To appear in *Advances in Cryptology—Crypto 2000, Lecture Notes in Computer Science 1880*, Springer-Verlag 2000, 1-20.

The XTR public key system

Arjen K. Lenstra[1], Eric R. Verheul[2]

[1] Citibank, N.A., 1 North Gate Road, Mendham, NJ 07945-3104, U.S.A.,
arjen.lenstra@citicorp.com
[2] PricewaterhouseCoopers, GRMS Crypto Group, Goudsbloemstraat 14, 5644 KE Eindhoven, The Netherlands,
Eric.Verheul@[nl.pwcglobal.com, pobox.com]

Abstract. This paper introduces the XTR public key system. XTR is based on a new method to represent elements of a subgroup of a multiplicative group of a finite field. Application of XTR in cryptographic protocols leads to substantial savings both in communication and computational overhead without compromising security.

1 Introduction

The Diffie-Hellman (DH) key agreement protocol was the first published practical solution to the key distribution problem, allowing two parties that have never met to establish a shared secret key by exchanging information over an open channel. In the basic DH scheme the two parties agree upon a generator g of the multiplicative group $\mathrm{GF}(p)^*$ of a prime field $\mathrm{GF}(p)$ and they each send a random power of g to the other party. Assuming both parties know p and g, each party transmits about $\log_2(p)$ bits to the other party.

In [7] it was suggested that finite extension fields can be used instead of prime fields, but no direct computational or communication advantages were implied. In [22] a variant of the basic DH scheme was introduced where g generates a relatively small subgroup of $\mathrm{GF}(p)^*$ of prime order q. This considerably reduces the computational cost of the DH scheme, but has no effect on the number of bits to be exchanged. In [3] it was shown for the first time how the use of finite extension fields and subgroups can be combined in such a way that the number of bits to be exchanged is reduced by a factor 3. More specifically, it was shown that elements of an order q subgroup of $\mathrm{GF}(p^6)^*$ can be represented using $2\log_2(p)$ bits if q divides $p^2 - p + 1$. Despite its communication efficiency, the method of [3] is rather cumbersome and computationally not particularly efficient.

In this paper we present a greatly improved version of the method from [3] that achieves the same communication advantage at a much lower computational cost. We refer to our new method as XTR, for Efficient and Compact Subgroup Trace Representation. XTR can be used in conjunction with any cryptographic protocol that is based on the use of subgroups and leads to substantial savings in communication and computational overhead. Furthermore, XTR key generation is very simple. We prove that using XTR in cryptographic protocols does not affect their security. The best attacks we are aware of are Pollard's rho method in the order q subgroup, or the Discrete Logarithm variant of the Number Field Sieve in the full multiplicative group $\mathrm{GF}(p^6)^*$. With primes p and q of about

$1024/6 \approx 170$ bits the security of XTR is equivalent to traditional subgroup systems using 170-bit subgroups and 1024-bit finite fields. But with XTR subgroup elements can be represented using only about $2 * 170$ bits, which is substantially less than the 1024-bits required for their traditional representation.

The amount of computation required by a full exponentiation using XTR is less than the time required by a full scalar multiplication in an Elliptic Curve cryptosystem (ECC) over a 170-bit prime field, and thus substantially less than the time required by a full exponentiation in either RSA or traditional subgroup discrete logarithm systems of equivalent security. XTR achieves security similar to RSA for much smaller key sizes than RSA. Although ECC key sizes can be somewhat further reduced than XTR key sizes, in many circumstances (e.g. storage), key sizes of ECC and XTR will be comparable. However, XTR is not affected by the uncertainty still marring ECC security. Key selection for XTR is very fast compared to RSA, and orders of magnitude easier and faster than for ECC. As a result XTR may be regarded as the best of two worlds, RSA and ECC. It is an excellent alternative to either RSA or ECC in applications such as SSL/TLS (Secure Sockets Layer, Transport Layer Security), public key smartcards, WAP/WTLS (Wireless Application Protocol, Wireless Transport Layer Security), IPSEC/IKE (Internet Protocol Security, Internet Key Exchange), and SET (Secure Electronic Transaction).

In [14] it is argued that ECC is the only public key system that is suitable for a variety of environments, including low-end smart cards and over-burdened web servers communicating with powerful PC clients. XTR shares this advantage with ECC, with the distinct additional advantage that XTR key selection is very easy. This makes it easily feasible for all users of XTR to have public keys that are not shared with others, unlike ECC where a large part of the public key is often shared between all users of the system. Also, compared to ECC, the mathematics underlying XTR is straightforward, thus avoiding two common ECC-pitfalls: ascertaining that unfortunate parameter choices are avoided that happen to render the system less secure, and keeping abreast of, and incorporating additional checks published in, newly obtained results. The latest example of the latter is [8], where yet another condition affecting the security of ECC over finite fields of characteristic two is described. As a consequence the ECC implementation of the draft Internet Key Exchange protocol (part of IPSec) had to be revised. It may be due to such examples that A.M. Odlyzko advises to use ECC key sizes of at least 300 bits in [16], even for moderate security needs.

XTR is the first method we are aware of that uses $\mathrm{GF}(p^2)$ arithmetic to achieve $\mathrm{GF}(p^6)$ security, without requiring explicit construction of $\mathrm{GF}(p^6)$. Let g be an element of order $q > 6$ dividing $p^2 - p + 1$. Because $p^2 - p + 1$ divides the order $p^6 - 1$ of $\mathrm{GF}(p^6)^*$ this g generates an order q subgroup of $\mathrm{GF}(p^6)^*$. Since q does not divide any $p^s - 1$ for $s = 1, 2, 3$ (cf. [11]), the subgroup generated by g cannot be embedded in the multiplicative group of any true subfield of $\mathrm{GF}(p^6)$. We show, however, that arbitrary powers of g can be represented using a single element of the subfield $\mathrm{GF}(p^2)$, and that such powers can be computed efficiently using arithmetic operations in $\mathrm{GF}(p^2)$ while avoiding arithmetic in $\mathrm{GF}(p^6)$.

In Section 2 we describe XTR, and in Section 3 we explain how the XTR parameters can be found quickly. Applications and comparisons to RSA and ECC are given in Section 4. In Section 5 we prove that using XTR does not have a negative impact on the security.

2 Subgroup representation and arithmetic

2.1 Preliminaries

Let $p \equiv 2 \bmod 3$ be a prime such that the sixth cyclotomic polynomial evaluated in p, i.e., $\phi_6(p) = p^2 - p + 1$, has a prime factor $q > 6$. In subsection 3.1 we give a fast method to select p and q. By g we denote an element of $\mathrm{GF}(p^6)^*$ of order q. Because of the choice of q, this g is not contained in any proper subfield of $\mathrm{GF}(p^6)$ (cf. [11]). Many cryptographic applications (cf. Section 4) make use of the subgroup $\langle g \rangle$ generated by g. In this section we show that actual representation of the elements of $\langle g \rangle$ and of any other element of $\mathrm{GF}(p^6)$ can be avoided. Thus, there is no need to represent elements of $\mathrm{GF}(p^6)$, for instance by constructing a sixth or third degree irreducible polynomial over $\mathrm{GF}(p)$ or $\mathrm{GF}(p^2)$, respectively. A representation of $\mathrm{GF}(p^2)$ is needed, however. This is done as follows.

From $p \equiv 2 \bmod 3$ it follows that $p \bmod 3$ generates $\mathrm{GF}(3)^*$, so that the zeros α and α^p of the polynomial $(X^3 - 1)/(X - 1) = X^2 + X + 1$ form an optimal normal basis for $\mathrm{GF}(p^2)$ over $\mathrm{GF}(p)$. Because $\alpha^i = \alpha^{i \bmod 3}$, an element $x \in \mathrm{GF}(p^2)$ can be represented as $x_1\alpha + x_2\alpha^p = x_1\alpha + x_2\alpha^2$ for $x_1, x_2 \in \mathrm{GF}(p)$. In this representation of $\mathrm{GF}(p^2)$ an element t of $\mathrm{GF}(p)$ is represented as $-t\alpha - t\alpha^2$, e.g. 3 is represented as $-3\alpha - 3\alpha^2$. Arithmetic operations in $\mathrm{GF}(p^2)$ are carried out as follows.

