Speeding Up XTR

Martijn Stam[1,⋆] and Arjen K. Lenstra[2]

[1] Technische Universiteit Eindhoven
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
stam@win.tue.nl

[2] Citibank, N.A. and Technische Universiteit Eindhoven
1 North Gate Road, Mendham, NJ 07945-3104, U.S.A.
arjen.lenstra@citicorp.com

Abstract. This paper describes several speedups and simplifications for XTR. The most important results are new XTR double and single exponentiation methods where the latter requires a cheap precomputation. Both methods are on average more than 60% faster than the old methods, thus more than doubling the speed of the already fast XTR signature applications. An additional advantage of the new double exponentiation method is that it no longer requires matrices, thereby making XTR easier to implement. Another XTR single exponentiation method is presented that does not require precomputation and that is on average more than 35% faster than the old method. Existing applications of similar methods to LUC and elliptic curve cryptosystems are reviewed.

Keywords: XTR, addition chains, Fibonacci sequences, binary Euclidean algorithm, LUC, ECC.

1 Introduction

The XTR public key system was introduced at Crypto 2000 [10]. From a security point of view XTR is a traditional subgroup discrete logarithm system, as was proved in [10]. It uses a non-standard way to represent and compute subgroup elements to achieve substantial computational and communication advantages over traditional representations. XTR of security equivalent to 1024-bit RSA achieves speed comparable to cryptosystems based on random elliptic curves over random prime fields (ECC) of equivalent security. The corresponding XTR public keys are only about twice as large as ECC keys, assuming global system parameters – without the last requirement the sizes of XTR and ECC public keys are about the same. Furthermore, parameter initialization from scratch for XTR takes a negligible amount of computing time, unlike RSA and ECC.

This paper describes several important speedups for XTR, while at the same time simplifying its implementation. In the first place the field arithmetic as described in [10] is improved by combining the modular reduction steps. More importantly, a new application of a method from [15] is presented that results in

[⋆] The first author is sponsored by STW project EWI.4536

© Springer-Verlag Berlin Heidelberg 2001

an XTR exponentiation iteration that can be used for three different purposes. In the first place these improvements result in an XTR double exponentiation method that is on average more than 60% faster than the double exponentiation from [10]. Such exponentiations are used in XTR ElGamal-like signature verifications. Furthermore, they result in two new XTR single exponentiation methods, one that is on average about 60% faster than the method from [10] but that requires a one-time precomputation, and a generic one without precomputation that is on average 35% faster than the old method.

Examples where precomputation can typically be used are the ‘first’ of the two exponentiations (per party) in XTR Diffie-Hellman key agreement, XTR ElGamal-like signature generation, and, to a lesser extent, XTR-ElGamal encryption. The new generic XTR single exponentiation can be used in the ‘second’ XTR Diffie-Hellman exponentiation and in XTR-ElGamal decryption. As a result the runtime of XTR signature applications is more than halved, the time required for XTR Diffie-Hellman is almost halved, and XTR-ElGamal encryption and decryption can both be expected to run at least 35% faster (with encryption running more than 60% faster after precomputation).

The method from [15] was developed to compute Lucas sequences. It can thus immediately be applied to the LUC cryptosystem [18]. It was shown [16] that it can also be applied to ECC. The resulting methods compare favorably to methods that have been reported in the literature [5]. Because they are not generally known their runtimes are reviewed at the end of this paper.

The double exponentiation method from [10] uses matrices. The new method does away with the matrices, thereby removing the esthetically least pleasing aspect of XTR. For completeness, another double exponentiation method is shown that does not require matrices. It is directly based on the iteration from [10] and does not achieve a noticeable speedup over the double exponentiation from [10], since the matrix steps that are no longer needed, though cumbersome, are cheap.

This paper is organized as follows. Section 2 reviews the results from [10] needed for this paper. It includes a description of the faster field arithmetic and matrix-less XTR double exponentiation based on the iteration from [10]. The 60% faster (and also matrix-less) XTR double exponentiation is presented in Section 3. Applications of the method from Section 3 to XTR single exponentiation with precomputation and to generic XTR single exponentiation are described in Sections 4 and 5, respectively. In Section 6 the runtime claims are substantiated by direct comparison with the timings from [10]. Section 7 reviews the related LUC and ECC results.

2 XTR Background

For background and proofs of the statements in this section, see [10]. Let p and q be primes with $p \equiv 2 \bmod 3$ and q dividing $p^2 - p + 1$, and let g be a generator of the order q subgroup of $\mathbf{F}_{p^6}^*$. For $h \in \mathbf{F}_{p^6}^*$ its trace $Tr(h)$ over $\mathbf{F}_{p^2}$ is defined as the sum of the conjugates over $\mathbf{F}_{p^2}$ of h:

$$Tr(h) = h + h^{p^2} + h^{p^4} \in \mathbf{F}_{p^2}.$$

Because the order of h divides p^6-1 the trace over $\mathbf{F}_{p^2}$ of h equals the trace of the conjugates over $\mathbf{F}_{p^2}$ of h:

$$Tr(h) = Tr(h^{p^2}) = Tr(h^{p^4}). \tag{1}$$

If $h \in \langle g \rangle$ then its order divides p^2-p+1, so that

$$Tr(h) = h + h^{p-1} + h^{-p}$$

since $p^2 \equiv p-1 \bmod (p^2-p+1)$ and $p^4 \equiv -p \bmod (p^2-p+1)$. In XTR elements of $\langle g \rangle$ are represented by their trace over $\mathbf{F}_{p^2}$. It follows from (1) that XTR makes no distinction between an element of $\langle g \rangle$ and its conjugates over $\mathbf{F}_{p^2}$.

The discrete logarithm (DL) problem in $\langle g \rangle$ is to compute for a given $h \in \langle g \rangle$ the unique $y \in \{0,1,\ldots,q-1\}$ such that $g^y = h$. The XTR-DL problem is to compute for a given $Tr(h)$ with $h \in \langle g \rangle$ an integer $y \in \{0,1,\ldots,q-1\}$ such that $Tr(g^y) = Tr(h)$. If y solves an XTR-DL problem then $(p-1)y$ and $-py$ (both taken modulo q) are solutions too. It is proved in [10, Theorem 5.2.1] that the XTR-DL problem is equivalent to the DL problem in $\langle g \rangle$, with similar equivalences with respect to the Diffie-Hellman and Decision Diffie-Hellman problems. Furthermore, it is argued in [10] that if q is sufficiently large (which will be the case), then the DL problem in $\langle g \rangle$ is as hard as it is in $\mathbf{F}^*_{p^6}$. This argument is the most commonly misunderstood aspect of XTR and therefore rephrased here.

Because of the Pohlig-Hellman algorithm [17] and the fact that $p^6-1 = (p-1)(p+1)(p^2+p+1)(p^2-p+1)$, the general DL problem in $\mathbf{F}^*_{p^6}$ reduces to the DL problems in the following four subgroups of $\mathbf{F}^*_{p^6}$:

- The subgroup of order $p-1$, which can efficiently be embedded in $\mathbf{F}_p$.
- The subgroup of order $p+1$ dividing p^2-1, which can efficiently be embedded in $\mathbf{F}_{p^2}$ but not in $\mathbf{F}_p$.
- The subgroup of order p^2+p+1 dividing p^3-1, which can efficiently be embedded in $\mathbf{F}_{p^3}$ but not in $\mathbf{F}_p$.
- The subgroup of order p^2-p+1, which cannot be embedded in any true subfield of $\mathbf{F}_{p^6}$.

So, to solve the DL problem in $\mathbf{F}^*_{p^6}$ in the most general case, four DL problems must be solved. Three of these DL problems can efficiently be reformulated as DL problems in multiplicative groups of the true subfields $\mathbf{F}_p$, $\mathbf{F}_{p^2}$, and $\mathbf{F}_{p^3}$ of $\mathbf{F}_{p^6}$. With the current state of the art of the DL problem in extension fields, these latter three problems are believed to be strictly (and substantially) easier than the DL problem in $\mathbf{F}^*_{p^6}$. But that means that the subgroup of order p^2-p+1 is, so to speak, the subgroup that is responsible for the difficulty of the DL problem in $\mathbf{F}^*_{p^6}$. With a proper choice of q dividing p^2-p+1, this subgroup DL problem is equivalent to the problem in $\langle g \rangle$. This implies that the DL problem in $\langle g \rangle$ is as hard as it is in $\mathbf{F}^*_{p^6}$, unless the latter problem is not as hard as it is currently believed to be. It also follows that, if the DL problem in $\langle g \rangle$ is easier than it is in $\mathbf{F}^*_{p^6}$, then the problem in $\mathbf{F}^*_{p^6}$ can be at most as hard as it is in $\mathbf{F}^*_p$, $\mathbf{F}^*_{p^2}$, or $\mathbf{F}^*_{p^3}$. Proving such a result would require a major breakthrough.

