# Delaying Mismatched Field Multiplications in Pairing Computations

Craig Costello, Colin Boyd, Juan Manuel Gonzalez Nieto, and Kenneth Koon-Ho Wong

Information Security Institute
Queensland University of Technology, GPO Box 2434,
Brisbane QLD 4001, Australia
{craig.costello,c.boyd,j.gonzaleznieto,kk.wong}@qut.edu.au

**Abstract.** Miller's algorithm for computing pairings involves performing multiplications between elements that belong to different finite fields. Namely, elements in the full extension field $\mathbb{F}_{p^k}$ are multiplied by elements contained in proper subfields $\mathbb{F}_{p^{k/d}}$, and by elements in the base field $\mathbb{F}_p$. We show that significant speedups in pairing computations can be achieved by delaying these "mismatched" multiplications for an optimal number of iterations. Importantly, we show that our technique can be easily integrated into traditional pairing algorithms; implementers can exploit the computational savings herein by applying only minor changes to existing pairing code.
**Keywords:** Pairings, Miller's algorithm, finite field arithmetic, Tate pairing, ate pairing.

## 1 Introduction

For the past decade, the public-key cryptographic community has witnessed an avalanche of novel and exciting protocols based on bilinear pairings; the major trigger being the discovery of identity-based encryption by Boneh and Franklin [8]. The algorithm for computing these pairings, initially proposed by Miller in the mid 1980's [27], was originally too slow for pairing-based protocols to be practically competitive with their RSA and Diffie-Hellman type rivals, and much research has since been invested towards speeding up Miller's algorithm. Consequently, Miller's algorithm has come a long way from its original form, to the point where many possible enhancements have now been fully optimized [2, 3, 29, 4, 31, 19].

The progress in the field of pairing improvements has seemingly steadied to a pace where a real world programmer could rest assured that their currently optimized (or close to optimized) implementation will most likely remain within arms length of the state-of-the-art implementation, for at least a couple of

* The first author acknowledges funding from the Queensland Government Smart State PhD Scholarship. This work has been supported in part by the Australian Research Council through Discovery Project DP0666065.

years. Nevertheless, so long as optimizations continue to be introduced [5, 12, 10], old pairing code could potentially become outdated quite quickly. The importance of code reusability and integrability might be the difference between an implementer continuing to modify and update their existing code in light of the latest breakthroughs, or shying away from such improvements because of the difficulty in integrating them.

It was shown very recently [10] that it is possible to avoid much of the costly, full extension field arithmetic encountered in pairing computations over large prime fields by replacing a multiplication in the full extension field with more minor multiplications in its proper subfields, decreasing the overall complexity of Miller's algorithm by over 30% in some cases. In these instances, an implementation not encompassing these techniques would perform substantially slower compared to one that does. However, a programmer wishing to implement the methods of avoiding extension field arithmetic in [10] would be facing the task of re-writing most (if not all) of their pairing code from scratch, having to employ new and potentially cumbersome explicit formulas.

In this paper we provide an alternative to the technique in [10] that offers much higher integrability into traditional pairing algorithms and existing pairing code. The idea used herein is the same as that used by Granger, Page and Stam [18, §6], who employ *loop unrolling* to combine two Miller iterations at a time, achieving fewer overall field operations in the case of characteristic three pairing implementations. In this paper we apply this same loop unrolling technique to pairings computed over large prime fields, by analyzing the cost of combining $n$ Miller iterations at a time, and choosing the optimal value of $n$ for various embedding degrees. Unlike the method in [10], our method requires no new explicit formulas for elliptic curve point operations and Miller line computations. Our aim is to inject a new and conceptually simple subroutine into Miller's algorithm that optimizes the field arithmetic occuring between elements of finite fields with different extension degree, with a parallel goal of minimizing the change imposed on existing pairing code.

The rest of this paper is organized as follows. In Section 2 we set notations and give a background on the computation of pairings. In Section 3 we discuss the proposed technique, before analyzing its computational complexity in Section 4. We provide necessary implementation details in Section 5, before providing parameters to optimize its implementation in Section 6, where we also draw comparisons against the traditional version of Miller's algorithm. In Appendix A, we provide MAGMA code that can be used as a basis to build implementations of our technique in other languages.

## 2 Preliminaries

Implementing pairings in cryptography most commonly requires the definition of two linearly independent groups, $\mathbb{G}_1$ and $\mathbb{G}_2$, of large prime order $r$, contained on an elliptic curve $E$ which is defined over a finite field $\mathbb{F}_q$ of large prime characteristic $p$. Herein, we choose to deal with the most common case of

prime fields, so that in fact we have $q = p$. Let $\pi_p$ be the $p$-power Frobenius endomorphism on $E$. In general, the most popular choices for $\mathbb{G}_1$ and $\mathbb{G}_2$ are the two eigenspaces of $\pi_p$, restricted to the $r$-torsion $E[r]$ of $E$, so that $\mathbb{G}_1 = E[r] \cap \ker(\pi_p - [1])$ and $\mathbb{G}_2 = E[r] \cap \ker(\pi_p - [p])$. Let $k$ be the smallest integer such that $r \mid p^k - 1$; a direct consequence of this is that the field $\mathbb{F}_{p^k}$ is the smallest extension of $\mathbb{F}_p$ that contains all of the points in $E[r]$, so that $\mathbb{F}_{p^k}$ houses both $\mathbb{G}_1$ and $\mathbb{G}_2$ in their entirity. We refer to $k$ the embedding degree, because computing the pairing of any two linearly independent points in $E[r]$ results in an element of an order-$r$ subgroup of the finite field $\mathbb{F}_{p^k}$, i.e. the pairing embeds the points of $E[r]$ into the $k$-degree extension of $\mathbb{F}_p$. We use $\mathbb{G}_T$ to denote this order-$r$ subgroup of $\mathbb{F}_{p^k}$, since this is the target group of the pairing map. For $k > 1$, the points in $\mathbb{G}_1$ are completely defined over the base field $\mathbb{F}_p$, whilst the points in $\mathbb{G}_2$ are defined over the larger field $\mathbb{F}_{p^k}$.

We assume that our pairing is defined by the Tate methodology rather than the Weil methodology (see [19]), since the Weil pairing has been phased out in practice due to its inefficient computation. The Tate methodology computes a bilinear pairing, $e$, of two linearly independent points $R, S \in E[r]$, as

$$e(R, S) = f_{m,R}(S)^{(p^k-1)/r}, \tag{1}$$

where $f_{m,R}$ is a function with divisor $\mathrm{div}(f_{m,R}) = m(R) - ([m]R) - (m-1)(\mathcal{O})$, with $\mathcal{O}$ being the neutral element on $E$. We refer to the function $f_{m,R}$ as the Miller function, since it is computed using Miller's algorithm. This algorithm uses relations between divisors of functions to build $f_{m,R}$ in $\log_2(m)$ iterations in a double-and-add like fashion, as summarized in Algorithm 1.

Pairings that fit into the Tate methodology can be naturally divided into two categories: *Miller-lite* pairings which take $R \in \mathbb{G}_1$ and $S \in \mathbb{G}_2$ and *Miller-full* pairings which take $R \in \mathbb{G}_2$ and $S \in \mathbb{G}_1$. That is, $e_{\text{lite}} : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$, whilst $e_{\text{full}} : \mathbb{G}_2 \times \mathbb{G}_1 \mapsto \mathbb{G}_T$. The Tate pairing and the twisted ate pairing [20] are examples of Miller-lite pairing, whilst the ate pairing [20] and its derivatives (the ate$_i$ pairing [26], the R-ate pairing [24], etc) sit under the umbrella of Miller-full pairings. Efficient pairing implementations make use of the twisted curve $E'$ to define a group $\mathbb{G}'_2 \in E'$ that is isomorphic to $\mathbb{G}_2 \in E$, but whose elements are contained in a much smaller subfield $\mathbb{F}_{p^e} \subset \mathbb{F}_{p^k}$, where $e = k/d$ and $d$ is the degree of the twist. We let $\psi : E' \to E$ denote the twisting isomorphism from $E'$ to $E$, so that $\psi(\mathbb{G}_2) = \mathbb{G}_1$. The bulk of the operations encountered in an iteration of Miller's algorithm are computed using the coordinates of $R$ or its image $R'$ under $\psi^{-1}$, so that Miller-lite pairings benefit from the majority of operations being performed over $\mathbb{G}_1$, which is defined over the base field $\mathbb{F}_p$. Alternatively, Miller-full pairings spend the majority of computations operating on coordinates that are defined over the larger extension field $\mathbb{F}_{p^e}$. Such computations are more costly over extension fields, however Miller-full pairings are usually more efficient than Miller-lite pairings in practice [20, 12], because they enjoy a much smaller loop parameter $m$, meaning that Miller's algorithm requires significantly less iterations.

**Algorithm 1** Miller's double-and-add Algorithm

**Input:** $R$, $S$, $m = (m_{l-1}...m_1, m_0)_2$.
**Output:** $f_{m,R}(S) \leftarrow f$.

1: $T \leftarrow R$, $f \leftarrow 1$.
2: **for** $i = l - 2$ to $0$ **do**
3: $T \leftarrow [2]T$.
4: Compute a function $g$, which has divisor $\text{div}(g) = 2(T) - (2T) - (\mathcal{O})$.
5: Compute $g = g(S)$ (evaluate $g$ at the coordinates of $S$).
6: $f \leftarrow f^2 \cdot g$.
7: **if** $m_i \neq 0$ **then**
8: $T \leftarrow T + R$.
9: Compute a function $g$, which has divisor $\text{div}(g) = (T) + (R) - (T + R) - (\mathcal{O})$.
10: Compute $g = g(S)$ (evaluate $g$ at the coordinates of $S$).
11: $f \leftarrow f \cdot g$.
12: **end if**
13: **end for**
14: **return** $f$.