For any $x = x_1\alpha + x_2\alpha^2 \in \mathrm{GF}(p^2)$ we have that $x^p = x_1^p\alpha^p + x_2^p\alpha^{2p} = x_2\alpha + x_1\alpha^2$. It follows that p^{th} powering in $\mathrm{GF}(p^2)$ does not require arithmetic operations and can thus be considered to be for free. Squaring of $x_1\alpha + x_2\alpha^2 \in \mathrm{GF}(p^2)$ can be carried out at the cost of two squarings and a single multiplication in $\mathrm{GF}(p)$, where as customary we do not count the cost of additions in $\mathrm{GF}(p)$. Multiplication in $\mathrm{GF}(p^2)$ can be done using four multiplications in $\mathrm{GF}(p)$. These straightforward results can simply be improved to three squarings and three multiplications, respectively, by using a Karatsuba-like approach (cf. [10]): to compute $(x_1\alpha + x_2\alpha^2) * (y_1\alpha + y_2\alpha^2)$ one computes $x_1 * y_1$, $x_2 * y_2$, and $(x_1 + x_2) * (y_1 + y_2)$, after which $x_1 * y_2 + x_2 * y_1$ follows using two subtractions. Furthermore, from $(x_1\alpha + x_2\alpha^2)^2 = x_2(x_2 - 2x_1)\alpha + x_1(x_1 - 2x_2)\alpha^2$ it follows that squaring in $\mathrm{GF}(p^2)$ can be done at the cost of two multiplications in $\mathrm{GF}(p)$. Under the reasonable assumption that a squaring in $\mathrm{GF}(p)$ takes 80% of the time of a multiplication in $\mathrm{GF}(p)$ (cf. [4]), two multiplications is faster than three squarings. Finally, to compute $x * z - y * z^p \in \mathrm{GF}(p^2)$ for $x, y, z \in \mathrm{GF}(p^2)$ four multiplications in $\mathrm{GF}(p)$ suffice, because, with $x = x_1\alpha + x_2\alpha^2$, $y = y_1\alpha + y_2\alpha^2$, and $z = z_1\alpha + z_2\alpha^2$, it is easily verified that $x * z - y * z^p = (z_1(y_1 - x_2 - y_2) + z_2(x_2 - x_1 + y_2))\alpha + (z_1(x_1 - x_2 + y_1) + z_2(y_2 - x_1 - y_1))\alpha^2$. Thus we have the following.

Lemma 2.1.1 *Let $x, y, z \in \mathrm{GF}(p^2)$ with $p \equiv 2 \bmod 3$.*

i. Computing x^p is for free.
ii. Computing x^2 takes two multiplications in $\mathrm{GF}(p)$.
iii. Computing $x * y$ takes three multiplications in $\mathrm{GF}(p)$.
iv. Computing $x * z - y * z^p$ takes four multiplications in $\mathrm{GF}(p)$.

For comparison purposes we review the following well known results.

Lemma 2.1.2 *Let $x, y, z \in \mathrm{GF}(p^6)$ with $p \equiv 2 \bmod 3$, and let $a, b \in \mathbf{Z}$ with $0 < a, b < p$. Assume that a squaring in $\mathrm{GF}(p)$ takes 80% of the time of a multiplication in $\mathrm{GF}(p)$ (cf. [4]).*

i. Computing x^2 takes 14.4 multiplications in $\mathrm{GF}(p)$.
ii. Computing $x * y$ takes 18 multiplications in $\mathrm{GF}(p)$.
iii. Computing x^a takes an expected $23.4 \log_2(a)$ multiplications in $\mathrm{GF}(p)$.
iv. Computing $x^a * y^b$ takes an expected $27.9 \log_2(\max(a, b))$ multiplications in $\mathrm{GF}(p)$.

Proof. Since $p \equiv 2$ mod 3, $\mathrm{GF}(p^6)$ can be represented using an optimal normal basis over $\mathrm{GF}(p)$ so that the 'reduction' modulo the minimal polynomial does not require any multiplications in $\mathrm{GF}(p)$. Squaring and multiplication in $\mathrm{GF}(p^6)$ can then be done in 18 squarings and multiplications in $\mathrm{GF}(p)$, respectively, from which *i* and *ii* follow. For *iii* we use the ordinary square and multiply method, so we get $\log_2(a)$ squarings and an expected $0.5 \log_2(a)$ multiplications in $\mathrm{GF}(p^6)$. For *iv* we use standard multi-exponentiation, which leads to $\log_2(\max(a, b))$ squarings and $0.75 \log_2(\max(a, b))$ multiplications in $\mathrm{GF}(p^6)$.

2.2 Traces

The *conjugates* over $\mathrm{GF}(p^2)$ of $h \in \mathrm{GF}(p^6)$ are h, h^{p^2}, and h^{p^4}. The *trace* $Tr(h)$ over $\mathrm{GF}(p^2)$ of $h \in \mathrm{GF}(p^6)$ is the sum of the conjugates over $\mathrm{GF}(p^2)$ of h, i.e., $Tr(h) = h + h^{p^2} + h^{p^4}$. Because the order of $h \in \mathrm{GF}(p^6)^*$ divides $p^6 - 1$, i.e., $p^6 \equiv 1$ modulo the order of h, we have that $Tr(h)^{p^2} = Tr(h)$, so that $Tr(h) \in \mathrm{GF}(p^2)$. For $h_1, h_2 \in \mathrm{GF}(p^6)$ and $c \in \mathrm{GF}(p^2)$ we have that $Tr(h_1 + h_2) = Tr(h_1) + Tr(h_2)$ and $Tr(c * h_1) = c * Tr(h_1)$. That is, the trace over $\mathrm{GF}(p^2)$ is $\mathrm{GF}(p^2)$-linear. Unless specified otherwise, conjugates and traces in this paper are over $\mathrm{GF}(p^2)$.

The conjugates of g of order dividing $p^2 - p + 1$ are g, g^{p-1} and g^{-p} because $p^2 \equiv p - 1 \bmod p^2 - p + 1$ and $p^4 \equiv -p \bmod p^2 - p + 1$.

Lemma 2.2.1 *The roots of $X^3 - Tr(g)X^2 + Tr(g)^p X - 1$ are the conjugates of g.*

Proof. We compare the coefficients of $X^3 - Tr(g)X^2 + Tr(g)^p X - 1$ with the coefficients of the polynomial $(X - g)(X - g^{p-1})(X - g^{-p})$. The coefficient of X^2 follows from $g + g^{p-1} + g^{-p} = Tr(g)$, and the constant coefficient from $g^{1+p-1-p} = 1$. The coefficient of X equals $g * g^{p-1} + g * g^{-p} + g^{p-1} * g^{-p} = g^p + g^{1-p} + g^{-1}$. Because $1 - p \equiv -p^2 \bmod p^2 - p + 1$ and $-1 \equiv p^2 - p \bmod p^2 - p + 1$, we find that $g^p + g^{1-p} + g^{-1} = g^p + g^{-p^2} + g^{p^2-p} = (g + g^{-p} + g^{p-1})^p = Tr(g)^p$, which completes the proof.

Similarly (and as proved below in Lemma 2.3.4.*ii*), the roots of $X^3 - Tr(g^n)X^2 + Tr(g^n)^pX - 1$ are the conjugates of g^n. Thus, the conjugates of g^n are fully determined by $X^3 - Tr(g^n)X^2 + Tr(g^n)^pX - 1$ and thus by $Tr(g^n)$. Since $Tr(g^n) \in \mathrm{GF}(p^2)$ this leads to a compact representation of the conjugates of g^n. To be able to use this representation in an efficient manner in cryptographic protocols, we need an efficient way to compute $Tr(g^n)$ given $Tr(g)$. Such a method can be derived from properties of g and the trace function. However, since we need a similar method in a more general context in Section 3, we consider the properties of the polynomial $X^3 - cX^2 + c^pX - 1$ for general $c \in \mathrm{GF}(p^2)$ (as opposed to c's that are traces of powers of g).

2.3 The polynomial $F(c, X)$

Definition 2.3.1 For $c \in \mathrm{GF}(p^2)$ let $F(c, X)$ be the polynomial $X^3 - cX^2 + c^pX - 1 \in \mathrm{GF}(p^2)[X]$ with (not necessarily distinct) roots h_0, h_1, h_2 in $\mathrm{GF}(p^6)$, and let $\tau(c, n) = h_0^n + h_1^n + h_2^n$ for $n \in \mathbf{Z}$. We use the shorthand $c_n = \tau(c, n)$.

In this subsection we derive some properties of $F(c, X)$ and its roots.

Lemma 2.3.2
i. $c = c_1$.
ii. $h_0 * h_1 * h_2 = 1$.
iii. $h_0^n * h_1^n + h_0^n * h_2^n + h_1^n * h_2^n = c_{-n}$ *for* $n \in \mathbf{Z}$.
iv. $F(c, h_j^{-p}) = 0$ *for* $j = 0, 1, 2$.
v. $c_{-n} = c_{np} = c_n^p$ *for* $n \in \mathbf{Z}$.
vi. Either all h_j *have order dividing* $p^2 - p + 1$ *and* > 3 *or all* $h_j \in \mathrm{GF}(p^2)$.
vii. $c_n \in \mathrm{GF}(p^2)$ *for* $n \in \mathbf{Z}$.

