmark runs, we solve the smaller DLP instances that are outlined in Table 2; if one of the walks gets trapped in a fruitless cycle, then, with overwhelming probability, one of the other concurrent walks will solve the DLP before this trapped walk has computed all of the fruitless $\alpha + \beta$ steps that are required to escape from this fruitless cycle. This behavior is not incorporated in our estimate for the total number of trapped fruitless steps. We ran larger instances of the DLP and, as expected, the total number of trapped fruitless steps increased.

5.2 Implementation Results

In order to optimize performance, we conducted several experiments to find the best parameters for instantiating the Pollard rho algorithm in practice: we varied the number of partitions in the adding walks (but restricted to $r \geq 1024$ when using the group automorphism optimization) and the number of concurrent walks. For all four curves, we found that 2048 concurrent

Table 4. The performance of our implementations expressed in the number of cycles without (32-adding walk) and with (1024-adding walk) the usage of the group automorphism running 2048 walks concurrently. For each curve, the expected speedup (which takes into account the additional cost of computing the equivalence class representative) and the speedup found in practice are stated together with the expected number of single-core years to solve a discrete logarithm. The security of each curve is given when taking NIST CurveP-256 as the baseline for the 128-bit security level.

Curve	Performance		speedup		core	sec
	without	with	exp.	real	years	
NIST CurveP-256	1129	1185	$\sqrt{2}$	$0.947\sqrt{2}$	$3.946 \cdot 10^{24}$	128.0
BN254	1030	1296	$\frac{6}{7} \cdot \sqrt{6} \approx 0.857\sqrt{6}$	$0.790\sqrt{6}$	$9.486 \cdot 10^{23}$	125.9
Generic1271	986	1043	$\sqrt{2}$	$0.940\sqrt{2}$	$1.736 \cdot 10^{24}$	126.8
4GLV127-BK	1398	1765	$\frac{120}{151} \cdot \sqrt{10} \approx 0.795\sqrt{10}$	$0.784\sqrt{10}$	$1.309 \cdot 10^{24}$	126.4

walks resulted in low costs for amortized inversions and gave the best performance. Using 2048 concurrent walks contradicts the advice from [9], which might be explained by the fact that our platform has a large cache so that "cache-misses" will only occur for a much larger number of concurrent walks. In regards to the optimal size of the lookup table, our benchmark runs showed that using 32-adding walks are best when the automorphism optimization is not used, and that 1024-adding walks are best when it is.

In Table 4 we state the performance numbers using the parameters above. We save computation by exploiting the fact that one does not need to update the a_i and b_i values [2]: this is especially significant for the curves with $m > 2$. Note that the number of computer cycles, when not using the group automorphism optimization, is lower for the BN254 curve compared to CurveP-256. This is surprising since the BN254 curve does not use a special prime. A partial explanation is that the CurveP-256 arithmetic is relatively slow, especially compared to the other NIST curves, and the addition of two residues might result in a carry occupying an additional word, which slows down the computation. On the other hand, the BN254 curve is defined over a 254-bit prime, such that subtraction-less Montgomery multiplication [37] can be used to save a conditional subtraction in every modular multiplication. Furthermore, the addition of two residues does not result in a carry occupying another word, which saves instructions. We suspect, however, that a hand-tweaked assembly implementation of NIST's CurveP-256 can be made slightly more efficient than the subtraction-less Montgomery arithmetic using the `x64` instruction set.

Table 4 states the expected speedup of Pollard rho using the automorphism (which takes into account the additional cost of choosing representatives), as well as the speedup we observed. This experimental speedup is consistently five to seven percent lower than the expected one, except for the 4GLV127-BK curve – such differences can be expected, as our analysis did not take extra modular additions, subtractions and negation into account, nor did we consider various overheads due to the usage of additional memory latencies. In the case of the BK curve, these additional factors constitute a much smaller fraction of the factors that were included in the analysis, which is why our experiments results match the expected numbers even closer.

For each curve, Table 4 also reports the expected number of single Intel Core i7-3520M core years required to solve a discrete logarithm instance. This estimate assumes that we use the group automorphism optimization and takes into account that we have to perform slightly more steps, increasing the estimate from Eq. (3) such that we take fruitless cycles into account, in line with the analysis from Section 4. Based on this estimate, we also give the security level

for each curve using the NIST CurveP-256 as the baseline for 128-bit security. Hence, this security estimate takes into account the different available optimizations for each curve, as well as the varying performance for the base field arithmetic.