Any extension fields of $\mathbb{F}_p$ that are required in the pairing computation are best constructed using towers of field extensions. The general method to construct towers of extension fields in pairing-based cryptography is originally due to Koblitz and Menezes [23], who introduced the notion of *pairing-friendly fields*, where the embedding degree is chosen to be of the form $k = 2^i 3^j$, and the characteristic of the field $\mathbb{F}_p$ is chosen to be $p \equiv 1 \bmod 12$. These conditions allow us to easily build a tower of extensions up to $\mathbb{F}_{p^k}$ using a sequence of $z = i + j$ cubic and quadratic sub-extensions, where the defining polynomial for each of the $d_{i,j}$-degree sub-extensions are actually binomial of the form $x^{d_{i,j}} - \alpha$. Such quadratic and cubic binomials facilitate fast arithmetic over extension fields. Very recently, Benger and Scott [5] broadened the definition of pairing-friendly fields to present the more general notion of *towering-friendly fields*, which are fields of the form $\mathbb{F}_{q^m}$ ($q$ not necessarily prime itself) for which all prime divisors of $m$ also divide $q - 1$, showing that efficient tower constructions can also be achieved without satisfying the more restrictive condition of $p \equiv 1 \bmod 12$ for characteristic $p$ fields.

For elliptic curves, there are only four twist degrees possible: $d = 2$ quadratic twists, $d = 3$ cubic twists, $d = 4$ quartic twists and $d = 6$ sextic twists. In both Miller-lite and Miller-full pairings, it is advantageous to choose the Weierstrass curve model (of the form $y^2 = x^3 + ax + b$) which supports the maximal twist degree $d$, such that $d \mid k$. Cubic and sextic twists are only possible when $a = 0$, quartic twists when $b = 0$, and quadratic twists impose no condition on the curve constants, although it is usually advantageous to set either $a$ or $b$ to be zero for computational efficiency anyway [1, 12].

For quadratic and cubic twists, $\mathbb{F}_{p^k}$ is the direct (quadratic or cubic) sub-extension of the field $\mathbb{F}_{p^e}$. For quartic and sextic extensions, however, we must

first extend $\mathbb{F}_{p^e}$ to an intermediate field $\mathbb{F}_{p^h}$, where $\mathbb{F}_{p^e} \subset \mathbb{F}_{p^h} \subset \mathbb{F}_{p^k}$, and the field extensions are formed by taking $\mathbb{F}_{p^h} = \mathbb{F}_{p^e}(\alpha)$ and $\mathbb{F}_{p^k} = \mathbb{F}_{p^h}(\beta)$. We denote the degree of the extensions as $\delta_\alpha = [\mathbb{F}_{p^h} : \mathbb{F}_{p^e}] = h/e$ and $\delta_\beta = [\mathbb{F}_{p^k} : \mathbb{F}_{p^h}] = k/h$, in agreement with $[\mathbb{F}_{p^k} : \mathbb{F}_{p^e}] = \delta_\alpha \delta_\beta = k/e = d$. For all twists, we have that an element of the full extension field, say the Miller function $f \in \mathbb{F}_{p^k}$, takes the form

$$f = \sum_{j=0}^{\delta_\beta - 1} \Big( \sum_{i=0}^{\delta_\alpha - 1} f_{j,i} \cdot \alpha^i \Big) \cdot \beta^j, \tag{2}$$

where each of the $f_{j,i}$ are contained in $\mathbb{F}_{p^e}$. For quadratic twists we must take $(\delta_\alpha, \delta_\beta) = (2, 1)$ and for cubic twists we must take $(\delta_\alpha, \delta_\beta) = (3, 1)$. For both quartic and sextic twists, Benger and Scott [5] suggest that the most efficient tower is constructed with $\delta_\alpha = 2$, so that quartic twists should take $(\delta_\alpha, \delta_\beta) = (2, 2)$, and sextic twists should take $(\delta_\alpha, \delta_\beta) = (2, 3)$. The nature of the tower for the fields that lie between $\mathbb{F}_p$ and $\mathbb{F}_{p^e}$ do not play a role in this work, so we pay no attention to these details, but point the interested reader to [5].

The general twist of a short Weierstrass curve is written as $E'(\mathbb{F}_{p^e}) : y^2 = x^3 + az^4x + bz^6$, where the isomorphism $\psi : E' \to E$ is defined as $\psi(x', y') = (z^2x', z^3y')$. For quartic twists when $b = 0$, we choose $z^4 \in \mathbb{F}_{p^e}$ such that $z^2 \in \mathbb{F}_{p^{k/2}} \notin \mathbb{F}_{p^e}$ and $z \in \mathbb{F}_{p^k} \notin \mathbb{F}_{p^{k/2}}$, so that we can set $\alpha = z^2$ and $\beta = z^3$, resulting in a twisting isomorphism $\psi(x', y') = (\alpha x', \beta y')$ that allows twisted coordinates to be easily integrated with general field elements taking the form of $f$ in (2). Similarly, for sextic twists when $a = 0$, we choose $z^6 \in \mathbb{F}_{p^e}$ such that $z^3 \in \mathbb{F}_{p^{k/3}} \notin \mathbb{F}_{p^e}$ and $z^2 \in \mathbb{F}_{p^{k/2}} \notin \mathbb{F}_{p^e}$, so that we can set $\alpha = z^3$ and $\beta = z^2$, and the twisting isomorphism conveniently becomes $\psi(x', y') = (\beta x', \alpha y')$.

We follow the general trend of reporting results for even $k$ [3, 23, 5], since such embedding degrees support the denominator elimination optimization [2]. Thus, any $k$ we consider which is divisible by 3 will also be divisible by 6 and admit a sextic twist, so that we do not need to consider cubic twists. Computationally speaking, the treatment of cubic twists is quite different to the other even degree twists and tends to be awkward anyway [12], so curves with odd embedding degree divisible by 3 are generally not chosen in practice, although Lin *et al.* [25] show that choosing $k = 9$ can be competitive at some security levels.

*Remark 1 (A notation for counting costs).* This paper is largely concerned with the computational cost of field operations, so we employ a notation that allows us to easily narrate such costs alongside the associated algebra. We use $\texttt{cost}\big[L \leftarrow J\big]$ to denote the computational cost of computing the set $L = \{L_1, ..., L_i\}$ from the already computed (or available) set $J = \{J_1, ..., J_j\}$. If the best way to compute the set $L$ from the set $J$ is to compute the intermediate set $K = \{K_1, ..., K_j\}$, then we can clearly split the cost, so that $\texttt{cost}\big[L \leftarrow J\big] = \texttt{cost}\big[L \leftarrow K\big] + \texttt{cost}\big[K \leftarrow J\big]$, under the assumption that there does not exist a cheaper way to compute $L$ from $J$ which does not require the computation of $K$. When referring to the cost of computing the set $L$ without assuming any prior computations, we simply use $\texttt{cost}\big[L\big]$.

*Remark 2 (The squaring vs. multiplication ratio).* Our cost analyses are primarily concerned with field multiplications and field squarings and we choose not pay any attention to the much cheaper cost of field additions, although the algorithms presented herein aim to minimize all field operations. We use $\mathbf{m}_i$ and $\mathbf{s}_i$ to represent the respective costs of a multiplication and a squaring in the field $\mathbb{F}_{p^i}$. Since the ratio of the complexity of a field squaring to a field multiplication is specific to the implementation, we leave the discussion general until Section 6 by using the parameter $\Omega$, which denotes the $\mathbf{s} : \mathbf{m}$ ratio. That is, $\mathbf{s} = \Omega\mathbf{m}$, where $0 << \Omega \leq 1$. For example, Bernstein [6] achieves $\Omega = 0.68$ and Hisil [21] reports $\Omega = 0.72$, while the EFD [7] presents results based on the more commonly accepted $\Omega = 0.8$ and $\Omega = 1$ values.

*Remark 3 (Non-specific field definitions).* In the pairing $e(R, S)$, the respective fields that $R$ and $S$ belong to are different depending on whether the pairing is a Miller-lite or Miller-full pairing. In a Miller-lite pairing computed as $e(R, \psi(S'))$, we take $R \in \mathbb{F}_p$ and $S' \in \mathbb{F}_{p^e}$, whilst a Miller-full pairing computed as $e(\psi^{-1}(R), \psi^{-1}(S))$ (see [12]) has $R' \in \mathbb{F}_{p^e}$ and $S \in \mathbb{F}_p$. Ignoring the twisting elements $\alpha$ and $\beta$, then the first and second arguments in a Miller-lite pairing are from $\mathbb{F}_p$ and $\mathbb{F}_{p^e}$ respectively, whilst the same arguments in a Miller-full pairing are from $\mathbb{F}_{p^e}$ and $\mathbb{F}_p$ respectively. In sections 3 and 4, we cover both cases simultaneously by saying that the first argument $R$ belongs to $\mathbb{F}_{p^u}$ and the second argument $S$ belongs to $\mathbb{F}_{p^v}$, where it is understood that $(u, v) = (1, e)$ for Miller-lite pairings and $(u, v) = (e, 1)$ for Miller-full pairings. Most importantly, in either case we have that multiplying an element of $\mathbb{F}_{p^u}$ by an element of $\mathbb{F}_{p^v}$ costs $e\mathbf{m}_1$ (cf. [12]).

*Remark 4 (Ignoring additions).* As is the common trend in papers discussing optimal pairing implementations, we assume that the loop parameter $m$ has low Hamming weight so that additions are sparse in Miller's algorithm. Thus, when discussing any consecutive iterations of the Miller loop, we assume that no such iterations involve additions.