Thus, for cryptographic purposes and given the current state of knowledge regarding the DL problem in extension fields, XTR and $\mathbf{F}_{p^6}$ give the same security. For p and q of about 170 bits the security is at least equivalent to 1024-bit RSA and approximately equivalent to 170-bit ECC.

XTR has two main advantages compared to ordinary representation of elements of $\langle g \rangle$:

- It is shorter, since $Tr(h) \in \mathbf{F}_{p^2}$, whereas representing an element of $\langle g \rangle$ requires in general an element of $\mathbf{F}_{p^6}$, i.e., three times more bits;
- It allows faster arithmetic, because given $Tr(g)$ and u the value $Tr(g^u)$ can be computed substantially faster than g^u can be computed given g and u.

In this paper it is shown that $Tr(g^u)$ can be computed even faster than shown in [10].

Throughout this paper, c_u denotes $Tr(g^u) \in \mathbf{F}_{p^2}$, for some fixed p and g of order q as above. Note that $c_0 = 3$. In [10,11,12] it is shown how p, q, and c_1 can be found quickly. In particular there is no need to find an explicit representation of $g \in \mathbf{F}_{p^6}$.

2.1 Improved $\mathbf{F}_{p^2}$ Arithmetic. Because $p \equiv 2 \bmod 3$, the zeros α and α^p of the polynomial $(X^3-1)/(X-1) = X^2+X+1$ form an optimal normal basis for $\mathbf{F}_{p^2}$ over $\mathbf{F}_p$. An element $x \in \mathbf{F}_{p^2}$ is represented as $x_1\alpha + x_2\alpha^2$ with $x_1, x_2 \in \mathbf{F}_p$. From $\alpha^2 = \alpha^p$ if follows that $x^p = x_2\alpha + x_1\alpha^2$, so that p-th powering in $\mathbf{F}_{p^2}$ is free. In [10] the product $(x_1\alpha + x_2\alpha^2)(y_1\alpha + y_2\alpha^2)$ is computed by computing x_1y_1, x_2y_2, $(x_1+x_2)(y_1+y_2) \in \mathbf{F}_p$, so that $x_1y_2 + x_2y_1 \in \mathbf{F}_p$ and the product

$$(x_2y_2 - x_1y_2 - x_2y_1)\alpha + (x_1y_1 - x_1y_2 - x_2y_1)\alpha^2 \in \mathbf{F}_{p^2}$$

follow using four subtractions. This implies that products in $\mathbf{F}_{p^2}$ can be computed at the cost of three multiplications in $\mathbf{F}_p$ (as usual, the small number of additions and subtractions is not counted).

For a regular multiplication of $u, v \in \mathbf{F}_p$ the field elements u and v are mapped to integers $\bar{u}, \bar{v} \in \{0, 1, \ldots, p-1\}$, the integer product $\bar{w} = \bar{u}\bar{v} \in \mathbf{Z}$ is computed (the 'multiplication step'), the remainder $\bar{w} \bmod p \in \{0, 1, \ldots, p-1\}$ is computed (the 'reduction step'), and finally the resulting integer $\bar{w} \bmod p$ is mapped to $\mathbf{F}_p$. The reduction step is somewhat costlier than the multiplication step; the mappings between $\mathbf{F}_p$ and $\mathbf{Z}$ are negligible. The same applies if Montgomery arithmetic [13] is used, but then the reduction and multiplication step are about equally costly.

It follows that the computation of $(x_1\alpha + x_2\alpha^2)(y_1\alpha + y_2\alpha^2)$ can be made faster by computing, in the above notation, $\bar{w}_1 = \bar{x}_2\bar{y}_2 - \bar{x}_1\bar{y}_2 - \bar{x}_2\bar{y}_1 \in \mathbf{Z}$ and $\bar{w}_2 = \bar{x}_1\bar{y}_1 - \bar{x}_1\bar{y}_2 - \bar{x}_2\bar{y}_1 \in \mathbf{Z}$ using four integer multiplications, followed by two reductions $\bar{w}_1 \bmod p$ and $\bar{w}_2 \bmod p$. This works both for regular and Montgomery arithmetic. Because the intermediate results are at most $3p^2$ in absolute value the resulting final reductions are of the same cost as the original reductions (with additional subtraction correction in Montgomery arithmetic, at negligible extra cost). As a result, products in $\mathbf{F}_{p^2}$ can be computed at the cost of just two and a half multiplications in $\mathbf{F}_p$, namely the usual three multiplication steps and just two reduction steps. If regular arithmetic is used the speedup can be expected to be somewhat larger. It follows in a similar way that the computation of $xz - yz^p \in \mathbf{F}_{p^2}$ for $x, y, z \in \mathbf{F}_{p^2}$ can be reduced from four multiplications in $\mathbf{F}_p$ to the same cost as three multiplications in $\mathbf{F}_p$; refer to [10, Section 2.1] for the details of that computation. Combining, or postponing, the reduction steps in this way is not at all new. See for instance [4] for a much earlier application.

This results in the following improved version of [10, Lemma 2.1.1].

Lemma 2.2 *Let $x, y, z \in \mathbf{F}_{p^2}$ with $p \equiv 2 \bmod 3$.*
i. *Computing x^p is free.*
ii. *Computing x^2 takes two multiplications in $\mathbf{F}_p$.*
iii. *Computing xy costs the same as two and a half multiplications in $\mathbf{F}_p$.*
iv. *Computing $xz - yz^p$ costs the same as three multiplications in $\mathbf{F}_p$.*

Efficient computation of c_u given p, q, and c_1 is based on the following facts.

2.3 Facts. Fact 2b follows from Lemma 2.2 and Facts 1b and 2a. The other facts are derived as in [10].

1. Identities involving traces of powers, with $u, v \in \mathbf{Z}$:
   a) $c_{-u} = c_{up} = c_u^p$. It follows from Lemma 2.2.i that negations and p-th powers can be computed for free.
   b) $c_{u+v} = c_u c_v - c_v^p c_{u-v} + c_{u-2v}$. It follows from Lemma 2.2.i and iv that c_{u+v} can be computed at the cost of three multiplications in $\mathbf{F}_p$ if c_u, c_v, c_{u-v}, and c_{u-2v} are given.
   c) If $c_u = \tilde{c}_1$, then $\tilde{c}_v$ denotes the trace of the v-th power g^{uv} of g^u, so that $c_{uv} = \tilde{c}_v$.
2. Computing traces of powers, with $u \in \mathbf{Z}$:
   a) $c_{2u} = c_u^2 - 2c_u^p$ takes two multiplications in $\mathbf{F}_p$.
   b) $c_{3u} = c_u^3 - 3c_u^{p+1} + 3$ costs four and a half multiplications in $\mathbf{F}_p$, and produces c_{2u} as a side-result.
   c) $c_{u+2} = c_1 c_{u+1} - c_1^p c_u + c_{u-1}$ costs three multiplications in $\mathbf{F}_p$.
   d) $c_{2u-1} = c_{u-1} c_u - c_1^p c_u^p + c_{u+1}^p$ costs three multiplications in $\mathbf{F}_p$.
   e) $c_{2u+1} = c_{u+1} c_u - c_1 c_u^p + c_{u-1}^p$ costs three multiplications in $\mathbf{F}_p$.

Let S_u denote the triple (c_{u-1}, c_u, c_{u+1}); thus $S_1 = (3, c_1, c_1^2 - 2c_1^p)$. The triple $S_{2u-1} = (c_{2(u-1)}, c_{2u-1}, c_{2u})$ can be computed from S_u and c_1 by applying Fact 2a twice to compute $c_{2(u-1)}$ and c_{2u} based on c_{u-1} and c_u, respectively, and by applying Fact 2d to compute c_{2u-1} based on $S_u = (c_{u-1}, c_u, c_{u+1})$ and c_1. This takes seven multiplications in $\mathbf{F}_p$. The triple S_{2u+1} can be computed in a similar fashion from S_u and c_1 at the cost of seven multiplications in $\mathbf{F}_p$ (using Fact 2e to compute c_{2u+1}).

Let v be a non-negative integer, and let $v = \sum_{i=0}^{r-1} v_i 2^i$ be the binary representation of v, where $v_i \in \{0, 1\}$, $r > 0$, and $v_{r-1} = 1$. It is well known that the v-th power of an element of, say, a finite field can be computed using the ordinary square and multiply method based on the binary representation of v. A similar iteration can be used to compute S_{2v+1}, given S_1.