Proof. The proofs of *i* and *ii* are immediate and *iii* follows from *ii*. From $F(c, h_j) = h_j^3 - ch_j^2 + c^ph_j - 1 = 0$ it follows that $h_j \neq 0$ and that $F(c, h_j)^p = h_j^{3p} - c^ph_j^{2p} + c^{p^2}h_j^p - 1 = 0$. With $c^{p^2} = c$ and $h_j \neq 0$ it follows that $-h_j^{3p}(h_j^{-3p} - ch_j^{-2p} + c^ph_j^{-p} - 1) = -h_j^{3p} * F(c, h_j^{-p}) = 0$, which proves *iv*.

From *iv* it follows, without loss of generality, that either $h_j = h_j^{-p}$ for $j = 0, 1, 2$, or $h_0 = h_0^{-p}$, $h_1 = h_2^{-p}$, and $h_2 = h_1^{-p}$, or that $h_j = h_{j+1 \bmod 3}^{-p}$ for $j = 0, 1, 2$. In either case *v* follows. Furthermore, in the first case all h_j have order dividing $p + 1$ and are thus in $\mathrm{GF}(p^2)$. In the second case, h_0 has order dividing $p+1$, $h_1 = h_2^{-p} = h_1^{p^2}$ and $h_2 = h_1^{-p} = h_2^{p^2}$ so that h_1 and h_2 both have order dividing p^2-1. It follows that they are all again in $\mathrm{GF}(p^2)$. In the last case it follows from $1 = h_0 * h_1 * h_2$ that $1 = h_0 * h_2^{-p} * h_0^{-p} = h_0 * h_0^{p^2} * h_0^{-p} = h_0^{p^2-p+1}$ so that h_0 and similarly h_1 and h_2 have order dividing $p^2 - p + 1$. If either one, say h_0, has order at most 3, then h_0 has order 1 or 3 since $p^2 - p + 1$ is odd. It follows that the order of h_0 divides $p^2 - 1$ so that $h_0 \in \mathrm{GF}(p^2)$. But then h_1

and h_2 are in $\mathrm{GF}(p^2)$ as well, because $h_j = h_{j+1 \bmod 3}^{-p}$. It follows that in the last case either all h_j have order dividing $p^2 - p + 1$ and > 3, or all h_j are in $\mathrm{GF}(p^2)$, which concludes the proof of *vi*.

If all $h_j \in \mathrm{GF}(p^2)$, then *vii* is immediate. Otherwise $F(c, X)$ is irreducible and its roots are the conjugates of h_0. Thus $c_n = Tr(h_0^n) \in \mathrm{GF}(p^2)$ (cf. 2.2). This concludes the proof of *vii* and Lemma 2.3.2.

Remark 2.3.3 It follows from Lemma 2.3.2.*vi* that $F(c, X) \in \mathrm{GF}(p^2)[X]$ is irreducible if and only if its roots have order dividing $p^2 - p + 1$ and > 3.

Lemma 2.3.4
i. $c_{u+v} = c_u * c_v - c_v^p * c_{u-v} + c_{u-2v}$ *for* $u, v \in \mathbf{Z}$.
ii. $F(c_n, h_j^n) = 0$ *for* $j = 0, 1, 2$ *and* $n \in \mathbf{Z}$.
iii. $F(c, X)$ *is reducible over* $\mathrm{GF}(p^2)$ *if and only if* $c_{p+1} \in \mathrm{GF}(p)$.

Proof. With the definition of c_n, $c_n^p = c_{-n}$ (cf. Lemma 2.3.2.*v*), and Lemma 2.3.2.*ii*, the proof of *i* follows from a straightforward computation.

For the proof of *ii* we compute the coefficients of $(X - h_0^n)(X - h_1^n)(X - h_2^n)$. We find that the coefficient of X^2 equals $-c_n$ and that the constant coefficient equals $-h_0^n * h_1^n * h_2^n = -(h_0 * h_1 * h_2)^n = -1$ (cf. Lemma 2.3.2.*ii*). The coefficient of X equals $h_0^n * h_1^n + h_0^n * h_2^n + h_1^n * h_2^n = c_{-n} = c_n^p$ (cf. Lemma 2.3.2.*iii* and *v*). It follows that $(X - h_0^n)(X - h_1^n)(X - h_2^n) = F(c_n, X)$ from which *ii* follows.

If $F(c, X)$ is reducible then all h_j are in $\mathrm{GF}(p^2)$ (cf. Remark 2.3.3 and Lemma 2.3.2.*vi*). It follows that $h_j^{(p+1)p} = h_j^{p+1}$ so that $h_j^{p+1} \in \mathrm{GF}(p)$ for $j = 0, 1, 2$ and $c_{p+1} \in \mathrm{GF}(p)$. Conversely, if $c_{p+1} \in \mathrm{GF}(p)$, then $c_{p+1}^p = c_{p+1}$ and $F(c_{p+1}, X) = X^3 - c_{p+1}X^2 + c_{p+1}X - 1$. Thus, $F(c_{p+1}, 1) = 0$. Because the roots of $F(c_{p+1}, X)$ are the $(p+1)^{\mathrm{st}}$ powers of the roots of $F(c, X)$ (cf. *iv*), it follows that $F(c, X)$ has a root of order dividing $p + 1$, i.e., an element of $\mathrm{GF}(p^2)$, so that $F(c, X)$ is reducible over $\mathrm{GF}(p^2)$. This proves *iii*.

Lemma 2.3.2.*v* and Lemma 2.3.4.*i* lead to a fast algorithm to compute c_n for any $n \in \mathbf{Z}$.

Corollary 2.3.5 *Let* c, c_{n-1}, c_n, *and* c_{n+1} *be given.*
i. Computing $c_{2n} = c_n^2 - 2c_n^p$ *takes two multiplications in* $\mathrm{GF}(p)$.
ii. Computing $c_{n+2} = c * c_{n+1} - c^p * c_n + c_{n-1}$ *takes four multiplications in* $\mathrm{GF}(p)$.
iii. Computing $c_{2n-1} = c_{n-1} * c_n - c^p * c_n^p + c_{n+1}^p$ *takes four multiplications in* $\mathrm{GF}(p)$.
iv. Computing $c_{2n+1} = c_{n+1} * c_n - c * c_n^p + c_{n-1}^p$ *takes four multiplications in* $\mathrm{GF}(p)$.

Proof. The identities follow from Lemma 2.3.2.*v* and Lemma 2.3.4.*i*: with $u = v = n$ and $c_0 = 3$ for *i*, with $u = n + 1$ and $v = 1$ for *ii*, $u = n - 1$, $v = n$ for *iii*, and $u = n + 1$, $v = n$ for *iv*. The cost analysis follows from Lemma 2.1.1.

Definition 2.3.6 Let $S_n(c) = (c_{n-1}, c_n, c_{n+1}) \in \mathrm{GF}(p^2)^3$.

Algorithm 2.3.7 (Computation of $S_n(c)$ given c) If $n < 0$, apply this algorithm to $-n$ and use Lemma 2.3.2.*v*. If $n = 0$, then $S_0(c) = (c^p, 3, c)$ (cf. Lemma 2.3.2.*v*). If $n = 1$, then $S_1(c) = (3, c, c^2 - 2c^p)$ (cf. Corollary 2.3.5.*i*). If $n = 2$, use Corollary 2.3.5.*ii* and $S_1(c)$ to compute c_3 and thereby $S_2(n)$. Otherwise, to compute $S_n(c)$ for $n > 2$ let $m = n$. If m is even, then replace m by $m-1$. Let $m = 2\bar{m}+1$, $\bar{S}_{\bar{m}}(c) = S_m(c)$, $\bar{k} = 1$, and compute $\bar{S}_{\bar{k}}(c) = S_3(c)$ using Corollary 2.3.5.*ii* and $S(2)$. Let $\bar{m} = \sum_{j=0}^{r} \bar{m}_j 2^j$ with $\bar{m}_j \in \{0, 1\}$ and $\bar{m}_r = 1$. For $j = r-1, r-2, \ldots, 0$ in succession do the following:

- If $\bar{m}_j = 0$ then use $\bar{S}_{\bar{k}}(c) = (c_{2\bar{k}}, c_{2\bar{k}+1}, c_{2\bar{k}+2})$ to compute $\bar{S}_{2\bar{k}}(c) = (c_{4\bar{k}}, c_{4\bar{k}+1}, c_{4\bar{k}+2})$ (using Corollary 2.3.5.*i* for $c_{4\bar{k}}$ and $c_{4\bar{k}+2}$ and Corollary 2.3.5.*iii* for $c_{4\bar{k}+1}$) and replace $\bar{k}$ by $2\bar{k}$.
- If $\bar{m}_j = 1$ then use $\bar{S}_{\bar{k}}(c) = (c_{2\bar{k}}, c_{2\bar{k}+1}, c_{2\bar{k}+2})$ to compute $\bar{S}_{2\bar{k}+1}(c) = (c_{4\bar{k}+2}, c_{4\bar{k}+3}, c_{4\bar{k}+4})$ (using Corollary 2.3.5.*i* for $c_{4\bar{k}+2}$ and $c_{4\bar{k}+4}$ and Corollary 2.3.5.*iv* for $c_{4\bar{k}+3}$) and replace $\bar{k}$ by $2\bar{k}+1$,

After this iteration we have that $\bar{k} = \bar{m}$ and $S_m(c) = \bar{S}_{\bar{m}}(c)$. If n is even use $S_m(c) = (c_{m-1}, c_m, c_{m+1})$ to compute $S_{m+1}(c) = (c_m, c_{m+1}, c_{m+2})$ (using Corollary 2.3.5.*ii*) and replace m by $m+1$. As a result we have $S_n(c) = S_m(c)$.

Theorem 2.3.8 *Given the sum c of the roots of $F(c, X)$, computing the sum c_n of the n^{th} powers of the roots takes $8\log_2(n)$ multiplications in* GF(p).

Proof. Immediate from Algorithm 2.3.7 and Corollary 2.3.5.

Remark 2.3.9 The only difference between the two different cases in Algorithm 2.3.7 (i.e., if the bit is off or on) is the application of Corollary 2.3.5.*iii* if the bit is off and of Corollary 2.3.5.*iv* if the bit is on. The two computations involved, however, are very similar and take the same number of instructions. Thus, the instructions carried out in Algorithm 2.3.7 for the two different cases are very much alike. This is a rather unusual property for an exponentiation routine and makes Algorithm 2.3.7 much less susceptible than usual exponentiation routines to environmental attacks such as timing attacks and Differential Power Analysis.

2.4 Computing with traces

It follows from Lemma 2.2.1 and Lemma 2.3.4.*ii* that

$$S_n(Tr(g)) = (Tr(g^{n-1}), Tr(g^n), Tr(g^{n+1}))$$

(cf. Definition 2.3.6). Furthermore, given $Tr(g)$ Algorithm 2.3.7 can be used to compute $S_n(Tr(g))$ for any n. Since the order of g equals q this takes $8\log_2(n \bmod q)$ multiplications in GF(p) (cf. Theorem 2.3.8). According to Lemma 2.1.2.*iii* computing g^n given g can be expected to take $23.4\log_2(q)$ multiplications in GF(p). Thus, computing $Tr(g^n)$ given $Tr(g)$ is almost three times faster than computing g^n given g. Furthermore, $Tr(g^n) \in \mathrm{GF}(p^2)$ whereas $g^n \in \mathrm{GF}(p^6)$. So representing, storing, or transmitting $Tr(g^n)$ is three times cheaper than it is for g^n. Unlike the methods from for instance [2], we do not assume that p

has a special form. Using such primes leads to additional savings by making the arithmetic in GF(p) faster (cf. Algorithm 3.1.1).

Thus, we replace the traditional representation of powers of g by their traces. The ability to quickly compute $Tr(g^n)$ based on $Tr(g)$ suffices for the implementation of many cryptographic protocols (cf. Section 4). In some protocols, however, the product of two powers of g must be computed. For the standard representation this is straightforward, but if traces are used, then computing products is relatively complicated. We describe how this problem may be solved in the cryptographic applications that we are aware of. Let $Tr(g) \in \mathrm{GF}(p^2)$ and $S_k(Tr(g)) \in \mathrm{GF}(p^2)^3$ (cf. Definition 2.3.6) be given for some secret integer k (the private key) with $0 < k < q$. We show that $Tr(g^a * g^{bk})$ can be computed efficiently for any $a, b \in \mathbf{Z}$.

Definition 2.4.1 Let $A(c) = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -c^p \\ 0 & 1 & c \end{pmatrix}$ and $M_n(c) = \begin{pmatrix} c_{n-2} & c_{n-1} & c_n \\ c_{n-1} & c_n & c_{n+1} \\ c_n & c_{n+1} & c_{n+2} \end{pmatrix}$ be 3×3-matrices over GF(p^2) with c and c_n as in Definition 2.3.1, and let $C(V)$ denote the center column of a 3×3 matrix V.

Lemma 2.4.2 *$S_n(c) = S_m(c) * A(c)^{n-m}$ and $M_n(c) = M_m(c) * A(c)^{n-m}$ for $n, m \in \mathbf{Z}$.*

Proof. For $n - m = 1$ the first statement is equivalent with Corollary 2.3.5.*ii*. The proof follows by induction to $n - m$.

Corollary 2.4.3 $c_n = S_m(c) * C(A(c)^{n-m})$.

Lemma 2.4.4 *The determinant of $M_0(c)$ equals $D = c^{2p+2} + 18c^{p+1} - 4(c^{3p} + c^3) - 27 \in \mathrm{GF}(p)$. If $D \neq 0$ then*

$$M_0(c)^{-1} = \frac{1}{D} * \begin{pmatrix} 2c^2 - 6c^p & 2c^{2p} + 3c - c^{p+2} & c^{p+1} - 9 \\ 2c^{2p} + 3c - c^{p+2} & (c^2 - 2c^p)^{p+1} - 9 & (2c^{2p} + 3c - c^{p+2})^p \\ c^{p+1} - 9 & (2c^{2p} + 3c - c^{p+2})^p & (2c^2 - 6c^p)^p \end{pmatrix}.$$

Proof. This follows from a simple computation using Lemma 2.3.2.*v* and Corollary 2.3.5 combined with the fact that $x \in \mathrm{GF}(p)$ if $x^p = x$.

Lemma 2.4.5 $\det(M_0(Tr(g))) = (Tr(g^{p+1})^p - Tr(g^{p+1}))^2 \neq 0$.

Proof. This follows by observing that $M_0(Tr(g))$ is the product of the Vandermonde matrix $\begin{pmatrix} g^{-1} & g^{-p^2} & g^{-p^4} \\ 1 & 1 & 1 \\ g & g^{p^2} & g^{p^4} \end{pmatrix}$ and its inverse, and therefore invertible. The determinant of the Vandermonde matrix equals $Tr(g^{p+1})^p - Tr(g^{p+1})$.

Lemma 2.4.6 *$A(Tr(g))^n = M_0(Tr(g))^{-1} * M_n(Tr(g))$ can be computed in a small constant number of operations in GF(p^2) given $Tr(g)$ and $S_n(Tr(g))$.*

Proof. $Tr(g^{n\pm 2})$ and thus $M_n(Tr(g))$ can be computed from $S_n(Tr(g))$ using Corollary 2.3.5.*ii*. The proof follows from Lemmas 2.4.2, 2.4.4, 2.4.5, and 2.1.1.*i*.

Corollary 2.4.7 $C(A(Tr(g))^n) = M_0(Tr(g))^{-1} * (S_n(Tr(g)))^T$.

Algorithm 2.4.8 (Computation of $Tr(g^a * g^{bk})$) Let $Tr(g)$, $S_k(Tr(g))$ (for unknown k), and $a, b \in \mathbf{Z}$ with $0 < a, b < q$ be given.

1. Compute $e = a/b \bmod q$.
2. Compute $S_e(Tr(g))$ (cf. Algorithm 2.3.7).
3. Compute $C(A(Tr(g))^e)$ based on $Tr(g)$ and $S_e(Tr(g))$ using Corollary 2.4.7.
4. Compute $Tr(g^{e+k}) = S_k(Tr(g)) * C(A(Tr(g))^e)$ (cf. Corollary 2.4.3).
5. Compute $S_b(Tr(g^{e+k}))$ (cf. Algorithm 2.3.7), and return $Tr(g^{(e+k)b}) = Tr(g^a * g^{bk})$.

Theorem 2.4.9 *Given* $M_0(Tr(g))^{-1}$, $Tr(g)$, *and* $S_k(Tr(g)) = (Tr(g^{k-1}), Tr(g^k), Tr(g^{k+1}))$ *the trace* $Tr(g^a * g^{bk})$ *of* $g^a * g^{bk}$ *can be computed at a cost of* $8\log_2(a/b \bmod q) + 8\log_2(b) + 34$ *multiplications in* $\mathrm{GF}(p)$.