## 3 Delaying Mismatched Multiplications

We begin this section by illustrating the potential advantage of delaying "mismatched" multiplications, through the use of a toy example. Suppose we have a basic algorithm that involves $n$ iterations, where the $i$-th iteration simply involves computing an element $a_i$, and updating the master function $A_i$ as $A_i \leftarrow a_i \cdot A_{i-1}$, where the master function was initialized as $A_0$. The output of the algorithm would be $A_n = (((...((A_0 \cdot a_1) \cdot a_2)...)))$, which could alternatively be written, or indeed computed as $A_n = A_0 \cdot (\prod_{i=1}^{n} a_i)$, where the product of the $a_i$'s is computed prior to multiplication with $A_0$. If both $A_0$ and the $a_i$'s are general elements of the same field, say $\mathbb{F}_{p^c}$, then both methods of computation would require $n$ multiplications in $\mathbb{F}_{p^c}$, as

$$\texttt{cost}\big[A_n \leftarrow \{a_1, ..., a_n, A_0\}\big] = \sum_{i=1}^{n} \texttt{cost}\big[A_i \leftarrow \{A_{i-1}, a_i\}\big] = \sum_{i=1}^{n} 1\mathbf{m}_c = n\mathbf{m}_c,$$

or

$$\texttt{cost}\big[A_n \leftarrow \{a_1, ...,a_n, A_0\}\big] = \texttt{cost}\big[A_n \leftarrow \{\prod_{i=1}^{n} a_i, A_0\}\big] + \texttt{cost}\big[\prod_{i=1}^{n} a_i \leftarrow \{a_1, ..., a_n\}\big] = 1\mathbf{m}_c + (n-1)\mathbf{m}_c = n\mathbf{m}_c.$$

However, suppose again that the $a_i$ values are general elements of the field $\mathbb{F}_{p^c}$, but instead suppose that $A_0$ is a general element of the degree-$w$ extension field $\mathbb{F}_{p^{cw}}$, of $\mathbb{F}_{p^c}$. For ease of exposition, we assume for now that $w$ is prime so that a general element of $\mathbb{F}_{p^{cw}}$ can be expressed as a $(w-1)$-degree polynomial with coefficients in $\mathbb{F}_{p^c}$. In this case, multiplying each of the $a_i$ values by $A_0$ would typically involve multiplying each of the $w$ coefficients of $A_0$ by $a_i$, costing $w\mathbf{m}_c$ each time. Clearly, it would be advantageous to form the product $\prod_{i=1}^{n} a_i$ prior to a multiplication by $A_0$, as we show by using the same comparison as before, where

$$\texttt{cost}\big[A_n \leftarrow \{a_1, ..., a_n, A_0\}\big] = \sum_{i=1}^{n} \texttt{cost}\big[A_i \leftarrow \{A_{i-1}, a_i\}\big] = \sum_{i=1}^{n} w\mathbf{m}_c = wn\mathbf{m}_c,$$

whilst

$$\texttt{cost}\big[A_n \leftarrow \{a_1, ..., a_n, A_0\}\big] = \texttt{cost}\big[A_n \leftarrow \{\prod_{i=1}^{n} a_i, A_0\}\big] + \texttt{cost}\big[\prod_{i=1}^{n} a_i \leftarrow \{a_1, ..., a_n\}\big] = w\mathbf{m}_c + (n-1)\mathbf{m}_c = (w+n-1)\mathbf{m}_c.$$

Forming the product of the $a_i$ elements from the smaller field prior to the multiplication by $A_0$ gives a count of $(w+n-1)\mathbf{m}_c$, as opposed to the $wn\mathbf{m}_c$ that it costs to multiply $a_i$ by $A_0$ in each iteration. When $n > 1$ and $w > 1$, it is always the case that $wn > (w+n-1)$, so that forming the product of $n > 1$ elements in the smaller field and *delaying* any multiplications by the element in the larger field is always advantageous. The central theme of this paper is applying this idea towards pairing computations, however the story in Miller's algorithm is more complicated than the example above. Firstly, the "mismatched" multiplications above were easy to spot, since we were multiplying general elements from different fields. However, there are other more subtle examples of mismatched multiplications, which we formalize in the following definitions.

**Definition 1 (General vs. special field elements).** *Let $\omega \in \mathbb{F}_{p^c}$, where $\mathbb{F}_{p^c}$ is constructed as a tower of extensions as $\mathbb{F}_p \subset \mathbb{F}_{p^{c_1}} \subset \mathbb{F}_{p^{c_1 c_2}} \subset ... \subset \mathbb{F}_{p^{c_1 c_2 \cdots c_t}} = \mathbb{F}_{p^c}$, i.e. $c_i$ is the degree of the $i$-th extension in the tower up to $\mathbb{F}_{p^c}$. Let $C_j = \prod_{i=1}^{j} c_i$ so that we can write the tower as $\mathbb{F}_p \subset \mathbb{F}_{p^{C_1}} \subset \mathbb{F}_{p^{C_2}} \subset ... \subset \mathbb{F}_{p^{C_t}} = \mathbb{F}_{p^c}$. Let $\#\omega(\mathbb{F}_{p^{C_j}})$ be the number of non-zero coefficients in the*

*polynomial representation of $\omega$ over the subfield $\mathbb{F}_{p^{C_j}}$. If $\#\omega(\mathbb{F}_{p^{C_j}}) < c/C_j$ for any $j$ where $1 \leq j \leq t$, then we call $\omega$ a special element of $\mathbb{F}_{p^c}$, otherwise we call $\omega$ a general element of $\mathbb{F}_{p^c}$.*

**Definition 2 (Mismatched multiplications).** *Let $\omega \in \mathbb{F}_{p^c}$ and $\hat{\omega} \in \mathbb{F}_{p^{\hat{c}}}$. We call the multiplication between $\omega$ and $\hat{\omega}$ mismatched if one of the following two conditions hold:*

$(i)$ $c \neq \hat{c}$.

$(ii)$ $c = \hat{c}$, *but at least one of $\omega$ and $\hat{\omega}$ are special.*

*We refer to a mismatched multiplication as a type (i) or type (ii) mismatch, depending on which of the above conditions it breaches.*

Equipped with the above definitions, we now focus on searching for mismatched multiplications in Miller's algorithm, with the aim of investigating the possibility and potential advantage of optimizing the delay or the avoidance of such multiplications. We start by taking a close look at the doubling stage of Miller's algorithm which is the combination of steps 3, 4, 5 and 6 of Algorithm 1. Steps 3 and 4 involve doubling the point $T$ (i.e. computing $[2]T$ from $T$), and computing the coefficients of the associated function $g$ with divisor $\mathrm{div}(g) = 2(T) - (2T) - (\mathcal{O})$. These computations only depend on the coordinates of the point $T = (T_x, T_y) \in \mathbb{F}_{p^u}$, and since $T_x$ and $T_y$ are assumed to be general elements of $\mathbb{F}_{p^u}$, we can safely assume that, in general, none of the field multiplications in steps 3 and 4 are mismatched. Many authors have achieved speed ups in pairing computations by focussing on reducing the combined cost of these two steps [9, 13, 22, 1, 11, 12], where the cost of encapsulated point doubling (step 3) and line computation (step 4) is generally presented together, in terms of the combined number of field multiplications (m) and squarings (s) encountered, as

$$\mathtt{cost}\big[\{g, [2]T\} \leftarrow T\big] = m\mathbf{m}_u + s\mathbf{s}_u = (m + \Omega s)\mathbf{m}_u. \tag{3}$$

For curves with even embedding degrees, the denominator elimination optimization greatly simplifies the form of the line function $g$, so that $g = g(x, y)$ always (cf. [12]) takes the form

$$g(x, y) = g_x \cdot x + g_y \cdot y + g_0, \tag{4}$$

where $g_x, g_y, g_0 \in \mathbb{F}_{p^u}$. Step 5 of Algorithm 1 involves evaluating $g$ at the coordinates of $S = (S_x, S_y)$, i.e. multiplying $g_x$ by $S_x$ and $g_y$ by $S_y$. From Section 2, we know that (unless $e = 1$) $R$ and $S$ are contained in different fields, so that the evaluation of $g$ at $S$ incurs two type (i) mismatched multiplications. Following this, step 6 of Algorithm 1 involves squaring the Miller function $f \in \mathbb{F}_{p^k}$, and multiplying this result by $g(S) \in \mathbb{F}_{p^k}$. Although the point $S$ belongs to the field $\mathbb{F}_{p^k}$, each of its coordinates are actually either very special elements of $\mathbb{F}_{p^k}$, or lie in proper subfields of $\mathbb{F}_{p^k}$. For example, in Section 2 we saw

that an implementation employing a quartic twist has $(S_x, S_y) = (\alpha\hat{S}_x, \beta\hat{S}_y)$, where $\hat{S}_x, \hat{S}_y \in \mathbb{F}_{p^v}$, or similarly an implementation using a sextic twist has $(S_x, S_y) = (\beta\hat{S}_x, \alpha\hat{S}_y)$ with $\hat{S}_x, \hat{S}_y \in \mathbb{F}_{p^v}$. In both cases, it is clear by Definition 1 that $g(S)$ is a special element of $\mathbb{F}_{p^k}$, so that the multiplication of the Miller function $f$ by $g(S)$ is, by Definition 2, a type (ii) mismatched multiplication.