2.4 XTR Single Exponentiation (cf. [10, Algorithm 2.3.7]). Let S_1, c_1, and $v_{r-1}, v_{r-2}, \ldots, v_0 \in \{0,1\}$ be given, let $y = 1$ and $e = 0$ (so that $2e+1 = y$; the values y and e are included for expository purposes only). To compute S_{2v+1} with $v = \sum_{i=0}^{r-1} v_i 2^i$, do the following for $i = r-1, r-2, \ldots, 0$ in succession:

Bit off If $v_i = 0$, then compute S_{2y-1} based on S_y and c_1, replace S_y by S_{2y-1} (and thus S_{2e+1} by $S_{2(2e)+1}$ because it follows from $2e+1 = y$ that $2(2e)+1 = 4e+1 = 2y-1$), replace y by $2y-1$, and e by $2e$ (so that the invariant $2e+1 = y$ is maintained).

Bit on Else if $v_i = 1$, then compute S_{2y+1} based on S_y and c_1, replace S_y by S_{2y+1} (and thus S_{2e+1} by $S_{2(2e+1)+1}$ because it follows from $2e+1 = y$ that $2(2e+1)+1 = 4e+3 = 2y+1$), replace y by $2y+1$, and e by $2e+1$ (so that the invariant $2e+1 = y$ is maintained).

As a result $e = v$. Because $2e+1 = y$ the final S_y equals S_{2v+1}. Note that v_{r-1}, or any other v_i, does not have to be non-zero.

Both the 'bit off' and the 'bit on' step of Algorithm 2.4 take seven multiplications in $\mathbf{F}_p$. Thus, given an odd positive integer $t < q$ and S_1, the triple $S_t = (c_{t-1}, c_t, c_{t+1})$ can be computed in $7\log_2 t$ multiplications in $\mathbf{F}_p$. In [10] this was $8\log_2 t$ because of the slower field arithmetic used there. The restriction that t is odd and positive is easily removed: if t is even, then first compute S_{t-1} and next apply Fact 2c, and if t is negative, then use Fact 1a.

In Algorithm 2.4, the trace c_1 of g in $S_1 = (c_0, c_1, c_2) = (3, c_1, c_1^2 - 2c_1^p)$ can be replaced by the trace c_t of the t-th power g^t of g (cf. Fact 1c): with $\tilde{c}_1 = c_t$, $\tilde{S}_1 = (\tilde{c}_0, \tilde{c}_1, \tilde{c}_2) = (3, c_t, c_{2t}) = (3, c_t, c_t^2 - 2c_t^p)$, and the previous paragraph, the triple $\tilde{S}_v = (\tilde{c}_{v-1}, \tilde{c}_v, \tilde{c}_{v+1}) = (c_{(v-1)t}, c_{vt}, c_{(v+1)t})$ can be computed in $7\log_2 v$ multiplications in $\mathbf{F}_p$, for any positive integer $v < q$.

Now let $v = \sum_{i=0}^{r-1} v_i 2^i$ as above and let

$$v' = 2^r k + v = \sum_{i=0}^{s+r-1} v_i 2^i$$

for some integer $k \geq 1$. After the first s iterations of the application of Algorithm 2.4 to S_1, c_1, and $v_{s+r-1}, v_{s+r-2}, \ldots, v_0$ the value for e equals k and $S_y = S_{2k+1}$. The remaining r iterations result in $S_{2v'+1} = S_{2^{r+1}k+2v+1}$, and are the same as if Algorithm 2.4 was applied to S_y (as opposed to S_1) and $v_{r-1}, v_{r-2}, \ldots, v_0$. It follows that if Algorithm 2.4 is applied to S_{2k+1}, c_1, and $v_{r-1}, v_{r-2}, \ldots, v_0$, then the resulting value is $S_{2^{r+1}k+2v+1}$. Note that the v_i's do not have to be non-zero. Thus, given any (odd or even) $t < 2^{r+1}$, S_k, and c_1, the triple $S_{2^{r+1}k+t}$ can be computed in $7\log_2 t$ multiplications in $\mathbf{F}_p$. This leads to the following double exponentiation method for XTR.

2.5 Matrix-Less XTR Double Exponentiation. Let a and b be integers with $0 < a, b < q$, and let S_k and c_1 be given. To compute c_{bk+a} do the following.

1. Let r be such that $2^r < q < 2^{r+1}$.
2. Compute $d = b/2^{r+1} \bmod q$ and $t = a/d \bmod q$.
3. Compute $S_{2^{r+1}k+t}$:
   - Use Facts 2a and 2e to compute S_{2k+1} based on S_k.
   - If t is odd let $t' = t$, else let $t' = t - 1$.
   - Let $t' = 2v + 1$.
   - Let $v = \sum_{i=0}^{r-1} v_i 2^i$ with $v_i \in \{0, 1\}$ (and $v_{r-1}, v_{r-2}, \ldots$ possibly zero).
   - Apply Algorithm 2.4 to S_{2k+1}, c_1, and $v_{r-1}, v_{r-2}, \ldots, v_0$, resulting in $S_{2^{r+1}k+t'}$.
   - If t is odd then $S_{2^{r+1}k+t} = S_{2^{r+1}k+t'}$, else use Fact 2c to compute $S_{2^{r+1}k+t} = S_{2^{r+1}k+t'+1}$ based on $S_{2^{r+1}k+t'}$.
4. Let $\tilde{c}_1 = c_{2^{r+1}k+t}$.
5. Compute $\tilde{S}_1 = (\tilde{c}_0, \tilde{c}_1, \tilde{c}_2) = (3, \tilde{c}_1, \tilde{c}_1^2 - 2\tilde{c}_1^p)$ (cf. Fact 1c).
6. Apply Algorithm 2.4 to $\tilde{S}_1$, $\tilde{c}_1$ and the bits containing the binary representation of d, resulting in $\tilde{S}_d = (\tilde{c}_{d-1}, \tilde{c}_d, \tilde{c}_{d+1})$.
7. The resulting $\tilde{c}_d$ equals $c_{d(2^{r+1}k+t) \bmod q} = c_{bk+a}$.

Algorithm 2.5 takes about $14 \log_2 q$ multiplications in $\mathbf{F}_p$. This is a small constant number of multiplications in $\mathbf{F}_p$ better than [10, Algorithm 2.4.8] (assuming the faster field arithmetic is used there too). For realistic choices of q the speedup achieved using Algorithm 2.5 is thus barely noticeable. Nevertheless, it is a significant result because the fact that the matrices as required for [10, Algorithm 2.4.8] are no longer needed, facilitates implementation of XTR. In Section 3 of this paper a more substantial improvement over the double exponentiation method from [10] is described that does not require matrices either.

3 Improved Double Exponentiation

In this section it is shown how c_{bk+a} can be computed based on S_k and c_1 (or, equivalently, based on $S_{k-1} = (c_{k-2}, c_{k-1}, c_k)$ and c_1, cf. Fact 2.3.1b) in a single iteration, as opposed to the two iterations in Algorithm 2.5. For greater generality, it is shown how $c_{bk+a\ell}$ is computed, based on c_k, c_ℓ, $c_{k-\ell}$, and $c_{k-2\ell}$.

A rough outline of the new XTR double exponentiation method is as follows. Let $u = k$, $v = \ell$, $d = b$, and $e = a$. It follows that $ud + ve = bk + a\ell$ and that c_u, v_v, c_{u-v}, and c_{u-2v} are known. The values of d and e are decreased, while at the same time u and v (and thereby c_u, c_v, c_{u-v}, and c_{u-2v}) are updated, in order to maintain the invariant $ud + ve = bk + a\ell$. The changes in d and e are effected in such a way that at a given point $d = e$. But if $d = e$, then $bk + a\ell = ud + ve = d(u + v)$, so that $c_{bk+a\ell}$ follows by computing c_{u+v} and next $c_{d(u+v)}$ (cf. Fact 2.3.1c).

There are various ways in which d and e can be changed. The most efficient method to date was proposed by P.L. Montgomery in [15], for the computation of second degree recurrent sequences. The method below is an adaptation of [15, Table 4] to the present case of third degree sequences.

3.1 Simultaneous XTR Double Exponentiation. Let a, b, c_k, c_ℓ, $c_{k-\ell}$, and $c_{k-2\ell}$ be given, with $0 < a, b < q$. To compute $c_{bk+a\ell}$ do the following.