Proof. The proof follows from a straightforward analysis of the cost of the required matrix vector operations and Theorem 2.3.8.

Assuming that $M_0(Tr(g))^{-1}$ is computed once and for all (at the cost of a small constant number of operations in $\mathrm{GF}(p^2)$), we find that $Tr(g^a * g^{bk})$ can be computed at a cost of $16\log_2(q)$ multiplications in $\mathrm{GF}(p)$. According to Lemma 2.1.2.*iv* this computation would cost about $27.9\log_2(q)$ multiplications in $\mathrm{GF}(p)$ using the traditional representation. Thus, in this case the trace representation achieves a speed-up of a factor 1.75 over the traditional one. We conclude that both single and double exponentiations can be done substantially faster using traces than using previously published techniques.

3 Parameter selection

3.1 Finite field and subgroup size selection

We describe fast and practical methods to select the field characteristic p and subgroup size q such that q divides $p^2 - p + 1$. Denote by P and Q the sizes of the primes p and q to be generated, respectively. To achieve security at least equivalent to 1024-bit RSA, $6P$ should be set to about 1024, i.e., $P \approx 170$, and Q can for instance be set at 160. Given current cryptanalytic methods we do not recommend choosing P much smaller than Q.

Algorithm 3.1.1 (Selection of q and 'nice' p) Find $r \in \mathbf{Z}$ such that $q = r^2 - r + 1$ is a Q-bit prime, and next find $k \in \mathbf{Z}$ such that $p = r + k * q$ is a P-bit prime that is 2 mod 3.

Algorithm 3.1.1 is quite fast and it can be used to find primes p that satisfy a degree two polynomial with small coefficients. Such p lead to fast arithmetic operations in $\mathrm{GF}(p)$. In particular if the search for k is restricted to $k = 1$ (i.e., search for an r such that both $r^2 - r + 1$ and $r^2 + 1$ are prime and such that $r^2 + 1 \equiv 2 \bmod 3$) the primes p have a very nice form; note that in this case r must be even and $p \equiv 1 \bmod 4$. On the other hand, such 'nice' p may be undesirable from a security point of view because they may make application of the Discrete Logarithm variant of the Number Field Sieve easier. Another method to generate p and q that does not have this disadvantage (and thus neither the advantage of fast arithmetic modulo p) is the following.

Algorithm 3.1.2 (Selection of q and p) First, select a Q-bit prime $q \equiv 7 \bmod 12$. Next, find the roots r_1 and r_2 of $X^2 - X + 1 \bmod q$. It follows from $q \equiv 1 \bmod 3$ and quadratic reciprocity that r_1 and r_2 exist. Since $q \equiv 3 \bmod 4$ they can be found using a single $((q+1)/4)^{\text{th}}$ powering modulo q. Finally, find a $k \in \mathbf{Z}$ such that $p = r_i + k * q$ is a P-bit prime that is $2 \bmod 3$ for $i = 1$ or 2.

The run time of Algorithms 3.1.1 and 3.1.2 is dominated by the time to find the primes q and p. A precise analysis is straightforward and left to the reader.

3.2 Subgroup selection

We consider the problem of finding a proper $Tr(g)$ for an element $g \in \mathrm{GF}(p^6)$ of order q dividing $p^2 - p + 1$ and > 3. Note that there is no need to find g itself, finding $Tr(g)$ suffices. Given $Tr(g)$ for an unspecified g, a subgroup generator can be computed by finding a root in $\mathrm{GF}(p^6)$ of $F(Tr(g), X)$. We refer to this generator as g and to the subgroup $\langle g \rangle$ as the *XTR group*. Note that all roots of $F(Tr(g), X)$ lead to the same XTR group.

A straightforward approach to find $Tr(g)$ would be to find a third degree irreducible polynomial over $\mathrm{GF}(p^2)$, use it to represent $\mathrm{GF}(p^6)$, to pick an element $h \in \mathrm{GF}(p^6)$ until $h^{(p^6-1)/q} \neq 1$, to take $g = h^{(p^6-1)/q}$, and to compute $Tr(g)$. Although conceptually easy, this method is less attractive from an implementation point of view. A faster method that is also easier to implement is based on the following lemma.

Lemma 3.2.1 *For a randomly selected $c \in \mathrm{GF}(p^2)$ the probability that $F(c, X) \in \mathrm{GF}(p^2)[X]$ is irreducible is about one third.*

Proof. This follows from a straightforward counting argument. About $p^2 - p$ elements of the subgroup of order $p^2 - p + 1$ of $\mathrm{GF}(p^6)^*$ are roots of monic irreducible polynomials of the form $F(c, X)$ (cf. Lemma 2.2.1 and Lemma 2.3.4.*ii*). Since each of these polynomials has three distinct roots, there must be about $(p^2-p)/3$ different values for c in $\mathrm{GF}(p^2) \backslash \mathrm{GF}(p)$ such that $F(c, X)$ is irreducible.

With Remark 2.3.3 it follows that it suffices to pick a $c \in \mathrm{GF}(p^2)$ until $F(c, X)$ is irreducible and until $c_{(p^2-p+1)/q} \neq 3$ (cf. Definition 2.3.1), and to take $Tr(g) = c_{(p^2-p+1)/q}$. The resulting $Tr(g)$ is the trace of some g of order q, but explicit computation of g is avoided. As shown in [13] the irreducibility test for $F(c, X) \in \mathrm{GF}(p^2)[X]$ can be done very fast, but, obviously, it requires additional code. We now present a method that requires hardly any additional code on top of Algorithm 2.3.7.

Algorithm 3.2.2 (Computation of $Tr(g)$)

1. Pick $c \in \mathrm{GF}(p^2) \backslash \mathrm{GF}(p)$ at random and compute c_{p+1} using Algorithm 2.3.7.
2. If $c_{p+1} \in \mathrm{GF}(p)$ then return to Step 1.
3. Compute $c_{(p^2-p+1)/q}$ using Algorithm 2.3.7.
4. If $c_{(p^2-p+1)/q} = 3$, then return to Step 1.
5. Let $Tr(g) = c_{(p^2-p+1)/q}$.

Theorem 3.2.3 *Algorithm 3.2.2 computes an element of* $\mathrm{GF}(p^2)$ *that equals* $Tr(g)$ *for some* $g \in \mathrm{GF}(p^6)$ *of order* q. *It can be expected to require* $3q/(q-1)$ *applications of Algorithm 2.3.7 with* $n = p + 1$ *and* $q/(q-1)$ *applications with* $n = (p^2 - p + 1)/q$.

Proof. The correctness of Algorithm 3.2.2 follows from the fact that $F(c, X)$ is irreducible if $c_{p+1} \notin \mathrm{GF}(p)$ (cf. Lemma 2.3.4.*iii*). The run time estimate follows from Lemma 3.2.1 and the fact that $c_{p+1} \notin \mathrm{GF}(p)$ if $F(c, X)$ is irreducible (cf. Lemma 2.3.4.*iii*).

In [13] we present an even faster method to compute $Tr(g)$ if $p \not\equiv 8 \bmod 9$.

3.3 Key size

The XTR public key data contain two primes p and q as in 3.1 and the trace $Tr(g)$ of a generator of the XTR group (cf. 3.2). In principle the XTR public key data p, q, and $Tr(g)$ can be shared among any number of participants, just as in DSA (and EC-DSA) finite field (and curve), subgroup order, and subgroup generator may be shared. Apart from the part that may be shared, someone's XTR public key may also contain a public point $Tr(g^k)$ for an integer k that is kept secret (the private key). Furthermore, for some applications the values $Tr(g^{k-1})$ and $Tr(g^{k+1})$ are required as well (cf. Section 4). In this section we discuss how much overhead is required for the representation of the XTR public key in a certificate, i.e., on top of the user ID and other certification related bits.

The part $(p, q, Tr(g))$ that may be shared causes overhead only if it is not shared. In that case, $(p, q, Tr(g))$ may be assumed to belong to a particular user or group of users in which case it is straightforward to determine $(p, q, Tr(g))$, during initialization, as a function of the user (or user group) ID and a small number of additional bits. For any reasonable choice of P and Q (cf. 3.1) the number of additional bits on top of the user ID, i.e., the overhead, can easily be limited to 48 (6 bytes) (cf. [13]), at the cost of a one time application of Algorithm 2.3.7 with $n = (p^2 - p + 1)/q$ by the recipient of the public key data.

We are not aware of a method to reduce the overhead caused by a user's public point $Tr(g^k) \in \mathrm{GF}(p^2)$. Thus, representing $Tr(g^k)$ in a certificate requires representation of $2P$ bits. The two additional values $Tr(g^{k-1}), Tr(g^{k+1}) \in \mathrm{GF}(p^2)$, however, can be represented using far fewer than $4P$ bits, at the cost of a very reasonable one time computation by the recipient of the public key.