We concretize the above discussion with an example, where we assume a sextic twist has been used, so that lines 5 and 6 of Algorithm 1 require that we compute a multiplication between

$$f = (f_{2,1} \cdot \alpha + f_{2,0}) \cdot \beta^2 + (f_{1,1} \cdot \alpha + f_{1,0}) \cdot \beta + (f_{0,1} \cdot \alpha + f_{0,0}) \in \mathbb{F}_{p^k}$$

and

$$g(S_x, S_y) = (g_x\hat{S}_x) \cdot \beta + (g_y\hat{S}_y) \cdot \alpha + g_0 \in \mathbb{F}_{p^k},$$

where the $f_{i,j}$'s and both $g_x\hat{S}_x$ and $g_y\hat{S}_y$ are contained in $\mathbb{F}_{p^e}$ (see Remark 3), and $g_0$ is contained in $\mathbb{F}_p$ for Miller-lite implementations or $\mathbb{F}_{p^e}$ in Miller full implementations. Since $g_x, g_y \in \mathbb{F}_{p^u}$ and $\hat{S}_x, \hat{S}_y \in \mathbb{F}_{p^v}$, the products formed to create $g(S_x, S_y)$ are type (i) mismatches, whilst the multiplication between $f$ and $g(S)$ is a type (ii) mismatch.

There are two natural questions that now arise: *are these mismatches a problem?* and, if so, *what can we do about them?* We can immediately answer the first question by referring back to the toy example at the beginning of this section, where we saw that delaying the multiplications between elements of different sized fields can be very advantageous, particularly if the difference in the extension degrees is large.

We start the answer to the second question by noting the main complication, in terms of mismatched multiplications, that we encounter in an iteration of Miller's algorithm; that being the simultaneous presence of both type (i) and type (ii) mismatches. Specifically, each iteration of Miller's algorithm involves two type (i) mismatches buried inside a larger type (ii) mismatch. An ideal solution might involve minimizing both mismatches simultaneously, but unfortunately we will soon see that this is not possible; namely, that the type (i) mismatches are somewhat unavoidable in Miller's algorithm. However, the solution we adopt follows quite naturally if we start by *trying* to avoid the type (i) mismatches, as follows. In a particular iteration of Miller's algorithm, it seems that the only way we can avoid the type (i) mismatched multiplications is to delay them until the following iteration: let $g$ and $\tilde{g}$ represent two consecutive $g$'s in two iterations of Miller's algorithm, and suppose we temporarily delay evaluating $g$ as $S$ until the following iteration when $\tilde{g}$ is computed. Instead of evaluating both functions separately at $S$, we form the product two indeterminate functions, $g(x, y)$ and $\tilde{g}(x, y)$, modulo the curve equation, and call it $G(x, y)$. In fact, $g$ would have been multiplied by $f$ and squared in the previous iteration, so that $G(x, y)$ is

actually computed as

$$G(x,y) = g(x,y)^2 \cdot \tilde{g}(x,y) = (g_x \cdot x + g_y \cdot y + g_0)^2 \cdot (\tilde{g}_x \cdot x + \tilde{g}_y \cdot y + \tilde{g}_0)$$
$$= \sum_{i=0}^{4} G_{x^i} \cdot x^i + \big[\sum_{i=0}^{3} G_{x^i y} \cdot x^i\big] \cdot y, \tag{5}$$

where we reduce any higher powers of $y$ via the curve equation. We could then evaluate $G(x,y)$ at $S$ and multiply $G(S)$ by the Miller function $f$, so that delaying the evaluation of $g$ at $S$ and the multiplication of $f$ by $g(S)$ avoided both types of mismatched multiplications for one iteration. However, at this next iteration, we now have many more type (i) multiplications to deal with. Namely, what would have been 4 type (i) mismatches in total (2 for the evaluation of $g$ at $S$ and likewise for the evaluation of $\hat{g}$ at $S$), has now become 8 type (i) mismatches (multiplying $G_{x^i y}$ by $x^i y$ and $G_{x^i}$ by $x^i$ above). At a first glance then, this idea seems somewhat counterproductive. However, let us assume for now that we are employing a sextic twist so that $(\delta_\alpha, \delta_\beta) = (2,3)$ and observe the new function $G$ evaluated at $S = (\hat{S_x}\beta, \hat{S_y}\alpha)$, as

$$G(x,y) = \sum_{i=0}^{4} G_{x^i} \cdot S_x^i \cdot \beta^i + \big[\sum_{i=0}^{3} G_{x^i y} \cdot S_x^i \cdot \beta^i\big] \cdot \alpha = \sum_{j=0}^{\delta_\beta} \Big( \sum_{i=0}^{\delta_\alpha} \hat{G}_{j,i} \cdot \alpha^i \Big) \cdot \beta^j, \tag{6}$$

where each of the $\hat{G}_{j,i}$ are easily derived combinations of the $G_{x^i y}$ and $G_{x^i}$ terms in (5). Importantly, we now have that $G(x,y)$ has become a general element of $\mathbb{F}_{p^k}$, so that performing the multiplication between $f$ and $G$ will fully exploit a routine written to perform optimized multiplication over $\mathbb{F}_{p^k}$. More importantly, we have only had to perform one full extension field multiplication in two Miller iterations. In short, we delayed the multiplication between $f$ and $g$ until $g$ was built up into $G$ (a product of $g$'s), a general element of $\mathbb{F}_{p^k}$, and in doing so we saved a mismatched multiplication in $\mathbb{F}_{p^k}$. The price we pay for this saving is the increased number of type (i) mismatched multiplications that are required to evaluate $G$ at $S$, as well as an increased number of standard $\mathbb{F}_{p^u}$ multiplications that are required to form the coefficients of $G$ from $g$ and $\hat{g}$. Our goal becomes clear then; we wish to explore whether it is advantageous to spend extra computations in order to achieve the savings offered by avoiding type (ii) mismatches altogether.

In the following sections, we explore these trade-offs in detail. Specifically, we consider delaying the multiplication between the $g$'s and the Miller function $f$ for an arbitrary number ($N$) of iterations, a process we refer to as $N$-delay. We track the computational cost of $N$-delay and determine the optimum values of $N$ for implementations over a variety of embedding degrees. Before moving to the next section, we make the following remarks.

*Remark 5.* Since we are forced to accept the presence of type (i) mismatches in pairings, one possible solution to the problem described above would be to write

a specialized multiplication routine for the type (ii) mismatched multiplication between the general element $f$ and special element $g$, of $\mathbb{F}_{p^k}$. However, replacing the full $\mathbb{F}_{p^k}$ multiplication routine (that takes two general field elements as inputs) with such a specialized routine means that, to some extent, we are sacrificing the tricks that speed up general multiplications, such as the Karatsuba and Toom-Cook methods. Such optimizations are the reason we build extension fields up as towers of degree 2 and 3 sub-extensions, so we argue that avoiding these optimizations is potentially counterproductive, instead favoring the $N$-delay techniques herein.

*Remark 6.* The discussion in this section (and in the next) essentially describes the technique of *loop unrolling*, which was first introduced into pairing computations by Granger *et al.* [18], who merged iterations to exploit the sparsity of $g$. Speedups were achieved in [18] by combining two consecutive iterations into one merged iteration, in implementations over fields of characteristic three. This technique was later used by Shirase *et al.* [30] in pairing implementations over binary fields. To the best of the authors knowledge, this paper is the first to describe a general algorithm for loop unrolling which merges any number of iterations over large prime fields.

## 4 The Cost of $N$-delay

We let $N$-delay refer to the process of delaying the multiplication of the Miller function $f$ by consecutive function updates $g$, $N$ times in a row. We make note that $N = 0$ corresponds to the standard Miller routine which delays zero multiplications between $f$ and $g$, whilst $N = 1$ corresponds to the Miller routine which delays one multiplication (combines two iterations), and so on, so that in general $N$ can be thought of as the number of times a multiplication by $f$ is delayed, whilst $N + 1$ is the number of iterations that are combined. The aim of this section is to obtain an expression for the computational cost of $N$-delay, in terms of $N$, so that we can determine the optimal $N$ value for specific implementations. To do this, we determine the cost of delaying a single, but general iteration. That is, we write a general expression for the product of the $n$ different powers of $g$'s accumulated after $n$ iterations, and use this to determine the cost of updating this to the $n + 1$-th product. We then sum this cost from $n = 0$ to $n = N - 1$ to obtain the entire cost of performing $N$-delay.

We let $G_n(x, y)$ be the cumulative product of the first $n$ indeterminate $g(x, y)$ functions, reduced modulo the curve equation, as

$$G_n(x, y) = \sum_{i=0}^{A_n} a_i \cdot x^i \cdot y + \sum_{i=0}^{B_n} b_i \cdot x^i = G_{n_a}(x) \cdot y + G_{n_b}(x), \tag{7}$$

where $G_{n_a}(x) = \sum_{i=0}^{A_n} a_i \cdot x^i$ and $G_{n_b}(x) = \sum_{i=0}^{B_n} b_i \cdot x^i$. Similarly, we define the $n$-th Miller function update from (4) as

$$g_n(x, y) = g_{n_x} \cdot x + g_{n_y} \cdot y + g_{n_0} = g_{n_a} \cdot y + g_{n_b}(x), \tag{8}$$

where $g_{n_a} = g_{n_y}$ and $g_{n_b}(x) = g_{n_x} \cdot x + g_{n_0}$. The $(n+1)$-th consecutive Miller iteration would multiply the square of $G_n(x, y)$ by the $(n+1)$-th Miller function update, $g_{n+1}(x, y)$, as

$$\begin{aligned} G_{n+1}(x,y) =& G_n^2(x,y) \cdot g_{n+1}(x,y) \quad = \quad \big(G_{n_a}(x) \cdot y + G_{n_b}(x)\big)^2 \cdot g_{n+1}(x,y) \\ =& \Big(G_{n_a}(x)^2 C(x) + 2G_{n_a}(x)G_{n_b}(x) \cdot y + G_{n_b}(x)^2\Big) \cdot g_{n+1}(x,y) \\ =& \Big(g_{n+1_a} h_1(x) + g_{n+1_b}(x) h_2(x)\Big) \cdot y + \big(g_{n+1_b}(x) h_1(x) + g_{n+1_a} h_3(x)\big) \\ =& \sum_{i=0}^{A_{(n+1)}} \hat{a}_i \cdot x^i \cdot y + \sum_{i=0}^{B_{(n+1)}} \hat{b}_i \cdot x^i = G_{n+1_a}(x) \cdot y + G_{n+1_b}(x), \end{aligned} \tag{9}$$

where $h_1(x) = G_{n_a}(x)^2 C(x) + G_{n_b}(x)^2$ , $h_2(x) = 2G_{n_a}(x)G_{n_b}(x)$, $h_3(x) = 2G_{n_a}(x)G_{n_b}(x)C(x)$, and $y^2$ was replaced with $C(x) = x^3 + ax + b$. Paying close attention to (9) allows us to determine the cost of obtaining $G_{n+1}$ from $G_n$. We make the following observations.