6 Conclusions

We analyzed the practical security of elliptic curves and genus 2 hyperelliptic curves over prime fields using the Pollard rho algorithm. We developed a software framework implementing the state-of-the-art techniques to make use of the group automorphism optimization, which is targeted at 64-bit architectures that support the `x64` instruction set. We detailed how to optimize parameter selection when dealing with practical issues, such as reducing, detecting and escaping fruitless cycles; in particular, we analyzed these choices for curves with large automorphism groups, which have not yet received a detailed analysis in the literature.

We studied the performance of the Pollard rho algorithm on two elliptic curves and two genus 2 curves of cryptographic interest, all of which are estimated to provide around 128 bits of security. Our first conclusion is that, reassuringly, the practical security of all four curves considered is almost equivalent. Our second conclusion is that curves having large a large group automorphism of cardinality $m > 2$ can not achieve a speedup of $\sqrt{m}$: one has to pay a penalty for finding the representative of the equivalence class. Nevertheless, a constant-factor improvement is possible when dealing with fruitless cycles, and our analysis shows how to optimize this improvement in practice.

References

1. D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López. Faster explicit formulas for computing pairings over ordinary curves. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 48–68. Springer, 2011.
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. P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2005.
4. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, Heidelberg, 2006.
5. D. J. Bernstein. Elliptic vs. Hyperelliptic, part I. Talk at the ECC (slides at `http://cr.yp.to/talks/2006.09.20/slides.pdf`), September 2006.
6. D. J. Bernstein, T. Lange, and P. Schwabe. On the correct use of the negation map in the Pollard rho method. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography – PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 128–146. Springer, Heidelberg, 2011.
7. J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus 2. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 194–210. Springer, 2013.
8. J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012.
9. J. W. Bos, T. Kleinjung, and A. K. Lenstra. On the use of the negation map in the Pollard rho method. In G. Hanrot, F. Morain, and E. Thomé, editors, *Algorithmic Number Theory – ANTS-IX*, volume 6197 of *Lecture Notes in Computer Science*, pages 67–83. Springer, Heidelberg, 2010.

10. R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36(154):627–630, 1981.
11. J. Buhler and N. Koblitz. Lattice basis reduction, Jacobi sums and hyperelliptic cryptosystems. *Bulletin of the Australian Mathematical Society*, 58(1):147–154, 1998.
12. 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.
13. Certicom Research. Standards for efficient cryptography 1: Elliptic curve cryptography. Standard SEC1, Certicom, 2000.
14. C. Costello and K. Lauter. Group law computations on Jacobians of hyperelliptic curves. In A. Miri and S. Vaudenay, editors, *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 92–117. Springer, 2011.
15. I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *ASIACRYPT*, volume 1716 of *Lecture Notes in Computer Science*, pages 103–121. Springer, Heidelberg, 1999.
16. S. D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012.
17. S. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *J. Cryptology*, 24(3):446–469, 2011.
18. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.
19. P. Gaudry, D. R. Kohel, and B. A. Smith. Counting points on genus 2 curves with real multiplication. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 504–519. Springer, 2011.
20. P. Gaudry and É. Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012.
21. C. C. F. P. Geovandro, M. A. Simplício Jr., M. Naehrig, and P. S. L. M. Barreto. A family of implementation-friendly BN elliptic curves. *Journal of Systems and Software*, 84(8):1319–1326, 2011.
22. R. Harley. Elliptic curve discrete logarithms project. `http://pauillac.inria.fr/~harley/`.
23. B. Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, 31:1045–1062, 1960.
24. H. Hisil. *Elliptic curves, group law, and efficient computation*. PhD thesis, 2010.
25. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
26. N. Koblitz. Hyperelliptic cryptosystems. *Journal of cryptology*, 1(3):139–150, 1989.
27. T. Lange. Elliptic vs. Hyperelliptic, part II. Talk at the ECC (slides at `http://www.hyperelliptic.org/tanja/vortraege/ECC_06.ps`), September 2006.
28. P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In X. Wang and K. Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 718–739. Springer, 2012.
29. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, Heidelberg, 1986.
30. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
31. K. Nagao. Improving group law algorithms for Jacobians of hyperelliptic curves. In W. Bosma, editor, *ANTS*, volume 1838 of *Lecture Notes in Computer Science*, pages 439–448. Springer, 2000.
32. J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.
33. B. A. Smith. Families of fast elliptic curves from $\mathbb{Q}$-curves (to appear). In K. Sako and P. Sarkar, editors, *ASIACRYPT*, Lecture Notes in Computer Science. Springer, Heidelberg, 2013.
34. E. Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.
35. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4, 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.
36. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
37. C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.

38. M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography – (SAC) 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer New York, 1999.