This can be seen as follows. Since $\det(A(c)^k) = 1$, the equation from Lemma 2.4.6 leads to a third degree equation in $Tr(g^{k-1})$, given $Tr(g)$, $Tr(g^k)$, and $Tr(g^{k+1})$, by taking the determinants of the matrices involved. Thus, at the cost of a small number of p^{th} powerings in $\mathrm{GF}(p^2)$, $Tr(g^{k-1})$ can be determined based on $Tr(g)$, $Tr(g^k)$, and $Tr(g^{k+1})$ and two bits to indicate which of the roots equals $Tr(g^{k-1})$. In [13] we present, among others, a conceptually more complicated method to determine $Tr(g^{k-1})$ based on $Tr(g)$, $Tr(g^k)$, and $Tr(g^{k+1})$ that requires only a small constant number of operations in $\mathrm{GF}(p)$, and a method to quickly determine $Tr(g^{k+1})$ given $Tr(g)$ and $Tr(g^k)$ that works if $p \not\equiv 8 \bmod 9$. Because this condition is not unduly restrictive we may assume that the two additional values $Tr(g^{k-1}), Tr(g^{k+1}) \in \mathrm{GF}(p^2)$ do not have to be included in the XTR public key data, assuming the public key recipient is able and willing to carry out a fast one time computation given the XTR public key data $(p, q, Tr(g), Tr(g^k))$. If this computation if infeasible for the recipient, then $Tr(g^{k+1})$ must be included in the XTR public key data; computation of $Tr(g^{k-1})$ then takes only a small constant number of operations in $\mathrm{GF}(p)$.

4 Cryptographic applications

XTR can be used in any cryptosystem that relies on the (subgroup) discrete logarithm problem. In this section we describe some applications of XTR in more detail: Diffie-Hellman key agreement in 4.1, ElGamal encryption in 4.2, and Nyberg-Rueppel message recovery digital signatures in 4.3, and we compare XTR to RSA and ECC (cf. [15]).

4.1 XTR-DH

Suppose that Alice and Bob who both have access to the XTR public key data p, q, $Tr(g)$ want to agree on a shared secret key K. This can be done using the following XTR version of the Diffie-Hellman protocol:

1. Alice selects at random $a \in \mathbf{Z}$, $1 < a < q-2$, uses Algorithm 2.3.7 to compute $S_a(Tr(g)) = (Tr(g^{a-1}), Tr(g^a), Tr(g^{a+1})) \in \mathrm{GF}(p^2)^3$, and sends $Tr(g^a) \in \mathrm{GF}(p^2)$ to Bob.
2. Bob receives $Tr(g^a)$ from Alice, selects at random $b \in \mathbf{Z}$, $1 < b < q-2$, uses Algorithm 2.3.7 to compute $S_b(Tr(g)) = (Tr(g^{b-1}), Tr(g^b), Tr(g^{b+1})) \in \mathrm{GF}(p^2)^3$, and sends $Tr(g^b) \in \mathrm{GF}(p^2)$ to Bob.
3. Alice receives $Tr(g^b)$ from Bob, uses Algorithm 2.3.7 to compute $S_a(Tr(g)^b) = (Tr(g^{(a-1)b}), Tr(g^{ab}), Tr(g^{(a+1)b})) \in \mathrm{GF}(p^2)^3$, and determines K based on $Tr(g^{ab}) \in \mathrm{GF}(p^2)$.
4. Bob uses Algorithm 2.3.7 to compute $S_b(Tr(g)^a) = (Tr(g^{a(b-1)}), Tr(g^{ab}), Tr(g^{a(b+1)})) \in \mathrm{GF}(p^2)^3$, and determines K based on $Tr(g^{ab}) \in \mathrm{GF}(p^2)$.

The communication and computational overhead of XTR-DH are both about one third of traditional implementations of the Diffie-Hellman protocol that are based on subgroups of multiplicative groups of finite fields, and that achieve the same level of security (cf. Subsection 2.4).

4.2 XTR-ElGamal encryption

Suppose that Alice is the owner of the XTR public key data p, q, $Tr(g)$, and that Alice has selected a secret integer k, computed $S_k(Tr(g))$, and made public the resulting value $Tr(g^k)$. Given Alice's XTR public key data $(p, q, Tr(g), Tr(g^k))$, Bob can encrypt a message M intended for Alice using the following XTR version of the ElGamal encryption protocol:

1. Bob selects at random $b \in \mathbf{Z}$, $1 < b < q-2$, and uses Algorithm 2.3.7 to compute $S_b(Tr(g)) = (Tr(g^{b-1}), Tr(g^b), Tr(g^{b+1})) \in \mathrm{GF}(p^2)^3$.
2. Bob uses Algorithm 2.3.7 to compute $S_b(Tr(g^k)) = (Tr(g^{(b-1)k}), Tr(g^{bk}), Tr(g^{(b+1)k})) \in \mathrm{GF}(p^2)^3$.
3. Bob determines a symmetric encryption key K based on $Tr(g^{bk}) \in \mathrm{GF}(p^2)$.
4. Bob uses an agreed upon symmetric encryption method with key K to encrypt M, resulting in the encryption E.
5. Bob sends $(Tr(g^b), E)$ to Alice.

Upon receipt of $(Tr(g^b), E)$, Alice decrypts the message in the following way:

1. Alice uses Algorithm 2.3.7 to compute $S_k(Tr(g^b)) = (Tr(g^{b(k-1)}), Tr(g^{bk}), Tr(g^{b(k+1)})) \in \mathrm{GF}(p^2)^3$.
2. Alice determines the symmetric encryption key K based on $Tr(g^{bk}) \in \mathrm{GF}(p^2)$.
3. Alice uses the agreed upon symmetric encryption method with key K to decrypt E, resulting in the encryption M.

The message $(Tr(g^b), E)$ sent by Bob consists of the actual encryption E, whose length strongly depends on the length of M, and the overhead $Tr(g^b) \in \mathrm{GF}(p^2)$, whose length is independent of the length of M. The communication and computational overhead of XTR-ElGamal encryption are both about one third of traditional implementations of the ElGamal encryption protocol that are based on subgroups of multiplicative groups of finite fields, and that achieve the same level of security (cf. Subsection 2.4).

Remark 4.2.1 XTR-ElGamal encryption as described above is based on the common hybrid version of ElGamal's method, i.e., where the key K is used in conjunction with an agreed upon symmetric key encryption method. In more traditional ElGamal encryption the message is restricted to the key space and 'encrypted' using, for instance, multiplication by the key, an invertible operation that takes place in the key space. In our description this would amount to requiring that $M \in \mathrm{GF}(p^2)$, and by computing E as $K * M \in \mathrm{GF}(p^2)$. Compared to non-hybrid ElGamal encryption, XTR saves a factor three on the length of both parts of the encrypted message, for messages that fit in the key space (of one third of the 'traditional' size).

Remark 4.2.2 As in other descriptions of ElGamal encryption it is implicitly assumed that the first component of an ElGamal encrypted message represents

$Tr(g^b)$, i.e. the conjugate of a power of g, which should be explicitly verified in some situations. This can explicitly be tested by checking that $Tr(g^b) \in \mathrm{GF}(p^2) \setminus \mathrm{GF}(p)$, that $Tr(g^b) \neq 3$, and by using Algorithm 2.3.7 to compute $S_q(Tr(g^b)) = (Tr(g^{b(q-1)}), Tr(g^{bq}), Tr(g^{b(q+1)}))$ and to verify that $Tr(g^{bq}) = 3$. This follows using methods similar to the ones presented in Section 3.

4.3 XTR-Nyberg-Rueppel signatures

Let, as in 4.2, Alice's XTR public key data consist of p, q, $Tr(g)$, and $Tr(g^k)$. Furthermore, assume that $Tr(g^{k-1})$ and $Tr(g^{k+1})$ (and thus $S_k(Tr(g))$) are available to the verifier, either because they are part of the public key, or because they were reconstructed by the verifier (either from $(p, q, Tr(g), Tr(g^k), Tr(g^{k+1}))$ or from $(p, q, Tr(g), Tr(g^k))$). We describe the XTR version of the Nyberg-Rueppel (NR) message recovery signature scheme, but XTR can also be used in other 'ElGamal-like' signature schemes. To sign a message M containing an agreed upon type of redundancy, Alice does the following:

1. Alice selects at random $a \in \mathbf{Z}$, $1 < a < q-2$, and uses Algorithm 2.3.7 to compute $S_a(Tr(g)) = (Tr(g^{a-1}), Tr(g^a), Tr(g^{a+1})) \in \mathrm{GF}(p^2)^3$.
2. Alice determines a symmetric encryption key K based on $Tr(g^a) \in \mathrm{GF}(p^2)$.
3. Alice uses an agreed upon symmetric encryption method with key K to encrypt M, resulting in the encryption E.
4. Alice computes the (integer valued) hash h of E.
5. Alice computes $s = (k * h + a) \bmod q \in \{0, 1, \ldots, q-1\}$.
6. Alice's resulting signature on M is (E, s).