1. Let $u = k$, $v = \ell$, $d = b$, $e = a$, $c_u = c_k$, $c_v = c_\ell$, $c_{u-v} = c_{k-\ell}$, $c_{u-2v} = c_{k-2\ell}$, $f_2 = 0$, and $f_3 = 0$ (u and v are carried along for expository purposes only).
2. As long as d and e are both even, replace (d, e) by $(d/2, e/2)$ and f_2 by f_2+1.
3. As long as d and e are both divisible by 3, replace (d, e) by $(d/3, e/3)$ and f_3 by $f_3 + 1$.
4. As long as $d \neq e$ replace $(d, e, u, v, c_u, c_v, c_{u-v}, c_{u-2v})$ by the 8-tuple given below.
   a) If $d > e$ then
      i. if $d \leq 4e$, then $(e, d-e, u+v, u, c_{u+v}, c_u, c_v, c_{v-u})$.
      ii. else if d is even, then $(\frac{d}{2}, e, 2u, v, c_{2u}, c_v, c_{2u-v}, c_{2(u-v)})$.
      iii. else if e is odd, then $(\frac{d-e}{2}, e, 2u, u+v, c_{2u}, c_{u+v}, c_{u-v}, c_{-2v})$.
      iv. **optional:**
      else if $d \equiv e \bmod 3$, then $(\frac{d-e}{3}, e, 3u, u+v, c_{3u}, c_{u+v}, c_{2u-v}, c_{u-2v})$.
      v. else (e is even), then $(\frac{e}{2}, d, 2v, u, c_{2v}, c_u, c_{2v-u}, c_{2(v-u)})$.
   b) Else (if $e > d$)
      i. if $e \leq 4d$, then $(d, e-d, u+v, v, c_{u+v}, c_v, c_u, c_{u-v})$.
      ii. else if e is even, then $(\frac{e}{2}, d, 2v, u, c_{2v}, c_u, c_{2v-u}, c_{2(v-u)})$.
      iii. else if d is odd, then $(\frac{e-d}{2}, d, 2v, u+v, c_{2v}, c_{u+v}, c_{v-u}, c_{-2u})$.
      iv. **optional:**
      else if $e \equiv 0 \bmod 3$, then $(\frac{e}{3}, d, 3v, u, c_{3v}, c_u, c_{3v-u}, c_{3v-2u})$.
      v. **optional:**
      else if $e \equiv d \bmod 3$, then $(\frac{e-d}{3}, d, 3v, u+v, c_{3v}, c_{u+v}, c_{2v-u}, c_{v-2u})$.
      vi. else (d is even), then $(\frac{d}{2}, e, 2u, v, c_{2u}, c_v, c_{2u-v}, c_{2(u-v)})$.
5. Apply Fact 2.3.1b to c_u, c_v, c_{u-v}, and c_{u-2v}, to compute $\tilde{c}_1 = c_{u+v}$.
6. Apply Algorithm 2.4 to $\tilde{S}_1 = (3, \tilde{c}_1, \tilde{c}_1 - 2\tilde{c}_1^p)$, $\tilde{c}_1$, and the binary representation of d, resulting in $\tilde{c}_d = c_{d(u+v)}$ (cf. Fact 2.3.1c). Alternatively, and on average faster, apply Algorithm 5.1 described below to compute $\tilde{c}_d = c_{d(u+v)}$ based on $\tilde{c}_1$ (note that this results in a recursive call to Algorithm 3.1).
7. Compute $c_{2^{f_2}d(u+v)}$ based on $c_{d(u+v)}$ by applying Fact 2.3.2a f_2 times.
8. Compute $c_{3^{f_3}2^{f_2}d(u+v)}$ based on $c_{2^{f_2}d(u+v)}$ by applying Fact 2.3.2b f_3 times.

The asymmetry between Steps 4a and 4b is caused by the asymmetry between u and v, i.e., c_{u-2v} is available but c_{v-2u} is not. As a consequence, the case '$d \equiv 0 \bmod 3$' is slower than the case '$e \equiv 0 \bmod 3$' (Step 4(b)iv), and its inclusion would slow down Algorithm 3.1.

Steps 4(a)i and 4(b)i each require a single application of Fact 2.3.1b at the cost of three multiplications in $\mathbf{F}_p$. Steps 4(a)v and 4(b)ii each require two applications of Fact 2.3.2a at the cost of $2 + 2 = 4$ multiplications in $\mathbf{F}_p$. Steps 4(a)ii, 4(a)iii, 4(b)iii, and 4(b)vi each require an application of Fact 2.3.1b and two applications of Fact 2.3.2a at the cost of $3+2+2 = 7$ multiplications in $\mathbf{F}_p$. The three optional steps 4(a)iv, 4(b)iv, and 4(b)v each require two applications of Fact 2.3.1b and one application of Fact 2.3.2b for a total cost of $3+3+4.5 = 10.5$ multiplications in $\mathbf{F}_p$.

In Table 1 the number of multiplications in $\mathbf{F}_p$ required by Algorithm 3.1 is given, both with and without optional steps 4(a)iv, 4(b)iv, and 4(b)v. Each set of entries is averaged over the same collection of 2^{20} randomly selected t's, a's, and b's, with t of the size specified in Table 1 and a and b randomly selected from $\{1, 2, \ldots, t-1\}$. For regular double exponentiation $t \approx q$, but $t \approx \sqrt{q}$ for the application in Section 4. It follows from Table 1 that inclusion of the optional steps leads to an overall reduction of more than 6% in the expected number of multiplications in $\mathbf{F}_p$. For the optional steps it is convenient to keep track of the residue classes of d and e modulo 3. These are easily updated if any of the other steps applies, but require a division by 3 if either one of the optional steps is carried out. It depends on the implementation and the platform whether or not an overall saving is obtained by including the optional steps. In most software implementations it will most likely be worthwhile.

Table 1. Empirical performance of Algorithm 3.1, with $0 < a, b < t$.

multiplications in $\mathbf{F}_p$

	including steps 4(a)iv, 4(b)iv, and 4(b)v			without steps 4(a)iv, 4(b)iv, and 4(b)v		
$\lceil \log_2 t \rceil = T$	average	standard deviation σ	$\sigma/\sqrt{T}$	average	standard deviation σ	$\sigma/\sqrt{T}$
60	$350.01 = 5.83T$	$20.5 = 0.34T$	2.65	$372.89 = 6.21T$	$30.0 = 0.50T$	3.88
70	$410.42 = 5.86T$	$22.2 = 0.32T$	2.65	$437.41 = 6.25T$	$32.6 = 0.47T$	3.89
80	$470.84 = 5.89T$	$23.7 = 0.30T$	2.65	$501.94 = 6.27T$	$34.8 = 0.44T$	3.90
90	$531.21 = 5.90T$	$25.2 = 0.28T$	2.66	$566.36 = 6.29T$	$37.0 = 0.41T$	3.90
100	$591.63 = 5.92T$	$26.5 = 0.27T$	2.65	$630.85 = 6.31T$	$39.1 = 0.39T$	3.91
110	$652.03 = 5.93T$	$27.8 = 0.25T$	2.65	$695.40 = 6.32T$	$41.1 = 0.37T$	3.92
120	$712.39 = 5.94T$	$29.1 = 0.24T$	2.66	$759.87 = 6.33T$	$43.0 = 0.36T$	3.93
130	$772.78 = 5.94T$	$30.2 = 0.23T$	2.65	$824.31 = 6.34T$	$44.6 = 0.34T$	3.92
140	$833.19 = 5.95T$	$31.5 = 0.22T$	2.66	$888.91 = 6.35T$	$46.4 = 0.33T$	3.92
150	$893.66 = 5.96T$	$32.5 = 0.22T$	2.65	$953.34 = 6.36T$	$48.1 = 0.32T$	3.93
160	$953.98 = 5.96T$	$33.6 = 0.21T$	2.66	$1017.79 = 6.36T$	$49.7 = 0.31T$	3.93
170	$1014.42 = 5.97T$	$34.7 = 0.20T$	2.66	$1082.36 = 6.37T$	$51.3 = 0.30T$	3.93
180	$1074.84 = 5.97T$	$35.7 = 0.20T$	2.66	$1146.88 = 6.37T$	$52.7 = 0.29T$	3.93
190	$1135.19 = 5.97T$	$36.6 = 0.19T$	2.66	$1211.34 = 6.38T$	$54.3 = 0.29T$	3.94
200	$1195.58 = 5.98T$	$37.6 = 0.19T$	2.66	$1275.82 = 6.38T$	$55.7 = 0.28T$	3.94
210	$1256.05 = 5.98T$	$38.5 = 0.18T$	2.66	$1340.23 = 6.38T$	$57.1 = 0.27T$	3.94
220	$1316.42 = 5.98T$	$39.5 = 0.18T$	2.66	$1404.75 = 6.39T$	$58.5 = 0.27T$	3.94
230	$1376.87 = 5.99T$	$40.3 = 0.18T$	2.66	$1469.36 = 6.39T$	$59.7 = 0.26T$	3.94
240	$1437.25 = 5.99T$	$41.2 = 0.17T$	2.66	$1533.89 = 6.39T$	$61.1 = 0.25T$	3.94
250	$1497.61 = 5.99T$	$42.0 = 0.17T$	2.66	$1598.22 = 6.39T$	$62.3 = 0.25T$	3.94
260	$1558.00 = 5.99T$	$42.9 = 0.17T$	2.66	$1662.80 = 6.40T$	$63.7 = 0.24T$	3.95
270	$1618.47 = 5.99T$	$43.8 = 0.16T$	2.66	$1727.31 = 6.40T$	$64.9 = 0.24T$	3.95
280	$1678.74 = 6.00T$	$44.5 = 0.16T$	2.66	$1791.85 = 6.40T$	$66.1 = 0.24T$	3.95
290	$1739.17 = 6.00T$	$45.3 = 0.16T$	2.66	$1856.32 = 6.40T$	$67.2 = 0.23T$	3.94
300	$1799.57 = 6.00T$	$46.1 = 0.15T$	2.66	$1920.88 = 6.40T$	$68.4 = 0.23T$	3.95

Conjecture 3.2 *Given integers a and b with $0 < a, b < q$ and trace values c_k, c_ℓ, $c_{k-\ell}$, and $c_{k-2\ell}$, the trace value $c_{bk+a\ell}$ can on average be computed in about $6\log_2(\max(a, b))$ multiplications in $\mathbf{F}_p$ using Algorithm 3.1.*

It follows that XTR double exponentiation using Algorithm 3.1 is on average faster than the XTR single exponentiation from [10] (given in Algorithm 2.4), and more than twice as fast as the previous methods to compute $c_{bk+a\ell}$ ([10, Algorithm 2.4.8 and Theorem 2.4.9] and Algorithm 2.5). An additional advantage of Algorithm 3.1 is that, like Algorithm 2.5, it does not require matrices.