- **Observation 1.** To determine the values of $A_{n+1}$ and $B_{n+1}$, (9) reveals that

$$\begin{aligned} A_{n+1} &= \text{Max}\{\deg(g_{n+1_a}) + \deg(h_1), \deg(g_{n+1_b}) + \deg(h_2)\} \\ &= \text{Max}\{2A_n + 3, 2B_n, A_n + B_n + 1\}, \end{aligned}$$

  and similarly

$$\begin{aligned} B_{n+1} &= \text{Max}\{\deg(g_{n+1_b}) + \deg(h_1), \deg(g_{n+1_a}) + \deg(h_3)\} \\ &= \text{Max}\{2A_n + 4, 2B_n + 1, A_n + B_n + 3\}. \end{aligned}$$

  Since $(A_0, B_0) = (0, 1)$, we always have that $(A_{n+1}, B_{n+1}) = (2A_n + 3, 2A_n + 4)$, from which it follows that

$$(A_n, B_n) = (3(2^n - 1), 3(2^n - 1) + 1). \tag{10}$$

- **Observation 2.** The three necessary terms $G_{n_a}^2 = (\sum_{i=0}^{A_n} a_i \cdot x^i)^2$, $G_{n_b}^2 = (\sum_{i=0}^{B_n} b_i \cdot x^i)^2$ and $2G_{n_a}G_{n_b} = 2(\sum_{i=0}^{A_n} a_i \cdot x^i)(\sum_{i=0}^{B_n} b_i \cdot x^i)$ can be computed using only field squarings as follows. Each of the $a_i^2$ terms in $G_{n_a}^2$ can be computed first and used to compute (via a squaring) the remaining terms of the form $2a_i a_j$ in $G_{n_a}^2$, where $i \neq j$. In total, there are $\sum_{i=0}^{A_n} \sum_{j=0}^{i} = (A_n + 1)(A_n + 2)/2$ different $a_i a_j$ combinations contributing to $G_{n_a}^2$, so that $\texttt{cost}\big[G_{n_a}^2 \leftarrow G_{n_a}\big] = [(A_n + 1)(A_n + 2)/2]\mathbf{s}_u$. Identically, we have that $\texttt{cost}\big[G_{n_b}^2 \leftarrow G_{n_b}\big] = [(B_n + 1)(B_n + 2)/2]\mathbf{s}_u$. Lastly, each of the terms of the form $2a_i b_j$ in $2G_{n_a}G_{n_b}$ can be computed at the cost of a squaring using the previously computed $a_i^2$ and $b_j^2$ values. There are $(A_n + 1)(B_n + 1)$ such terms contributing to $2G_{n_a}G_{n_b}$, so that $\texttt{cost}\big[2G_{n_a}G_{n_b} \leftarrow \{G_{n_a}^2, G_{n_b}^2\}\big] =$

$(A_n+1)(B_n+1)\mathbf{s}_u$. Importantly, we use (10) to give

$$\begin{aligned}\texttt{cost}\big[\{G^2_{n_a}, 2G_{n_a}G_{n_b}, G^2_{n_b}\} \leftarrow \{G_{n_a}, G_{n_b}\}\big] &= \texttt{cost}\big[G^2_{n_a} \leftarrow G_{n_a}\big] \\ &\quad + \texttt{cost}\big[G^2_{n_b} \leftarrow G_{n_b}\big] + \texttt{cost}\big[2G_{n_a}G_{n_b} \leftarrow \{G^2_{n_a}, G^2_{n_b}\}\big] \\ &= \big[(A_n+1)(A_n+2)/2 + (A_n+1)(B_n+1) + (B_n+1)(B_n+2)/2\big]\mathbf{s}_u \\ &= [3(3\cdot 2^n - 1)(2^{n+1}-1)\Omega]\mathbf{m}_u. \end{aligned} \tag{11}$$

– **Observation 3.** Aside from additions, computing the three polynomials $h_1$, $h_2$ and $h_3$ from $G^2_{n_a}$, $2G_{n_a}G_{n_b}$, and $G^2_{n_b}$ requires multiplications by $C$ only. Since we are ignoring additions and assuming that multiplications by curve constants are negligible, we assume that there is no extra cost associated in these computations. That is,

$$\texttt{cost}\big[\{h_1, h_2, h_3\} \leftarrow \{G^2_{n_a}, 2G_{n_a}G_{n_b}, G^2_{n_b}\}\big] = 0. \tag{12}$$

– **Observation 4.** The cost of multiplying $G^2_n$ by $g_{n+1}$ is determined by the cost of the required multiplications of the $g_{n_a}$ and $g_{n_b}$ values, and the polynomials $h_1$, $h_2$ and $h_3$. Since $g_{n_a} = g_{n_y} \in \mathbb{F}_{p^u}$, multiplying a $d$-degree polynomial by $g_{n_a}$ requires $d+1$ multiplications in $\mathbb{F}_{p^u}$, and since $g_{n_b} = g_{n_x}\cdot x + g_{n_0}$ has $g_{n_x}, g_{n_0} \in \mathbb{F}_{p^u}$, multiplying a $d$-degree polynomial by $g_{n_b}$ requires $2(d+1)$ $\mathbb{F}_{p^u}$-multiplications. There are four of these types of multiplications required in (9).

$$\begin{aligned}
\text{(i)}: &\quad \texttt{cost}\big[g_{n+1_a}\cdot h_1 \leftarrow \{g_{n+1_a}, h_1\}\big] &&= (\deg(h_1)+1)\mathbf{m}_u \\
\text{(ii)}: &\quad \texttt{cost}\big[g_{n+1_b}\cdot h_2 \leftarrow \{g_{n+1_b}, h_2\}\big] &&= 2(\deg(h_2)+1)\mathbf{m}_u \\
\text{(iii)}: &\quad \texttt{cost}\big[g_{n+1_b}\cdot h_1 \leftarrow \{g_{n+1_b}, h_1\}\big] &&= 2(\deg(h_1)+1)\mathbf{m}_u \\
\text{(iv)}: &\quad \texttt{cost}\big[g_{n+1_a}\cdot h_3 \leftarrow \{g_{n+1_a}, h_3\}\big] &&= (\deg(h_2)+1)\mathbf{m}_u
\end{aligned}$$

In the case of (iv), since $h_3 = h_2\cdot C$, we save 3 multiplications by multiplying $g_{n+1_a}$ and $h_2$ prior to multiplying by $C$. Thus, the total cost of obtaining $G_{n+1}$ given $G^2_n$ and $g_{n+1}$ is the combined costs of (i), (ii), (iii) and (iv) above, which is

$$\begin{aligned}\texttt{cost}\big[G_{n+1} \leftarrow \{G^2_n, g_{n+1}\}\big] &= (3\cdot\deg(h_1) + 2\cdot\deg(h_2) + \deg(h_3) + 6)\mathbf{m}_u \\ &= (3(2A_n+3) + 3(A_n+B_n) + 3)\mathbf{m}_u \\ &= (9A_n + 3B_n + 15)\mathbf{m}_u = (36(2^n-1)+18)\mathbf{m}_u, \end{aligned} \tag{13}$$

**The cost of computing $G_{n+1}$ from $G_n$.** We now collect all of the costs calculated in (3), (11), (12) and (13) to determine the cost of computing $G_{n+1}$ from $G_n$, as

$$\begin{aligned}&\texttt{cost}\big[G_{n+1} \leftarrow G_n\big] = \texttt{cost}\big[G_{n+1} \leftarrow \{G^2_n, g_{n+1}\}\big] + \texttt{cost}\big[g_{n+1}\big] + \texttt{cost}\big[G^2_n \leftarrow G_n\big] \\ &= \texttt{cost}\big[G_{n+1} \leftarrow \{G^2_n, g_{n+1}\}\big] + \texttt{cost}\big[g_{n+1}\big] + \texttt{cost}\big[\{h_1,h_2,h_3\} \leftarrow \{G_{n_a}, G_{n_b}\}\big] \\ &= \texttt{cost}\big[G_{n+1} \leftarrow \{G^2_n, g_{n+1}\}\big] + \texttt{cost}\big[g_{n+1}\big] + \texttt{cost}\big[\{h_1,h_2,h_3\} \\ &\qquad \leftarrow \{G^2_{n_a}, 2G_{n_a}G_{n_b}, G^2_{n_b}\}\big] + \texttt{cost}\big[\{G^2_{n_a}, 2G_{n_a}G_{n_b}, G^2_{n_b}\} \leftarrow \{G_{n_a}, G_{n_b}\}\big] \\ &= \Big[(36(2^n-1)+18) + (m+\Omega s) + 3(3\cdot 2^n - 1)(2^{n+1}-1)\Omega\Big]\mathbf{m}_u, \end{aligned} \tag{14}$$

**The total cost of $N$-delay.** Display (14) allows us to determine the number of $\mathbb{F}_{p^u}$ multiplications required to compute $G_N(x,y)$ from scratch, as follows.