To verify Alice's signature (E, s) and to recover the signed message M, the verifier Bob does the following.

1. Bob checks that $0 \leq s < q$; if not failure.
2. Bob computes the hash h of E.
3. Bob replaces h by $-h \bmod q \in \{0, 1, \ldots, q-1\}$.
4. Bob uses Algorithm 2.4.8 to compute $Tr(g^s * g^{hk})$ based on $Tr(g)$ and $S_k(Tr(g))$.
5. Bob uses $Tr(g^s * g^{hk})$ (which equals $Tr(g^a)$) to decrypt E resulting in M.
6. The signature is accepted $\Longleftrightarrow$ M contains the agreed upon redundancy.

XTR-NR is considerably faster than traditional implementations of the NR scheme that are based on subgroups of multiplicative groups of finite fields of the same security level. The length of the signature is identical to other variants of the hybrid version of the NR scheme (cf. Remark 4.2.1): an overhead part of length depending on the desired security (i.e., the subgroup size) and a message part of length depending on the message itself and the agreed upon redundancy. Similar statements hold for other digital signature schemes, such as DSA.

4.4 Comparison to RSA and ECC

We compare XTR to RSA and ECC. For the RSA comparison we give the run times of 1020-bit RSA and 170-bit XTR obtained using generic software. For ECC we assume random curves over prime fields of about 170-bits with a curve subgroup of 170-bit order, and we compare the number of multiplications in GF(p) required for 170-bit ECC and 170-bit XTR applications. This 'theoretical' comparison is used because we do not have access to ECC software.

If part of the public key is shared (ECC or XTR only), XTR and ECC public keys consist of just the public point. For ECC its y-coordinate can be derived from the x-coordinate and a single bit. In the non-shared case, public keys may be ID-based or non-ID-based[3]. For ECC, the finite field, random curve, and group order take $\approx$ 595 bits, plus a small number of bits for a point of high order. Using methods similar to the one alluded to in Subsection 3.3 this can be reduced to an overhead of, say, 48 bits (to generate curve and field based on the ID and 48 bits) plus 85 bits for the group order information. For XTR the sizes given in Table 1 follow from Subsection 3.3. For both RSA and XTR 100 random keys were generated. (ECC parameter generation is much slower and more complicated than for either RSA or XTR and not included in Table 1.) For RSA we used random 32-bit odd public exponents and 1020-bit moduli picked by randomly selecting 510-bit odd numbers and adding 2 until they are prime. For XTR we used Algorithm 3.1.2 with $Q = 170$ and $P \geq 170$ and the fast $Tr(g)$ initialization method mentioned at the end of Subsection 3.2. For each RSA key 10 encryptions and decryptions of random 1020-bit messages were carried out, the latter with Chinese remaindering (CRT) and without (in parentheses in Table 1). For each XTR key 10 single and double exponentiations (i.e., applications of Algorithms 2.3.7 and 2.4.8, respectively) were carried out for random exponents $< q$. For RSA encryption and decryption correspond to signature verification

Table 1. RSA, XTR, ECC key sizes and RSA, XTR run times.

	shared keysize	ID-based keysize	non-ID-based keysize	key selection	encrypting (verifying)	decrypting (signing)
1020-bit RSA	n/a	510 bits	1050 bits	1224 ms	5 ms	40 (no CRT: 123) ms
170-bit XTR	340	388 bits	680 bits	73 ms	23 ms	11 ms
170-bit ECC	171	304 bits	766 bits			

Table 2. 170-bit ECC, XTR comparison of number of multiplications in GF(p).

	encrypting	decrypting	encryption overhead	signing	verifying	signature overhead	DH speed	DH size
ECC	3400	1921 (1700)	171 (340) bits	1700	2575	170 bits	3842 (3400)	171 (340) bits
XTR	2720	1360	340 bits	1360	2754	170 bits	2720	340 bits

[3] ID based key generation for RSA affects the way the secret factors are determined. The ID based approach for RSA is therefore viewed with suspicion and not generally used. A method from [23], for instance, has been broken, but no attack against the methods from [12] is known. For discrete logarithm based methods (such as ECC and XTR) ID-based key generation affects only the part of the public key that is not related to the secret information, and is therefore not uncommon for such systems.

and generation, respectively. For XTR single exponentiation corresponds to decryption and signature generation, and double exponentiation corresponds to signature verification and, approximately, encryption. The average run times are in milliseconds on a 450 MHz Pentium II NT workstation. The ECC figures in Table 2 are based on the results from [4]; speed-ups that may be obtained at the cost of specifying the full y-coordinates are given between parentheses. The time or number of operations to reconstruct the full public keys from their compressed versions (for either system) is not included.

5 Security

5.1 Discrete logarithms in GF(p^t)

Let $\langle\gamma\rangle$ be a multiplicative group of order ω. The security of the Diffie-Hellman protocol in $\langle\gamma\rangle$ relies on the *Diffie-Hellman* (DH) problem of computing γ^{xy} given γ^x and γ^y. We write $DH(\gamma^x, \gamma^y) = \gamma^{xy}$. Two other problems are related to the DH problem. The first one is the *Diffie-Hellman Decision* (DHD) problem: given $a, b, c \in \langle\gamma\rangle$ determine whether $c = DH(a, b)$. The DH problem is at least as difficult as the DHD problem. The second one is the *Discrete Logarithm* (DL) problem: given $a = \gamma^x \in \langle\gamma\rangle$ with $0 \leq x < \omega$, find $x = DL(a)$. The DL problem is at least as difficult as the DH problem. It is widely assumed that if the DL problem in $\langle\gamma\rangle$ is intractable, then so are the other two. Given the factorization of ω, the DL problem in $\langle\gamma\rangle$ can be reduced to the DL problem in all prime order subgroups of $\langle\gamma\rangle$, due to the Pohlig-Hellman algorithm [17]. Thus, for the DL problem we may assume that ω is prime.

Let p, q, $Tr(g)$ be (part of) an XTR public key. Below we prove that the security of the XTR versions of the DL, DHD, and DH problem is equivalent to the DL, DHD, and DH problem, respectively, in the XTR group (cf. Subsection 3.2). First, however, we focus on the DL problem in a subgroup $\langle\gamma\rangle$ of prime order ω of the multiplicative group GF$(p^t)^*$ of an extension field GF(p^t) of GF(p) for a fixed t. There are two approaches to this problem (cf. [1], [5], [9], [11], [16], [19], [21]): one can either attack the multiplicative group or one can attack the subgroup. For the first attack the best known method is the Discrete Logarithm variant of the Number Field Sieve. If s is the smallest divisor of t such that $\langle\gamma\rangle$ can be embedded in the subgroup GF$(p^s)^*$ of GF$(p^t)^*$, then the heuristic expected asymptotic run time for this attack is $L[p^s, 1/3, 1.923]$, where $L[n, v, u] = \exp((u + o(1))(\ln(n))^v(\ln(\ln(n)))^{1-v})$. If p is small, e.g. $p = 2$, then the constant 1.923 can be replaced by 1.53. Alternatively, one can use one of several methods that take $O(\sqrt{\omega})$ operations in $\langle\gamma\rangle$, such as Pollard's Birthday Paradox based rho method (cf. [18]).

This implies that the difficulty of the DL problem in $\langle\gamma\rangle$ depends on the size of the minimal surrounding subfield of $\langle\gamma\rangle$ and on the size of its prime order ω. If GF(p^t) itself is the minimal surrounding subfield of $\langle\gamma\rangle$ and ω is sufficiently large, then the DL problem in $\langle\gamma\rangle$ is as hard as the general DL problem in GF(p^t). If p is not small the latter problem is believed to be as hard as the DL problem with respect to a generator of prime order $\approx \omega$ in the multiplicative group of a prime field of cardinality $\approx p^t$ (cf. [6], [20]). The DL problem in that setting is generally considered to be harder than factoring $t * \log_2(p)$-bit RSA moduli.