These advantages have considerable practical consequences, not only for the performance of XTR signature verification (Section 6), but also for the accessibility and ease of implementation of XTR. In Sections 4 and 5 consequences of Algorithm 3.1 for XTR single exponentiation are given.

Based on Table 1 the expected practical behavior of Algorithm 3.1 is well understood, and the practical merits of the method are beyond doubt. However, a satisfactory theoretical analysis of Algorithm 3.1, or the second degree original from [15], is still lacking. The iteration in Algorithm 3.1 is reminiscent of the binary and subtractive Euclidean greatest common divisor algorithms. Iterations of that sort typically exhibit an unpredictable behavior with a wide gap between worst and average case performance; see for instance [1,7,19] and the analysis attempts and open problems in [15].

This is further illustrated in Figure 1. There the average number of multiplications for $\lceil \log_2 t \rceil = 170$ is given as a function of the value of the constant in Steps 4(a)i and 4(b)i of Algorithm 3.1. The value 4 is close to optimal and convenient for implementation. However, it can be seen from Figure 1 that a value close to 4.8 is somewhat better, if one's sole objective is to minimize the number of multiplications in $\mathbf{F}_p$, as opposed to minimizing the overall runtime. The curves in Figure 1 were generated for constants ranging from 2 to 8 with stepsize 1/16, per constant averaged over the same collection of 2^{20} randomly selected t's, a's, and b's. The remarkable shape of the curves – both with at least four local minima – is a clear indication that the exact behavior of Algorithm 3.1 will be hard to analyse. It is of no immediate importance for the present paper and left as a subject for further study.

Remark 3.3 As shown in Appendix A other small improvements can be obtained by distinguishing more different cases than in Algorithm 3.1. The version presented above represents a good compromise that combines reasonable overhead with decent performance. In practical circumstances the performance of Algorithm 3.1 is on average close to optimal.

Remark 3.4 If Algorithm 3.1 is implemented using the slower field arithmetic from [10, Lemma 2.1.1], as opposed to the improved arithmetic from 2.1, it can on average be expected to require $7.4 \log_2(\max(a, b))$ multiplications in $\mathbf{F}_p$. This is still more than twice as fast as the method from [10] (using the slower arithmetic), but more than 20% slower than Conjecture 3.2.

Remark 3.5 Unlike the XTR exponentiation methods from [10], different instructions are carried out by Algorithm 3.1 for different input values. This makes Algorithm 3.1 inherently more vulnerable to environmental attacks than the methods from [10] (cf. [10, Remark 2.3.9]). If the possibility of such attacks is a concern, then utmost care should be taken while implementing Algorithm 3.1.

4 Single Exponentiation with Precomputation

Suppose that for a fixed c_1 several c_u's for different u's, with $0 < u < q$, have to be computed. In this section it is shown that, after a small amount of precom-

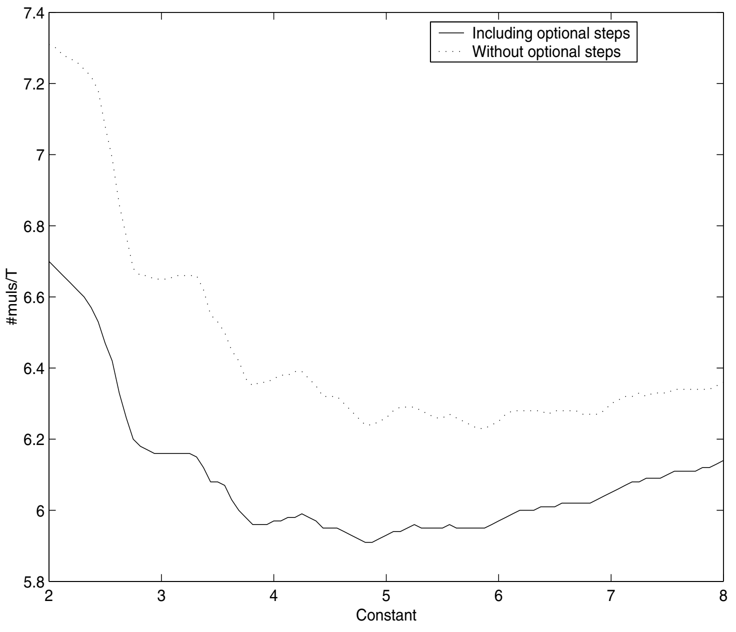

Fig. 1. Dependence on the value of the constant.

putation, this can be done using Algorithm 3.1 in less than half the number of multiplications in $\mathbf{F}_p$ that would be required by Algorithm 2.4.

Let $t = 2^{\lceil(\log_2 q)/2\rceil}$, and suppose that $S_{t-1} = (c_{t-2}, c_{t-1}, c_t)$ has been precomputed based on c_1. For any $u \in \{0, 1, \ldots, q-1\}$ non-negative integers a and b of at most $1 + (\log_2 q)/2$ bits can simply be computed such that $u = bt + a$. Given S_{t-1} and c_1, the value c_u can then be computed using Algorithm 3.1 with $k = t$ and $\ell = 1$. This leads to the following precomputation and XTR single exponentiation with precomputation.

4.1 Precomputation. Let c_1 be given. To precompute values t and $S_{t-1} = (c_{t-2}, c_{t-1}, c_t)$ do the following.

1. Let $t = 2^{\lceil(\log_2 q)/2\rceil}$, $v = (t-2)/2$, and let $v_{r-1}, v_{r-2}, \ldots, v_0$ be the binary representation of v (so $v_i = 1$ for $0 \le i < r$ for $t = 2^{\lceil\log_2 q)/2\rceil}$).
2. Apply Algorithm 2.4 to $S_1 = (3, c_1, c_1^2 - 2c_1^p)$, c_1, and $v_{r-1}, v_{r-2}, \ldots, v_0$ to compute $S_{2v+1} = S_{t-1}$.

The value S_{t-1} computed by Algorithm 4.1 consists of the traces of three consecutive powers of the subgroup generator corresponding to c_1. Algorithm 4.1 takes essentially a single application of Algorithm 2.4, and thus about $3.5 \log_2 q$ multiplications in $\mathbf{F}_p$, since $\log_2 t \approx (\log_2 q)/2$. Improved XTR single exponentiation Algorithm 5.1 given below would require more than a single application, because it produces just the trace of a single power, and not its two 'nearest neighbors' as well. With [11, Theorem 5.1], which for most t's allows fast computation of c_{t+1} given c_1, c_{t-1}, and c_t, two applications of Algorithm 5.1 would suffice. But that is still expected to be slower than a single application of Algorithm 2.4, as follows from Corollary 5.3.

4.2 XTR Single Exponentiation with Precomputation. Let u, c_1, t, and S_{t-1} be given, with $0 < u < q$. To compute c_u, do the following.

1. Compute non-negative integers a and b such that $u = bt + a \bmod q$ and a and b are at most about $\sqrt{q}$:
   – If $\log_2(t \bmod q) \approx (\log_2 q)/2$ (as in 4.1), then use long division to compute a and b such that $u = b(t \bmod q) + a$.
   – Otherwise, use the lattice-based method described in 4.4. With the proper choice of t this results in a and b that are small enough.
2. If $b = 0$, then compute $c_a = c_u$ using either Algorithm 2.4 or Algorithm 5.1, based on c_1.
3. Otherwise, if $a = 0$, then compute $\tilde{c}_b = c_{tb} = c_u$ using either Algorithm 2.4 or Algorithm 5.1, based on $\tilde{c}_1 = c_t$.
4. Otherwise, if $a \neq 0$ and $b \neq 0$, then do the following:
   – Let $k = t$, $\ell = 1$, so that $S_{t-1} = (c_{k-2\ell}, c_{k-\ell}, c_k)$ and $c_\ell = c_1$.
   – Use Algorithm 3.1 to compute $c_{bk+a\ell} = c_u$ based on a, b, c_k, c_ℓ, $c_{k-\ell}$, and $c_{k-2\ell}$.