$$\begin{aligned}
&\texttt{cost}\big[G_N(x,y)\big] = \texttt{cost}\big[G_0\big] + \sum_{n=0}^{N-1} \texttt{cost}\big[G_{n+1} \leftarrow G_n\big] \\
&= (m+s\Omega)\mathbf{m}_u + \sum_{n=0}^{N-1}(36(2^n-1)+21) + (m+\Omega s) + (18(2^n-1)+6)\Omega\mathbf{m}_u \\
&= [(N+1)(m+s\Omega) + 3N(\Omega-6) + 3(2^N-1)((2^{N+1}-3)\Omega+12)]\mathbf{m}_u. \quad (15)
\end{aligned}$$

We note that the above cost also incorporates the cost of transforming the point $T$ into $[2^{N+1}]T$, as these costs are accounted for in the multiples of $(m+s\Omega)\mathbf{m}_u$ (see (3)). The other computations we need to consider in an iteration involving $N$-delay are those that occur when evaluating $G_N(x,y)$ at the point $S=(S_x,S_y)$. Setting $n=N$ into (7) reveals that $N$-delay will require the precomputation of the set $\{S_x^i,\ i=1...B_N\}$, and the set $\{S_x^i \cdot S_y,\ i=0...A_N\}$, each of which will be multiplied by an element in $\mathbb{F}_{p^u}$. From Remark 3, we have that such a multiplication costs $e\mathbf{m}_1$, and since there are $A_N+B_N+1$ such elements, we have that

$$\texttt{cost}\big[G_N(S) \leftarrow G_N(x,y)\big] = [A_N+B_N+1]e\mathbf{m}_1 = [6(2^N-1)+2]e\mathbf{m}_1. \quad (16)$$

We combine (15) and (16) to obtain the total cost of $N$-delay as

$$\begin{aligned}
\texttt{cost}\big[G_N(S)\big] &= \texttt{cost}\big[G_N(S) \leftarrow G_N(x,y)\big] + \texttt{cost}\big[G_N(x,y)\big] \\
&= \big[6(2^N-1)+2\big]e\mathbf{m}_1 + \big[(N+1)(m+s\Omega) + 3N(\Omega-6) \\
&\qquad + 3(2^N-1)((2^{N+1}-3)\Omega+12)\big]\mathbf{m}_u + (1+(N+1)\Omega)\mathbf{m}_k, \quad (17)
\end{aligned}$$

where the $(1+(N+1)\Omega)\mathbf{m}_k$ accounts for the $(N+1)$ squarings of the Miller function $f$, as well as the full field multiplication of $f$ with $G_N(S)$ that occurs after $N$-delay.

## 5 Implementing $N$-delay

The advantage of employing $N$-delay over the technique in [10] is the ease at which a standard implementation of Miller's algorithm can be updated to incorporate $N$-delay. The routines for the point doublings/additions and encapsulated line computations that are used in the standard version of Miller's algorithm are the same routines used in $N$-delay, so that this (existing) code is not altered when employing $N$-delay. We refer to these two standard subroutines as `MillerDBL`, which performs steps 3 and 4 in Algorithm 1, and `MillerADD`, which performs steps 8 and 9 in Algorithm 1, both of which are the same subroutines we call in Algorithm 2.

Since $N$-delay performs $N+1$ squarings in the same iteration, we follow the algorithm description in [10] and write the loop parameter in base $2^{N+1}$. Our

goal is to incorporate $N$-delay by injecting a new subroutine into Algorithm 1, and slightly tweaking the original Miller code to account for this alteration. After calling `MillerDBL`, we call the new subroutine `GetNewabArrays`, which transforms $G_n$ into $G_{n+1}$, based on equation (9) and the four observations that followed.

**Algorithm 2** Miller $N$-delay

**Input:** $R$, $S$, $N$, $m = (m_{l_N-1}...m_1, m_0)_{2^{N+1}}$, $f_{[w]R}$ for each unique non-zero $w \in \{m_{l_N-1}, ..., m_1, m_0\}$.
**Output:** $f_{m,R}(S) \leftarrow f$.

1: $T \leftarrow R$, $f \leftarrow 1$.
2: **if** $m_{l_N-1} \neq 1$ **then**
3: $[g_x, g_y, g_0, T] =$ `MillerADD`$(T, [m_i]R)$.
4: $f \leftarrow f \cdot f_{[m_i]R}(S)$.
5: **end if**
6: **for** $i = l_N - 2$ to $0$ **do**
7: Compute $[g_x, g_y, g_0, T] =$ `MillerDBL`$(T)$.
8: $\underline{a}(1) \leftarrow g_y$, $\underline{b}(1) \leftarrow g_0$, $\underline{b}(2) \leftarrow g_x$.
9: $A_n \leftarrow 0$, $B_n \leftarrow 1$.
10: **for** $i = 0$ to $N - 1$ by $1$ **do**
11: Compute $[g_x, g_y, g_0, T] =$ `MillerDBL`$(T)$.
12: Compute $\underline{a}, \underline{b}, A_n, B_n =$ `GetNewabArrays`$(\underline{a}, \underline{b}, A_n, B_n, g_x, g_y, g_0)$.
13: **end for**
14: Evaluate $G = (\underline{a}, \underline{b})$ at $S$.
15: $f \leftarrow f^{2^{N+1}} \cdot G$.
16: **if** $m_i \neq 0$ **then**
17: Compute $[g_x, g_y, g_0, T] =$ `MillerADD`$(T, [m_i]R)$.
18: $f \leftarrow f \cdot f_{[m_i]R} \cdot (g_x \cdot S_x + g_y \cdot S_y + g_0)$
19: **end if**
20: **end for**
21: **return** $f$.

Since non-zero $m_i$ that appear in $m = (m_{l-1}...m_1, m_0)_{2^{N+1}}$ can now take values up to $2^{N+1} - 1$, Algorithm 2 must account for the additions of $[m_i]R$ to the point $T$. We follow the technique in [10] and adjust the step accordingly, by including the precomputed function $f_{[m_i]R}$, with divisor $\text{div}(f_{[m_i]R}) = m_i(R) - ([m_i]R) - (m_i - 1)(\mathcal{O})$, into the addition product on line 18 of Algorithm 2.

## 6 Optimal $N$-delay

This section makes use of (17) to determine the value of $N$ which gives the lowest operation count for all even embedding degrees less than $k = 50$. To obtain the $m$ and $s$ values described in (3) that are required in (17), we couple the recommendations for optimal curve construction in [16] with the fastest

applicable explicit formulas for $D = 1, 3$ curves that admit high-degree twists in [12]. For curves admitting only quadratic twists, we opt for the CM discriminant $D$ that facilitates the best $\rho$-value for the particular embedding degree (see [16]). If these curves do not have $D = 1$ or $D = 3$, we use the best operation count for general curves reported in [1] and [22]. For example, the maximal twist for $k = 10$ is a quadratic twist, and since such twists are admitted on any curve, we opt for Freeman's curves [15] with optimum $\rho = 1$, rather than the $D = 1$ or $D = 3$ curves that achieve $\rho = 1.5$. We report the optimal $N$ values for both $\Omega = 0.8$ and $\Omega = 1$, although we make note that lesser values of $\Omega$, such as those stated in Remark 2, would be more likely to favor higher values of $N$, since lower values of $\Omega$ give a greater weight to multiplications in the operation count, and $\mathbb{F}_{p^k}$-multiplications are what $N$-delay avoids. Since the operation count given by (17) is the total count (in terms of $\mathbb{F}_p$-multiplications) for the equivalent of $N+1$ double-and-add iterations, the counts presented in Table 1 are given as counts equivalent to one iteration of 0-delay (the standard Miller routine in Algorithm 1), and these counts are obtained by dividing the cost in (17) by $N + 1$. We are reporting results for even embedding degrees that are not necessarily 3-smooth. Thus we must extend the standard method of reporting multiplications in fields of extension degree $k = 2^i 3^j$ as $\mathbf{m}_k = 3^i 5^j$ [23, 20, 12], this complexity being a result of coupling Karatsuba multiplication with Toom-Cook multiplication, the former allowing us to write $\mathbf{m}_{2c} = 3\mathbf{m}_c$, whilst the latter allows us to write $\mathbf{m}_{3c} = 5\mathbf{m}_c$. Montgomery [28] extended Karatsuba-like multiplication methods to polynomials (or extension degrees) of degrees 5, 6 and 7, achieving $\mathbf{m}_{5c} = 13\mathbf{m}_c$, $\mathbf{m}_{6c} = 17\mathbf{m}_c$ and $\mathbf{m}_{7c} = 22\mathbf{m}_c$ respectively. We note that the degree 6 result is of no use here, since it is more advantageous to build a six degree extension as a combination of quadratic and cubic extensions. For higher prime extension degrees, we use the more general result given by Weimerskirch and Paar [32], who generalize the Karatsuba algorithm to arbitrary $w$-degree extensions to give $\mathbf{m}_{wc} = [w(w+1)/2]\mathbf{m}_c$. The complexity of multiplications in the field of extension degree $k = 2^{e_2} 3^{e_3} 5^{e_5} 7^{e_7} \cdot \prod p_i^{e_{p_i}}$ are reported in terms of $\mathbb{F}_p$-multiplications as

$$\mathbf{m}_k = \left[3^{e_2} 5^{e_3} 13^{e_5} 22^{e_7} \cdot \prod_{i=1}^{t} (p_i(p_i+1)/2)^{e_i}\right]\mathbf{m}_1, \tag{18}$$

where the $p_i$ are the primes greater than 7 in the prime factorization of $k$. We use (18) to give a fair and relative comparison across all embedding degrees, not to overlook the substantial speed ups recently achieved by El Mrabet and Negre for particular extension degrees [14].