The XTR parameters are chosen in such away that the minimal surrounding field of the XTR group is equal to $\mathrm{GF}(p^6)$ (cf. Section 1), such that p is not small, and such that q is sufficiently large. It follows that, if the complexity of the DL problem in the XTR group is less than the complexity of the DL problem in $\mathrm{GF}(p^6)$, then the latter problem is at most as hard as the DL problem in $\mathrm{GF}(p^3)$, $\mathrm{GF}(p^2)$, or $\mathrm{GF}(p)$, i.e., the DL problem in $\mathrm{GF}(p^6)$ collapses to its true subfields. This contradicts the above mentioned assumption about the complexity of computing discrete logarithms in $\mathrm{GF}(p^t)$. It follows that the DL problem in the XTR group may be assumed to be as hard as the DL problem in $\mathrm{GF}(p^6)$, i.e., of complexity $L[p^6, 1/3, 1.923]$. Thus, with respect to known attacks, the DL problem in the XTR group is generally considered to be more difficult than factoring a $6 * \log_2(p)$-bit RSA modulus, provided the prime order q is sufficiently large. By comparing the computational effort required for both algorithms mentioned above, it turns out that if p and q each are about 170 bits long, then the DL problem in the XTR group is harder than factoring an RSA modulus of $6 * 170 = 1020$ bits.

5.2 Security of XTR

Discrete logarithm based cryptographic protocols can use many different types of subgroups, such as multiplicative groups of finite fields, subgroups thereof (such as the XTR group), or groups of points of elliptic curves over finite fields. As shown in Section 4 the XTR versions of these protocols follow by replacing elements of the XTR group by their traces. This implies that the security of those XTR versions is no longer based on the original DH, DHD, or DL problems but on the XTR versions of those problems. We define the *XTR-DH* problem as the problem of computing $Tr(g^{xy})$ given $Tr(g^x)$ and $Tr(g^y)$, and we write $XDH(g^x, g^y) = g^{xy}$. The *XTR-DHD* problem is the problem of determining whether $XDH(a, b) = c$ for $a, b, c \in Tr(\langle g \rangle)$. Given $a \in Tr(\langle g \rangle)$, the *XTR-DL* problem is to find $x = XDL(a)$, i.e., $0 \leq x < q$ such that $a = Tr(g^x)$. Note that if $x = DL(a)$, then so are $x * p^2 \bmod q$ and $x * p^4 \bmod q$.

We say that problem $\mathcal{A}$ is (a, b)*-equivalent* to problem $\mathcal{B}$, if any instance of problem $\mathcal{A}$ (or $\mathcal{B}$) can be solved by at most a (or b) calls to an algorithm solving problem $\mathcal{B}$ (or $\mathcal{A}$).

Theorem 5.2.1 *The following equivalences hold:*

1. *The XTR-DL problem is* $(1, 1)$*-equivalent to the DL problem in* $\langle g \rangle$.
2. *The XTR-DH problem is* $(1, 2)$ *equivalent to the DH problem in* $\langle g \rangle$.
3. *The XTR-DHD problem is* $(3, 2)$*-equivalent to the DHD problem in* $\langle g \rangle$.

Proof. For $a \in \mathrm{GF}(p^2)$ let $r(a)$ denote a root of $F(a, X)$.

To compute $DL(y)$, let $x = XDL(Tr(y))$, then $DL(y) = x * p^{2j} \bmod q$ for either $j = 0$, $j = 1$, or $j = 2$. Conversely, $XDL(a) = DL(r(a))$. This proves *i.*

To compute $DH(x,y)$, compute $d_i = XDH(Tr(x*g^i),Tr(y))$ for $i=0,1$, then $r(d_i) \in \{(DH(x,y)*y^i)^{p^{2j}} : j=0,1,2\}$, from which $DH(x,y)$ follows. Conversely, $XDH(a,b) = Tr(DH(r(a),r(b)))$. This proves *ii*.

To prove *iii*, it easily follows that $DH(x,y)=z$ if and only if $XDH(Tr(x), Tr(y)) = Tr(z)$ and $XDH(Tr(x*g),Tr(y)) = Tr(z*y)$. Conversely, $XDH(a,b) = c$ if and only if $DH(r(a),r(b)) = r(c)^{p^{2j}}$ for either $j=0$, $j=1$, or $j=2$. This proves *iii* and completes the proof of Theorem 5.2.1.

Remark 5.2.2 It follows from the arguments in the proof of Theorem 5.2.1 that an algorithm solving either DL, DH, or DHD with non-negligible probability can be transformed in an algorithm solving the corresponding XTR problem with non-negligible probability, and vice versa.

It follows from the arguments in the proof of Theorem 5.2.1.*ii* that in many practical situations a single call to an XTR-DH solving algorithm would suffice to solve a DL problem. As an example we mention DH key agreement where the resulting key is actually used after it has been established.

Remark 5.2.3 Theorem 5.2.1.*ii* states that determining the (small) XTR-DH key is as hard as determining the whole DH key in the representation group $\langle g \rangle$. From the results in [24] it actually follows that determining the image of the XTR-DH key under any non-trivial GF(p)-linear function is also as hard as the whole DH key. This means that, for example, finding the α or the α^2 coefficient of the XTR-DH key is as hard as finding the whole DH key, implying that cryptographic applications may be based on just one of the coefficients.

Acknowledgment. We are greatly indebted to Mike Wiener for his permission to include his improvements of our earlier versions of Algorithms 2.3.7 and 2.4.8.

References

1. L.M. Adleman, J. DeMarrais, *A subexponential algorithm for discrete logarithms over all finite fields*, Proceedings Crypto'93, LNCS 773, Springer-Verlag 1994, 147-158.
2. D.V. Bailey, C. Paar, *Optimal extension fields for fast arithmetic in public-key algorithms*, Proceedings Crypto'98, LNCS 1462, Springer-Verlag 1998, 472-485.
3. A.E. Brouwer, R. Pellikaan, E.R. Verheul, *Doing more with fewer bits*, Proceedings Asiacrypt99, LNCS 1716, Springer-Verlag 1999, 321-332.
4. H. Cohen, A. Miyaji, T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates*, Proceedings Asiacrypt'98, LNCS 1514, Springer-Verlag 1998, 51-65.
5. D. Coppersmith, *Fast evaluation of logarithms in fields of characteristic two*, IEEE Trans. Inform. Theory 30 (1984), 587-594.
6. D. Coppersmith, personal communication, March 2000.
7. T. ElGamal, *A Public Key Cryptosystem and a Signature scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory 31(4), 1985, 469-472.
8. P. Gaudry, F. Hess, N.P. Smart, *Constructive and destructive facets of Weil descent on elliptic curves*, manuscript, January, 2000, submitted to Journal of Cryptology.

9. D. Gordon, *Discrete logarithms in GF(p) using the number field sieve*, SIAM J. Discrete Math. 6 (1993), 312-323.
10. D.E. Knuth, *The art of computer programming, Volume 2, Seminumerical Algorithms*, second edition, Addison-Wesley, 1981.
11. A.K. Lenstra, *Using Cyclotomic Polynomials to Construct Efficient Discrete Logarithm Cryptosystems over Finite Fields*, Proceedings ACISP97, LNCS 1270, Springer-Verlag 1997, 127-138.
12. A.K. Lenstra, *Generating RSA moduli with a predetermined portion*, Proceedings Asiacrypt '98, LNCS 1514, Springer-Verlag 1998, 1-10.
13. A.K. Lenstra, E.R. Verheul, *Key improvements to XTR*, in preparation.
14. A.J. Menezes, *Comparing the security of ECC and RSA*, manuscript, January, 2000, available as www.cacr.math.uwaterloo.ca/ ajmeneze/misc/cryptogram-article.html.
15. A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.
16. A. Odlyzko, *Discrete Logarithms: The past and the future*, Designs, Codes and Cryptography, 19 (2000), 129-145.
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. J.M. Pollard, *Monte Carlo methods for index computation (mod p)*, Math. Comp., 32 (1978), 918-924.
19. O. Schirokauer, *Discrete logarithms and local units*, Phil. Trans. R. Soc. Lond. A 345, 1993, 409-423.
20. O. Schirokauer, personal communication, March 2000.
21. O. Schirokauer, D. Weber, Th.F. Denny, *Discrete logarithms: the effectiveness of the index calculus method*, Proceedings ANTS II, LNCS 1122 Springer-Verlag 1996.
22. C.P. Schnorr, *Efficient signature generation by smart cards*, Journal of Cryptology, 4 (1991), 161-174.
23. S.A. Vanstone, R.J. Zuccherato, *Short RSA keys and their generation*, Journal of Cryptology, 8 (1995), 101-114.
24. E. Verheul, *Certificates of Recoverability with Scalable Recovery Agent Security*, Proceedings of PKC 2000, LNCS 1751, Springer-Verlag 2000, 258-275.

This article was processed using the LaTeX macro package with LLNCS style