Obviously, any t of about the same size as $\sqrt{q}$ will do. A power of 2, however, facilitates the computation of a and b in Step 1 of Algorithm 4.2. Algorithm 4.2 allows easy implementation and, apart from the precomputation, the performance overhead on top of the call to Algorithm 2.4, 5.1, or 3.1 is negligible. The expected runtime of Algorithm 4.2 follows from Conjecture 3.2.

Corollary 4.3 *Given integers u and t with $0 < u < q$ and $\log_2 t \approx (\log_2 q)/2$ and trace values c_1, c_t, c_{t-1}, and c_{t-2}, the trace value c_u can on average be computed in about $3\log_2 u$ multiplications in $\mathbf{F}_p$ using Algorithm 4.2.*

This is more than 60% faster than Algorithm 2.4 as described in [10] using the slower field arithmetic. It can be used in the first place by the owner of the XTR key containing c_1. Thus, XTR signature generation can on average be done more than 60% faster than before [10, Section 4.3]. It can also be used by shared users of an XTR key, such as in Diffie-Hellman key agreement. However, it only affects the first exponentiation to be carried out by each party: party A's computation of c_a given c_1 and a random a can be done on average more than 60% faster, but the computation of c_{ab} based on the value c_b received from party B is not affected by this method. See Section 5 how to speedup the computation of c_{ab} as well.

The precomputation scheme may also be useful for XTR-ElGamal encryption [10, Section 4.2]. In XTR-ElGamal encryption the public key contains two trace values, c_1 and c_k, where k is the secret key. The sender (who does not know k)

picks a random integer b, computes c_b based on c_1, computes c_{bk} based on c_k, uses c_{bk} to (symmetrically) encrypt the message, and sends the resulting encryption and c_b to the owner of k. If the sender uses XTR-ElGamal encryption more than once with the same c_1 and c_k, then it is advantageous to use precomputation. In this application *two* precomputations have to be carried out, once for c_1 and once for c_k. The recipient has to compute c_{bk} based on the value c_b received (and its secret k). Because c_b will not occur again, precomputation based on c_b does not make sense for the party performing XTR-ElGamal decryption.

4.4 Fast Precomputation. It is shown that the choice $t = p$ leads to a faster precomputation, while only marginally slowing down Step 1 of Algorithm 4.2. The triple $S_{p-1} = (c_{p-2}, c_{p-1}, c_p)$ follows from $c_p = c_1^p$ (Fact 2.3.1a), $c_{p-1} = c_1$ (because if g is a root with trace c_1, then $g^{p^2} = g^{p-1}$ is one of its conjugates and has the same trace), and from the fact that, according to [12, Proposition 5.7], c_{p-2} can be computed at the cost of a square-root computation in $\mathbf{F}_p$. Here it is assumed that the public key containing p, q, and c_1 contains an additional single bit of information to resolve the square-root ambiguity[1]. Thus, if $p \equiv 3 \bmod 4$ recipients of XTR public key data with p and q of the above form can do the precomputation of S_{p-1} at a cost of at most $\approx 1.3 \log_2 p$ multiplications in $\mathbf{F}_p$, assuming the owner of the key sends the required bit along. The storage overhead (on top of c_1) for S_{p-1} is just a single element of $\mathbf{F}_{p^2}$, as opposed to three elements for S_{t-1} as in 4.1.

If $p \bmod q \approx \sqrt{q}$, then non-negative a and b of order about $\sqrt{q}$ in Step 1 of Algorithm 4.2 can be found at the cost of a division with remainder. This is, for instance, the case if p and q are chosen as $r^2 + 1$ and $r^2 - r + 1$, respectively, as suggested in [10, Section 3.1]. However, usage of such primes p and q is not encouraged in [10] because of potential security hazards related to the use of primes p of a 'special form'.

Interestingly, and perhaps more surprisingly, sufficiently small a and b exist and can be found quickly in the general case as well. Let L be the two-dimensional integral lattice $\{(e_1, e_2)^T \in \mathbf{Z}^2 : e_1 + e_2 p \equiv 0 \bmod q\}$. If $(e_1, e_2)^T \in L$, then

$$(e_1 + e_2) - e_1 p \equiv -e_2 p + e_2 + e_2 p^2 = e_2(p^2 - p + 1) \equiv 0 \bmod q$$

so that $(e_1 + e_2, -e_1)^T \in L$. Let $v_1 = (e_1, e_2)^T$ be the shortest non-zero vector of L (using the L_2-norm). It may be assumed that $e_1 \geq 0$. It follows that $e_2 \geq 0$, because otherwise $(e_1+e_2, -e_1)^T$ or $(-e_2, e_1+e_2)^T \in L$ would be shorter than v_1. If v_2 is the shortest of $(e_1 + e_2, -e_1)^T, (-e_2, e_1 + e_2)^T \in L$, then $|v_2| < 2|v_1|$ and $\{v_1, v_2\}$ is easily seen to be a shortest basis for L, with $e_1^2 + e_1 e_2 + e_2^2 = q$ and $e_1, e_2 \leq \sqrt{q}$. This implies that given $\{v_1, v_2\}$ and any integer vector $(-u, 0)^T$, there is a vector $(a, b)^T$ with $0 \leq a, b \leq 2\sqrt{q}$ such that $(-u + a, b)^T \in L$. It follows that $-u + a + bp \equiv 0 \bmod q$, i.e., $u \equiv bp + a \bmod q$ as desired. Using the initial basis $\{(q, 0)^T, (-p, 1)^T\}$, the vector v_1 can be found quickly [3, Algorithm

[1] The statement in [12, Proposition 5.7] that this requires a square-root computation in $\mathbf{F}_{p^2}$, as opposed to $\mathbf{F}_p$, is incorrect. This follows immediately from the proof of [12, Proposition 5.7].

1.3.14], and for any u the vector $(a, b)^T$ can easily be computed. In [6, Section 4] a similar construction was independently developed for ECC scalar multiplication.

Corollary 4.5 *Given an integer u with $0 < u < q$ and trace values c_1 and c_{p-2}, the trace value c_u can on average be computed in about $3 \log_2 u$ multiplications in $\mathbf{F}_p$ using Algorithm 4.2.*

The owner of the key must explicitly compute c_{p-2} in order to compute the ambiguity-resolving bit. Thus, the owner cannot take advantage of fast precomputation. This adds a minor cost to the key creation.

5 Improved Single Exponentiation

In this section it is shown how Algorithm 3.1 can be used to obtain an XTR single exponentiation method that is on average more than 25% faster than Algorithm 2.4. That is 35% faster than the single exponentiation from [10] based on the slower field arithmetic. Using Algorithm 3.1 to obtain an on average faster XTR single exponentiation is straightforward: to compute c_u with $0 < u < q$ based on c_1 just apply Algorithm 3.1 to $k = \ell = 1$ and any positive a, b with $a + b = u$, then a speedup of more than 14% over Algorithm 2.4 can be expected according to Table 1.

The 25% faster method uses this same approach, but exploits the freedom of choice of a and b: if a and b, i.e., d and e in Algorithm 3.1, can be selected in such a way that the iteration in Step 4 of Algorithm 3.1 favors the 'cheap' steps, while still quickly decreasing d and e, then Algorithm 3.1 should run faster than for randomly selected a and b. Given the various substeps of Step 4 of Algorithm 3.1 and the associated costs, a good way to split up u in the sum of positive a and b seems to be such that b/a is close to the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$, i.e., the asymptotic ratio between two consecutive Fibonacci numbers. This can be seen as follows. If the initial ratio between d and e is close to ϕ, then Step 4(a)i applies and d, e is replaced by $e, d - e$. This corresponds to a 'Fibonacci-step back' so that the ratio between the new d and e (i.e., e and $d - e$) can again be expected to be close to ϕ. Furthermore, the sum of d and e is reduced by a factor ϕ, which is a relatively good drop compared to the low cost of Step 4(a)i (namely, three multiplications in $\mathbf{F}_p$). This leads to the following improved XTR single exponentiation.

5.1 Improved XTR Single Exponentiation. Let u and c_1 be given, with $0 < u < q$. To compute c_u, do the following.

1. Let $a = \text{round}(\frac{3-\sqrt{5}}{2}u)$ and $b = u - a$ (where round(x) is the integer closest to x). As a result $b/a \approx \phi$ as above.
2. Let $k = \ell = 1$, $c_k = c_\ell = c_1$, $c_{k-\ell} = c_0 = 3$, $c_{k-2\ell} = c_{-1} = c_1^p$ (cf. Fact 2.3.1a).
3. Apply Algorithm 3.1 to a, b, c_k, c_ℓ, $c_{k-\ell}$, and $c_{k-2\ell}$, resulting in $c_{bk+a\ell} = c_u$.