For Miller-full pairings, $N = 0$ was optimal across all embedding degrees, so we do not report the results here (the standard $N = 0$ operation counts in the Miller-full setting for 3-smooth embedding degrees can be found in [12]). Table 1 shows that Miller-lite pairings on curves with even embedding degrees greater than $k = 10$ will always benefit from $N$-delay. Although $N > 2$ is never optimal, it is still interesting to see that $N = 2$ is optimal in many instances. Consider equation (6) which showed that, even after one delayed iterate, the

| | | | | $\Omega = 1$ ($\mathbf{s} = \mathbf{m}$) | | $\Omega = 0.8$ ($\mathbf{s} = 0.8\ \mathbf{m}$) | |
|---|---|---|---|---|---|---|---|
| k | D | $m$, $s$ | $\mathbb{F}_{p^u} \subseteq \mathbb{F}_{p^e} \subset \mathbb{F}_{p^k}$ | $N = 0$ count | **Optimal N** count | $N = 0$ | **Optimal N** count |
| 2 | 3 | 2, 7 | $\mathbb{F}_p = \mathbb{F}_p \subset \mathbb{F}_{p^2}$ | 17 | – | 15 | – |
| 4 | 1 | 2, 8 | $\mathbb{F}_p = \mathbb{F}_p \subset \mathbb{F}_{p^4}$ | 30 | – | 26.6 | – |
| 6 | 3 | 2, 7 | $\mathbb{F}_p = \mathbb{F}_p \subset \mathbb{F}_{p^6}$ | 41 | – | 36.6 | – |
| 8 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^8}$ | 68 | – | 61 | – |
| 10 | some | 1, 11 | $\mathbb{F}_p \subset \mathbb{F}_{p^5} \subset \mathbb{F}_{p^{10}}$ | 100 | – | 90 | – |
| 12 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^{12}}$ | 103 | **1** 96.5 | 92.6 | **1** 85.5 |
| 14 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^7} \subset \mathbb{F}_{p^{14}}$ | 155 | **1** 148 | 140.4 | **1** 132.8 |
| 16 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^4} \subset \mathbb{F}_{p^{16}}$ | 180 | **1** 159.5 | 162.2 | **1** 141.1 |
| 18 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^3} \subset \mathbb{F}_{p^{18}}$ | 165 | **1** 145.5 | 148.6 | **1** 128.5 |
| 20 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^{10}} \subset \mathbb{F}_{p^{20}}$ | 254 | **1** 217.5 | 229 | **1** 191.9 |
| 22 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^{11}} \subset \mathbb{F}_{p^{22}}$ | 428 | **1** 363 | 386.8 | **1** 321.2 |
| 24 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^4} \subset \mathbb{F}_{p^{24}}$ | 287 | **1** 239.5 | 258.6 | **1** 210.5 |
| 26 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^{13}} \subset \mathbb{F}_{p^{26}}$ | 581 | **1** 482.5 | 525 | **1** 425.9 |
| 28 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^7} \subset \mathbb{F}_{p^{28}}$ | 420 | **1** 347 | 378.8 | **1** 305.2 |
| 30 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^{10}} \subset \mathbb{F}_{p^{30}}$ | 409 | **1** 333.5 | 368.6 | **1** 292.5 |
| 32 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^8} \subset \mathbb{F}_{p^{32}}$ | 512 | **1** 418.5 | 461.8 | **1** 367.7 |
| 34 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^{17}} \subset \mathbb{F}_{p^{34}}$ | 961 | **2** 775.3 | 867.8 | **2** 678.7 |
| 36 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^6} \subset \mathbb{F}_{p^{36}}$ | 471 | **1** 382.5 | 424.6 | **1** 335.5 |
| 38 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^{19}} \subset \mathbb{F}_{p^{38}}$ | 1187 | **2** 936.7 | 1071.6 | **2** 817.9 |
| 40 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^{10}} \subset \mathbb{F}_{p^{40}}$ | 732 | **2** 585.6 | 660.2 | **2** 510.5 |
| 42 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^7} \subset \mathbb{F}_{p^{42}}$ | 683 | **2** 536.7 | 615.6 | **2** 465.9 |
| 44 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^{11}} \subset \mathbb{F}_{p^{44}}$ | 1220 | **2** 916.3 | 1099.6 | **2** 792.5 |
| 46 | 1 | 2, 8 | $\mathbb{F}_p \subset \mathbb{F}_{p^{23}} \subset \mathbb{F}_{p^{46}}$ | 1712 | **2** 1308.3 | 1544.8 | **2** 1137.7 |
| 48 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^8} \subset \mathbb{F}_{p^{48}}$ | 835 | **2** 643.3 | 752.6 | **2** 557.5 |
| 50 | 3 | 2, 7 | $\mathbb{F}_p \subset \mathbb{F}_{p^{25}} \subset \mathbb{F}_{p^{50}}$ | 1073 | **1** 881.5 | 970.2 | **1** 778.1 |

**Table 1.** Optimal $N$ values for Miller-lite pairings on different embedding degrees $k \leq 50$.

product of $g$'s becomes a general field element, i.e. $\#G_1(\mathbb{F}_{p^e}) = k/e$ (refer to Definition 1). Any further delay that occurs after the initial delay will actually involve squaring both $f$ and $G_1$ (or $G_n$ for $n > 1$) separately, which are both general elements of $\mathbb{F}_{p^k}$, prior to multiplying them. One might intuitively guess that multiplying $f$ and $G_1$ prior to performing the squaring might be preferred, but clearly this is not the case for embedding degrees where $N = 2$ is optimal. In the case of quadratic twists, this preferred delay is even more surprising since the function updates $g$ are already general elements of $\mathbb{F}_{p^k}$. In agreement with [10], it becomes clear that for Miller-lite pairings where the difference between the fields $\mathbb{F}_{p^u} = \mathbb{F}_p$ and $\mathbb{F}_{p^k}$ is larger than in Miller-full pairings where $\mathbb{F}_{p^u} = \mathbb{F}_{p^e}$, it can be very advantageous to spend many extra computations in $\mathbb{F}_p$ in order to delay one single (and most costly) $\mathbb{F}_{p^k}$-multiplication between $f$ and $G$.

## References


1. Christophe Arene, Tanja Lange, Michael Naehrig, and Christophe Ritzenthaler. Faster pairing computation. Cryptology ePrint Archive, Report 2009/155, 2009. http://eprint.iacr.org/.
2. Paulo S. L. M. Barreto, Hae Yong Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2002.
3. Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Efficient implementation of pairing-based cryptosystems. *J. Cryptology*, 17(4):321–334, 2004.
4. Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2005.
5. Naomi Benger and Michael Scott. Constructing tower extensions for the implementation of pairing-based cryptography. In *WAIFI 2010*, Lecture Notes in Computer Science. Springer, 2010. To appear.
6. Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.
7. Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. http://www.hyperelliptic.org/EFD.
8. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.
9. Sanjit Chatterjee, Palash Sarkar, and Rana Barua. Efficient computation of Tate pairing in projective coordinate over general characteristic fields. In Choonsik Park and Seongtaek Chee, editors, *ICISC*, volume 3506 of *Lecture Notes in Computer Science*, pages 168–181. Springer, 2004.
10. Craig Costello, Colin Boyd, Juan Manuel González Nieto, and Kenneth Koon-Ho Wong. Avoiding full extension field arithmetic in pairing computations. In *AFRICACRYPT 2010*, Lecture Notes in Computer Science. Springer, 2010. To appear.
11. Craig Costello, Huseyin Hisil, Colin Boyd, Juan Manuel González Nieto, and Kenneth Koon-Ho Wong. Faster pairings on special weierstrass curves. In Hovav Shacham and Brent Waters, editors, *Pairing*, volume 5671 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2009.
12. Craig Costello, Tanja Lange, and Michael Naehrig. Faster pairing computations on curves with high-degree twists. In *PKC 2010*, Lecture Notes in Computer Science. Springer, 2010. To appear.
13. M. Prem Laxman Das and Palash Sarkar. Pairing computation on twisted Edwards form elliptic curves. In Galbraith and Paterson [17], pages 192–210.
14. Nadia El Mrabet and Christophe Nègre. Finite field multiplication combining AMNS and DFT approach for pairing cryptography. In *ACISP*, pages 422–436, 2009.
15. David Freeman. Constructing pairing-friendly elliptic curves with embedding degree 10. In Florian Hess, Sebastian Pauli, and Michael E. Pohst, editors, *ANTS*, volume 4076 of *Lecture Notes in Computer Science*, pages 452–465. Springer, 2006.