Proposition 5.2 *In the call to Algorithm 3.1 in Step 3 of Algorithm 5.1, the values of d and e in Step 4 of Algorithm 3.1 are reduced to approximately half their original sizes using a sequence of approximately $\log_\phi \sqrt{u}$ iterations using just Step 4(a)i.*

Proof. Let $m = \text{round}(\log_\phi u)$. Asymptotically for $m \to \infty$ the values a and b in Algorithm 5.1 satisfy $b/a = \phi + \epsilon_1$ with $|\epsilon_1| = O(2^{-m})$. Furthermore, for $n \to \infty$, the n-th Fibonacci number F_n satisfies $\frac{F_n}{F_{n-1}} = \phi + \epsilon_2$ with $|\epsilon_2| = O(2^{-n})$. It follows that $a = \frac{F_{m-1}}{F_m} b + \epsilon_3$, where $|\epsilon_3|$ is bounded by a small positive constant.

Define $(d_0, e_0) = (b, a)$ and $(d_i, e_i) = (e_{i-1}, d_{i-1} - e_{i-1})$ for $i > 0$. With induction it follows from $a = \frac{F_{m-1}}{F_m} b + \epsilon_3$ that

$$d_i = \frac{F_{m-i}}{F_m} b - (-1)^i F_i \epsilon_3 \tag{2}$$

for $0 \leq i < m$. Algorithm 3.1 as called from Algorithm 5.1 will perform Fibonacci steps as long as $e_i < d_i < 2e_i$. But as soon as $d_i > 2e_i$ this nice behavior will be lost. From $e_i = d_{i+1}$ and (2) it follows that $d_i > 2e_i$ is equivalent to

$$\frac{F_{m-i-3}}{F_m} b < (-1)^{i-1} F_{i+3} \epsilon_3.$$

Because F_m/b and $|\epsilon_3|$ are both bounded by small positive constants, the first time this condition will hold is when F_{m-i-3} and F_{i+3} are of the same order of magnitude, i.e., $m - i - 3 \approx i + 3$. Thus, the Fibonacci behavior is lost after about $m/2 = \log_\phi \sqrt{u}$ iterations, at which point $d_i \approx \sqrt{u}$ (this follows from (2)). This completes the proof of Proposition 5.2.

Based on Proposition 5.2, a heuristic average runtime analysis of Algorithm 5.1 follows easily. The Fibonacci part consists of about $\log_\phi \sqrt{u}$ iterations consisting of just Step 4(a)i of Algorithm 3.1, at a total cost of $3 \log_\phi \sqrt{u} \approx 2.2 \log_2 u$ multiplications in $\mathbf{F}_p$. Once the Fibonacci behavior is lost, the remaining d and e are assumed to behave as random integers of about the same order of magnitude as $\sqrt{u}$, so that, according to Conjecture 3.2, the remainder can on average be expected to take about $6 \log_2 \sqrt{u} = 3 \log_2 u$ multiplications in $\mathbf{F}_p$.

Corollary 5.3 *Given an integer u with $0 < u < q$ and a trace value c_1, the trace value c_u can on average be computed in about $5.2 \log_2 u$ multiplications in $\mathbf{F}_p$ using Algorithm 5.1.*

This corresponds closely to the actual practical runtimes. It is more than 25% better than Algorithm 2.4. Without the optional steps in Algorithm 3.1 the speedup is reduced to about 22%.

Remark 5.4 If insufficient precision is used in the computation of a and b in Step 1 of Algorithm 5.1, then ϵ_3 in the proof of Proposition 5.2 is no longer bounded by a small constant. It follows that $d_i > 2e_i$ already holds for a smaller value of i, implying that the Fibonacci behavior is lost earlier. A precise analysis of the expected performance degradation as a function of the lack of precision is straightforward. In practice this effect is very noticeable.

If a and b happen to be such that all steps are Fibonacci steps, then the cost would be $4.3 \log_2 u$. This is fewer than $\log_2 u$ multiplications in $\mathbf{F}_p$ better than the average behavior obtained.

6 Timings

To make sure that the methods introduced in this paper actually work, and to discover their runtime characteristics, all new methods were implemented and tested. In this section the results are reported, in such a way that the results can easily and meaningfully be compared to the timings reported in [10].

Algorithm 2.5 was implemented, tested for correctness, and it was confirmed that the speedup over the double exponentiation from [10] is negligible. However, implementing Algorithm 2.5 was shown to be significantly easier than it was for the matrix-based method from [10]. Thus, Algorithm 2.5 may still turn out to be valuable if Algorithm 3.1 cannot be used (Remark 3.5).

The methods from Sections 3, 4, and 5 were implemented as well, and incorporated in cryptographic XTR applications along with the old methods from [10]. The resulting runtimes are reported in Table 2. Each runtime is averaged over 100 random keys and 100 cryptographic applications (on randomly selected data) per key. The timings for the XTR single exponentiations with precomputation do not include the time needed for the precomputations. The latter are given in the last two rows. All times are in milliseconds on a 600 MHz Pentium III NT laptop, and are based on the use of a generic and not particularly fast software package for extended precision integer arithmetic [8]. More careful implementation should result in much faster timings. The point of Table 2 is however not the absolute speed, but the relative speedup over the methods from [10].

The RSA timings are included to allow a meaningful interpretation of the timings: if the RSA signing operation runs x times faster using one's own software and platform, then most likely XTR will also run x times faster compared to the figures in Table 2. For each key an odd 32-bit RSA public exponent was randomly selected. 'CRT' stands for 'Chinese Remainder Theorem'. For a theoretical comparison of the runtimes of RSA, XTR, ECC, and various other public key systems at several security levels, refer to [9].

Table 2. RSA, old XTR, and new XTR runtimes.

method		key selection	signing	verifying	encrypting	decrypting
1020-bit RSA	with CRT	908 ms	40 ms	5 ms	5 ms	40 ms
	without CRT		123 ms			123 ms
170-bit XTR	old	64 ms	10 ms	21 ms	21 ms	10 ms
	new, no precomputation	62 ms	7.3 ms	8.6 ms	15 ms	7.3 ms
	new, with precomputation		4.3 ms		8.6 ms	
	precomputation 4.1		4.4 ms		8.8 ms	
	fast precomputation 4.4		1.6 ms		6.0 ms	

7 Application to LUC and ECC

The exponentiations in LUC [18] and ECC when using the curve parameterization proposed in [14] can be evaluated using second degree recurrences. For LUC this is described in detail in [15]. For ECC it is described in [16] and follows by combining [14] and [15]. For ease of reference the resulting runtimes are summarized in this section.

7.1 LUC. Let p and q be primes such that q divides $p+1$, and let g be a generator of the order q subgroup of $\mathbf{F}_{p^2}^*$. In LUC elements of $\langle g \rangle$ are represented by their trace over $\mathbf{F}_p$. Let $v_n \in \mathbf{F}_p$ denote the trace over $\mathbf{F}_p$ of g^n.

Conjecture 7.2 *(cf. Conjecture 3.2) Given integers a and b with $0 < a, b < q$ and trace values v_k, v_ℓ, and $v_{k-\ell}$, the trace value $v_{bk+a\ell}$ can on average be computed in about $1.49 \log_2(\max(a, b))$ multiplications and $0.33 \log_2(\max(a, b))$ squarings in $\mathbf{F}_p$, using the method implied by [15, Table 4].*

Corollary 7.3 *(cf. Corollary 4.3) Given integers u and t with $0 < u < q$ and $\log_2 t \approx (\log_2 q)/2$ and trace values v_1, v_t, and v_{t-1}, the trace value v_u can on average be computed in about $0.75 \log_2 u$ multiplications and $0.17 \log_2 u$ squarings in $\mathbf{F}_p$ using a generalization of Algorithm 4.2.*

Corollary 7.4 *(cf. Corollary 5.3) Given an integer u with $0 < u < q$ and a trace value v_1, the trace value v_u can on average be computed in about $1.47 \log_2 u$ multiplications and $0.17 \log_2 u$ squarings in $\mathbf{F}_p$ using a generalization of Algorithm 5.1.*

7.5 ECC. Let E be an elliptic curve over a prime field $\mathbf{F}_p$, let $E(\mathbf{F}_p)$ be the group of points of E over $\mathbf{F}_p$, and let $G \in E(\mathbf{F}_p)$ be a point of prime order q. As usual, the group operation in $E(\mathbf{F}_p)$ is written additively.

Conjecture 7.6 *(cf. Conjecture 3.2) Given integers a and b with $0 < a, b < q$ and points kG, ℓG, and $(k-\ell)G$, the x-coordinate of the point $(bk + a\ell)G$ can on average be computed in approximately $7 \log_2(\max(a, b))$ multiplications and $3.7 \log_2(\max(a, b))$ squarings in $\mathbf{F}_p$, using the method implied by [15, Table 4] combined with the elliptic curve parameterization from [14].*

Corollary 7.7 *(cf. Corollary 4.3) Given integers u and t with $0 < u < q$ and $\log_2 t \approx (\log_2 q)/2$ and points G, tG, and $(t-1)G$, the x-coordinate of the point uG can on average be computed in about $3.5 \log_2 u$ multiplications and $1.8 \log_2 u$ squarings in $\mathbf{F}_p$ using a generalization of Algorithm 4.2.*

Corollary 7.8 *(cf. Corollary 5.3) Given an integer u with $0 < u < q$ and a point G, the x-coordinate of the point uG can on average be computed in about $6.4 \log_2 u$ multiplications and $3.3 \log_2 u$ squarings in $\mathbf{F}_p$ using a generalization of Algorithm 5.1.*

The single scalar multiplication algorithms are competitive with the ones described in the literature [5]. The double scalar multiplication algorithm from [16] (and as slightly adapted to obtain Conjecture 7.6) is substantially better than other ECC double scalar multiplication methods reported in the literature [2]. For appropriate elliptic curves Corollary 7.7 can be combined with the method proposed in [6], so that the runtime of Corollary 7.7 would hold for Corollary 7.8.

8 Conclusion

The XTR public key system as published in [10] is one of the fastest, most compact, and easiest to implement public key systems. In this paper it is shown that it is even faster and easier to implement than originally believed. The matrices from [10] can be replaced by the more general iteration from Section 3. This results in 60% faster XTR signature applications, substantially faster encryption, decryption, and key agreement applications, and more compact implementations.

Acknowledgment. The authors thank Peter Montgomery from Microsoft Research whose remarks [16] stimulated this research.

References

1. E. Bach, J. Shallit, *Algorithmic Number Theory*, The MIT Press, 1996.
2. M. Brown, D. Hankerson, J. López, A. Menezes, *Software implementation of the NIST elliptic curves over prime fields*, Proceedings RSA Conference 2001, LNCS 2020, Springer-Verlag 2001, 250-265.
3. H. Cohen, *A course in computational algebraic number theory*, GTM 138, Springer-Verlag 1993.
4. H. Cohen, A.K. Lenstra, *Implementation of a new primality test*, Math. Comp. 48 (1987) 103-121.
5. H. Cohen, A. Miyaji, T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates*, Proceedings Asiacrypt'98, LNCS 1514, Springer-Verlag 1998, 51-65.
6. R.P. Gallant, R.J. Lambert, S.A. Vanstone, *Faster point multiplication on elliptic curves with efficient endomorphisms*, Proceedings Crypto 2001, LNCS 2139, Springer-Verlag 2001, 190-200.
7. D.E. Knuth, *The art of computer programming, Volume 2, Seminumerical Algorithms*, third edition, Addison-Wesley, 1998.
8. A.K. Lenstra, *The long integer package FREELIP*, available from www.ecstr.com.
9. A.K. Lenstra, *Unbelievable security: matching AES security using public key systems*, Proceedings Asiacrypt 2001, Springer-Verlag 2001, this volume.
10. A.K. Lenstra, E.R. Verheul, *The XTR public key system*, Proceedings of Crypto 2000, LNCS 1880, Springer-Verlag 2000, 1-19; available from www.ecstr.com.
11. A.K. Lenstra, E.R. Verheul, *Key improvements to XTR*, Proceedings of Asiacrypt 2000, LNCS 1976, Springer-Verlag 2000, 220-233; available from www.ecstr.com.
12. A.K. Lenstra, E.R. Verheul, *Fast irreducibility and subgroup membership testing in XTR*, Proceedings PKC 2001, LNCS 1992, Springer-Verlag 2001, 73-86; available from www.ecstr.com.

13. P.L. Montgomery, *Modular multiplication without trial division*, Math. Comp. 44 (1985) 519-521.
14. P.L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Math. Comp. 48 (1987) 243-264.
15. P.L. Montgomery, *Evaluating recurrences of form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas chains,* January 1992; ftp.cwi.nl: /pub/pmontgom/Lucas.pz.gz.
16. P.L. Montgomery, Private communication: *expon2.txt, Dual elliptic curve exponentiation*, manuscript, Microsoft Research, August 2000.
17. S.C. Pohlig, M.E. Hellman, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance*, IEEE Trans. on IT, 24 (1978), 106-110.
18. P. Smith, C. Skinner, *A public-key cryptosystem and a digital signature system based on the Lucas function analogue to discrete logarithms*, Proceedings of Asiacrypt '94, LNCS 917, Springer-Verlag 1995, 357-364.
19. B. Vallée, *Dynamics of the binary Euclidean algorithm: functional analysis and operators*, Algorithmica 22 (1998), 660-685; and other related papers available from www.users.info-unicaen.fr/~brigitte/Publications/.

A Further Improved Double Exponentiation

Almost 2% can be saved compared to Algorithm 3.1 by distinguishing more cases in Step 4. This is done by replacing Step 4 of Algorithm 3.1 by the following:

```
4. As long as $d \neq e$ replace $(d, e, u, v, c_u, c_v, c_{u-v}, c_{u-2v})$ by the 8-tuple given below.
   a) If $d > e$ then
      i. if $d \leq 5.5e$, then $(e, d-e, u+v, u, c_{u+v}, c_u, c_v, c_{v-u})$.
      ii. else if $d$ and $e$ are odd, then $(\frac{d-e}{2}, e, 2u, u+v, c_{2u}, c_{u+v}, c_{u-v}, c_{-2v})$.
      iii. else if $d \leq 6.4e$, then $(e, d-e, u+v, u, c_{u+v}, c_u, c_v, c_{v-u})$.
      iv. else if $d \equiv e \bmod 3$, then $(\frac{d-e}{3}, e, 3u, u+v, c_{3u}, c_{u+v}, c_{2u-v}, c_{u-2v})$.
      v. else if $d$ is even, then $(\frac{d}{2}, e, 2u, v, c_{2u}, c_v, c_{2u-v}, c_{2(u-v)})$.
      vi. else if $d \leq 7.5e$, then $(e, d-e, u+v, u, c_{u+v}, c_u, c_v, c_{v-u})$.
      vii. else if $de \equiv 2 \bmod 3$, then $(\frac{d-2e}{3}, e, 3u, 2u+v, c_{3u}, c_{2u+v}, c_{u-v}, c_{-u-2v})$.
      viii. else ($e$ is even), then $(\frac{e}{2}, d, 2v, u, c_{2v}, c_u, c_{2v-u}, c_{2(v-u)})$.
   b) Else (if $e > d$)
      i. if $e \leq 5.5d$, then $(d, e-d, u+v, v, c_{u+v}, c_v, c_u, c_{u-v})$.
      ii. else if $e$ is even, then $(\frac{e}{2}, d, 2v, u, c_{2v}, c_u, c_{2v-u}, c_{2(v-u)})$.
      iii. else if $e \equiv d \bmod 3$, then $(\frac{e-d}{3}, d, 3v, u+v, c_{3v}, c_{u+v}, c_{2v-u}, c_{v-2u})$.
      iv. else if $de \equiv 2 \bmod 3$, then $(d, \frac{e-2d}{3}, u+2v, 3v, c_{u+2v}, c_{3v}, c_{u-v}, c_{u-4v})$.
      v. else if $e \leq 7.4d$, then $(d, e-d, u+v, v, c_{u+v}, c_v, c_u, c_{u-v})$.
      vi. else if $d$ is odd, then $(\frac{e-d}{2}, d, 2v, u+v, c_{2v}, c_{u+v}, c_{v-u}, c_{-2u})$.
      vii. else if $e \equiv 0 \bmod 3$, then $(\frac{e}{3}, d, 3v, u, c_{3v}, c_u, c_{3v-u}, c_{3v-2u})$.
      viii. else ($d$ is even), then $(\frac{d}{2}, e, 2u, v, c_{2u}, c_v, c_{2u-v}, c_{2(u-v)})$.
```

Steps 4(a)vii and 4(b)iv require 13.5 and 12.5 multiplications in $\mathbf{F}_p$, respectively. The cost of the other steps is as in Section 3. The average cost to compute $c_{bk+a\ell}$ turns out to be about $5.9 \log_2(\max(a, b))$ multiplications in $\mathbf{F}_p$. Omission of Steps 4(a)iii, 4(a)vi, and 4(b)v, combined with a constant 4 instead of 5.5 in Steps 4(a)i and 4(b)i leads to an almost 1% speedup over Algorithm 3.1.