16. David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *J. Cryptology*, 23(2):224–280, 2010.
17. Steven D. Galbraith and Kenneth G. Paterson, editors. *Pairing-Based Cryptography - Pairing 2008, Second International Conference, Egham, UK, September 1-3, 2008. Proceedings*, volume 5209 of *Lecture Notes in Computer Science*. Springer, 2008.
18. Robert Granger, Dan Page, and Martijn Stam. On small characteristic algebraic tori in pairing-based cryptography. *LMS J. Comput. Math*, 9:64–85, 2006.
19. Florian Hess. Pairing lattices. In Galbraith and Paterson [17], pages 18–38.
20. Florian Hess, Nigel P. Smart, and Frederik Vercauteren. The eta pairing revisited. *IEEE Transactions on Information Theory*, 52(10):4595–4602, 2006.
21. Huseyin Hisil. *Elliptic Curves, Group Law, and Efficient Computation*. PhD thesis, Queensland University of Technology, 2010.
22. Sorina Ionica and Antoine Joux. Another approach to pairing computation in Edwards coordinates. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT*, volume 5365 of *Lecture Notes in Computer Science*, pages 400–413. Springer, 2008. http://eprint.iacr.org/2008/292.
23. Neal Koblitz and Alfred Menezes. Pairing-based cryptography at high security levels. In Nigel P. Smart, editor, *IMA Int. Conf.*, volume 3796 of *Lecture Notes in Computer Science*, pages 13–36. Springer, 2005.
24. Eunjeong Lee, Hyang-Sook Lee, and Cheol-Min Park. Efficient and generalized pairing computation on abelian varieties. *IEEE Transactions on Information Theory*, 55(4):1793–1803, 2009.
25. Xibin Lin, Changan Zhao, Fangguo Zhang, and Yanming Wang. Computing the ate pairing on elliptic curves with embedding degree $k = 9$. *IEICE Transactions*, 91-A(9):2387–2393, 2008.
26. Seiichi Matsuda, Naoki Kanayama, Florian Hess, and Eiji Okamoto. Optimised versions of the ate and twisted ate pairings. In Steven D. Galbraith, editor, *IMA Int. Conf.*, volume 4887 of *Lecture Notes in Computer Science*, pages 302–312. Springer, 2007.
27. Victor S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17:235–261, 2004.
28. Peter L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Trans. Computers*, 54(3):362–369, 2005.
29. Michael Scott and Paulo S. L. M. Barreto. Compressed pairings. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2004.
30. M. Shirase, T. Takagi, D. Choi, D.G. Han, and H. Kim. Efficient computation of Eta pairing over binary field with Vandermonde matrix. *ETRI journal*, 31(2):129–139, 2009.
31. Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, 2010.
32. André Weimerskirch and Christof Paar. Generalizations of the Karatsuba algorithm for efficient implementations. Cryptology ePrint Archive, Report 2006/224, 2006. http://eprint.iacr.org/.

## A Magma Code

We provide a MAGMA implementation of $N$-delay (Algorithm 2) for illustrative purposes. The code below is heavily condensed (with limited commenting)

due to space considerations. The function `MillerNDelay` can be called with inputs $R$, $S$, as described in Section 2, the loop parameter $m$, the two curve constants $a$ and $b$ on $y^2 = x^3 + ax + b$, the parameter $N$ (for $N$-delay), and the field $K = \mathbb{F}_{p^k}$ that contains the coordinates of $R$ and $S$. `MillerNDelay` calls the function `GetNewabArrays`, that transforms the terms $G_{n_a}$ and $G_{n_b}$ into $G_{n+1_a}$ and $G_{n+1_b}$, in accordance with the algebra in equation (9) and the four observations that follow. Although the function `GetNewabArrays` is optimized for general $N$, a real-world implementation would most likely be $N$-specific, and hence much more simple, depending on the $N$ suggested in Table 1. The function `EvaluateLineProduct` is called in the Miller loop to perform the function evaluation of $G$ at $S$ (see line 14 of Algorithm 2). Lastly, the Miller function calls two functions `MillerDBL` and `MillerADD`, which are not provided since optimized versions are specific to the curve equation. We assume `MillerDBL` takes the coordinates of $T$ and the curve constants as inputs, and returns the coefficients $(g_x, g_y, g_0)$ of the Miller doubling line function $g$. We assume `MillerADD` takes the coordinates of $T$ and $R$, and returns the coefficients $(g_x, g_y, g_0)$ of the Miller addition line function $g$.

```
function EvaluateLineProduct(aArray, bArray, An, Bn, Sx, Sy) // Computes G(S)
SxVec:=[Sx^i: i in [1..Bn]]; SxSyVec:=[Sx^i*Sy: i in [0..An]];
G:=0; G+:=bArray[1];     for i:=0 to An do    G+:=aArray[i+1]*SxSyVec[i+1];    end for;
for i:=1 to Bn do    G+:=bArray[i+1]*SxVec[i];     end for;     return G;   end function;
```

```
function GetNewabArrays(aArray, bArray, An, Bn, gx, gy, g0, A, B)     // Initialize Arrays
aSquaresArray:=[0: i in [0..An]]; bSquaresArray:=[0: i in [0..Bn]]; aaProductsArray:=[0: i in
[0..2*An]];
bbProductsArray:=[0: i in [0..2*Bn]]; abProductsArray:=[0: i in [0..(An+Bn)]]; h1Array:=[0: i
in [0..2*An+3]];
h2Array:= [0: i in [0..2*An+1]]; h3Array:=[0: i in [0..2*An+4]];     //Fill in aa, bb, and ab
products
for i:=0 to An by 1 do     aSquaresArray[i+1]:=aArray[i+1]^2;     end for;
for i:=0 to Bn by 1 do     bSquaresArray[i+1]:=bArray[i+1]^2;     end for;
for i:=0 to An by 1 do     aaProductsArray[2*i+1]+:=aSquaresArray[i+1];     end for;
for i:=0 to Bn by 1 do     bbProductsArray[2*i+1]+:=bSquaresArray[i+1];    end for;
for i:=1 to An by 1 do     for j:=0 to (i-1) by 1 do
    aaProductsArray[i+j+1] +:=
(aArray[i+1]+aArray[j+1])^2-aSquaresArray[i+1]-aSquaresArray[j+1];
end for;     end for;
for i:=1 to Bn by 1 do     for j:=0 to (i-1) by 1 do
    bbProductsArray[i+j+1] +:=
(bArray[i+1]+bArray[j+1])^2-bSquaresArray[i+1]-bSquaresArray[j+1];
end for;    end for;
for i:=0 to An by 1 do     for j:=0 to Bn by 1 do
    abProductsArray[i+j+1] +:=
(aArray[i+1]+bArray[j+1])^2-aSquaresArray[i+1]-bSquaresArray[j+1];
end for;    end for;                // Create h arrays
for i:=0 to (2*An) by 1 do         h1Array[i+1]+:= B * aaProductsArray[i+1];
h1Array[i+2]+:= A * aaProductsArray[i+1];     h1Array[i+4]+:= aaProductsArray[i+1];    end
for;
for i:=0 to (2*Bn) by 1 do     h1Array[i+1]+:= bbProductsArray[i+1];    end for;
for i:=0 to (An+Bn) by 1 do
    h2Array[i+1]+:=abProductsArray[i+1]; h3Array[i+1]+:= B*abProductsArray[i+1];
    h3Array[i+2]+:= A*abProductsArray[i+1]; h3Array[i+4]+:= abProductsArray[i+1];     end
for;
aArray:=[0: i in [0..2*An+3]]; bArray:=[0: i in [0..2*An+4]];     // Create and fill output
arrays
for i:=0 to (2*An+3) by 1 do     aArray[i+1]+:=gy*h1Array[i+1];     end for;
for i:=0 to (An+Bn) by 1 do aArray[i+1]+:=g0*h2Array[i+1]; aArray[i+2]+:=gx*h2Array[i+1]; end for;
for i:=0 to (2*An+3) by 1 do bArray[i+1]+:=g0*h1Array[i+1]; bArray[i+2]+:=gx*h1Array[i+1]; end
for;
for i:=0 to (An+Bn+3) by 1 do  bArray[i+1]+:=gy*h3Array[i+1];   end for;
An:= 2*An+3; Bn:= An+1;   return aArray, bArray, An, Bn;   end function;
```

```
function MillerNDelay(R,S,m, a, b,N, K)
Rx:=R[1]; Ry:=R[2]; Rz:=R[3]; Sx:=S[1]; Sy:=S[2]; RMultiplesMatrix:=[[Rx, Ry, Rz]]; // Precompute
R multiples
for i:=2 to (2^(N+1)-1) by 1 do iR:=i*R; RMultiplesMatrix:= Append(RMultiplesMatrix,[iR[1],iR[2],iR[3]]);
end for; // Precompute fR functions
fRAddVec:=[K!1]; addProduct:=fRAddVec[1]; gx, gy, g0, ptx, pty, ptz := MillerDBL(Rx, Ry, Rz,a,
b); addProduct*:= gx*Sx + gy*Sy + g0; fRAddVec:=Append(fRAddVec, addProduct);
for i:=3 to (2^(N+1)-1) by 1 do
    ptx, pty, ptz, fAddValue := MillerADD(ptx, pty, ptz, Rx, Ry, Rz, Sx, Sy); addProduct*:=fAddValue;
    fRAddVec:=Append(fRAddVec, addProduct);   end for;
f:=1; B:= IntegerToSequence(m, 2^(N+1));       if B[#B] ne 1 then     Rx,Ry,Rz,F:=
    MillerADD(Rx,Ry,Rz,RMultiplesMatrix[B[#B]][1],RMultiplesMatrix[B[#B]][2],RMultiplesMatrix[B[#B]][3],Sx,Sy);
F:=F*fRAddVec[B[#B]]; f:=f*F;   end if;
Tx:=Rx; Ty:=Ry; Tz:=Rz;
for i:=#B-1 to 1 by -1 do // START MILLER LOOP
    gx, gy, g0,Tx,Ty,Tz := MillerDBL(Tx, Ty, Tz, a, b); aArray:=[gy]; bArray:=[g0, gx]; An:=0;
Bn:=1;
    for i:=0 to (N-1) by 1 do     gx, gy, g0,Tx,Ty,Tz:=MillerDBL(Tx,Ty,Tz,a, b);
        aArray, bArray, An, Bn:= GetNewabArrays(aArray, bArray, An, Bn, gx, gy, g0, a, b); end
for;
    G:=EvaluateLineProduct(aArray, bArray, An, Bn, Sx, Sy);
    for i:=0 to N by 1 do     f:=f^2;     end for;
    f:=f*G;     if B[i] ne 0 then     Tx, Ty, Tz, F:=
MillerADD(Tx,Ty,Tz,RMultiplesMatrix[B[i]][1],RMultiplesMatrix[B[i]][2],RMultiplesMatrix[B[i]][3],Sx,Sy);
    F:=F*fRAddVec[B[i]];     f:=f*F;     end if;
end for;     return f;     end function;
```