Binary Elligator Squared

Diego F. Aranha[1], Pierre-Alain Fouque[2], Chen Qian[3],
Mehdi Tibouchi[4], and Jean-Christophe Zapalowicz[5]

[1] Institute of Computing, University of Campinas, `dfaranha@ic.unicamp.br`
[2] Université de Rennes 1 and Institut Universitaire de France, `fouque@irisa.fr`
[3] ENS Rennes, `chen.qian@ens-rennes.fr`
[4] NTT Secure Platform Laboratories, `tibouchi.mehdi@lab.ntt.co.jp`
[5] Inria, `jean-christophe.zapalowicz@inria.fr`

Abstract. Applications of elliptic curve cryptography to anonymity, privacy and censorship circumvention call for methods to represent uniformly random points on elliptic curves as uniformly random bit strings, so that, for example, ECC network traffic can masquerade as random traffic.

At ACM CCS 2013, Bernstein et al. proposed an efficient approach, called "Elligator," to solving this problem for arbitrary elliptic curve-based cryptographic protocols, based on the use of efficiently invertible maps to elliptic curves. Unfortunately, such invertible maps are only known to exist for certain classes of curves, excluding in particular curves of prime order and curves over binary fields. A variant of this approach, "Elligator Squared," was later proposed by Tibouchi (FC 2014) supporting not necessarily injective encodings to elliptic curves (and hence a much larger class of curves), but, although some rough efficiency estimates were provided, it was not clear how an actual implementation of that approach would perform in practice.

In this paper, we show that Elligator Squared can indeed be implemented very efficiently with a suitable choice of curve encodings. More precisely, we consider the binary curve setting (which was not discussed in Tibouchi's paper), and implement the Elligator Squared bit string representation algorithm based on a suitably optimized version of the Shallue–van de Woestijne characteristic 2 encoding, which we show can be computed using only multiplications, trace and half-trace computations, and a few inversions.

On the fast binary curve of Oliveira et al. (CHES 2013), our implementation runs in an average of only 22850 Haswell cycles, making uniform bit string representations possible for a very reasonable overhead—much smaller even than Elligator on Edwards curves.

As a side contribution, we also compare implementations of Elligator and Elligator Squared on a curve supported by Elligator, namely Curve25519. We find that generating a random point and its uniform bitstring representation is around 35–40% faster with Elligator for protocols using a fixed base point (such as static ECDH), but 30–35% faster with Elligator Squared in the case of a variable base point (such as ElGamal encryption). Both are significantly slower than our binary curve implementation.

Keywords: Elligator, Binary Elliptic Curves, Efficient Implementation, PCLMULQDQ, Anonymity & Privacy.

1 Introduction

Elliptic curves offer many advantages for public-key cryptography compared to more traditional settings like RSA and finite field discrete logarithms, including higher efficiency, a much smaller key size that scales gracefully with security requirements, and a rich geometric structure that enables the construction of additional primitives like bilinear pairings. On the Internet, adoption of elliptic curve cryptography is growing in general-purpose protocols like TLS, SSH and S/MIME, as well as anonymity and privacy-enhancing tools like Tor (which favors ECDH key exchange in recent versions) and Bitcoin (which is based on ECDSA).

For circumvention applications, however, ECC presents a weakness: points on a given elliptic curve, when represented in a usual way (even in compressed form) are easy to distinguish from random bit strings. For example, the usual compressed bit string representation of an elliptic curve point is essentially the x-coordinate of the point, and only about half of all possible x-coordinates correspond to valid points (the other half being x-coordinates of points of the quadratic twist). This makes it relatively easy for an attacker to distinguish ECC traffic (the transcripts of multiple ECDH key exchanges, say) from random traffic, and then proceed to intercept, block or otherwise tamper with such traffic.

To alleviate that problem, one possible approach is to modify protocols so that transmitted points randomly lie either on the given elliptic curve or on its quadratic twist (and the curve parameters must therefore be chosen to be twist-secure). This is the approach taken by Möller [23], who constructed a CCA-secure KEM with uniformly random ciphertexts using an elliptic curve and its twist. This approach has also been used in the context of kleptography, as considered by Young and Yung [30,31], and has already been deployed in circumvention tools, including StegoTorus [28], a camouflage proxy for Tor, and Telex [29], an anticensorship technology that uses a covert channel in TLS handshakes to securely communicate with friendly proxy servers. However, since protocols and security proofs have to be adapted to work on both a curve and its twist, this approach is not particularly versatile, and it imposes additional security requirements (twist-security) on the choice of curve parameters.

A different approach, called "Elligator," was presented at ACM CCS 2013 by Bernstein, Hamburg, Krasnova and Lange [6]. Their idea is to leverage an efficiently computable, efficiently invertible algebraic function that maps the integer interval $S = \{0, \dots, (p-1)/2\}$, p prime, *injectively* to the group $E(\mathbb{F}_p)$ where E is an elliptic curve over $\mathbb{F}_p$. Bernstein et al. observe that, since ι is injective, a uniformly random point P in $\iota(S) \subset E(\mathbb{F}_p)$ has a uniformly random preimage $\iota^{-1}(P)$ in S, and use that observation to represent an elliptic curve point P as the bit string representation of the unique integer $\iota^{-1}(P)$ if it exists. If the prime p is close to a power of 2, a uniform point in $\iota(S)$ will have a close to uniform bit string representation.

This method has numerous advantages over Möller's twisted curve method: it is easier to adapt to existing protocols using elliptic curves, since there is no need to modify them to also deal with the quadratic twist; it avoids the need to publish a twisted curve counterpart of each public key element, hence allowing a more compact public key; and it doesn't impose additional security requirements like twist-security. But it crucially relies on the existence of an injective encoding ι, only a few examples of which are known [13,17,6], all of them for elliptic curves of non-prime order over large characteristic fields. This makes the method inapplicable to implementations based on curves of prime order or on binary fields, which rules out most standardized ECC parameters [15,11,22,1], in particular. Moreover, the rejection sampling involved (when a point P is picked outside $\iota(S)$, the protocol has to start over) can impose a significant performance penalty.

To overcome these limitations, Tibouchi [27] recently proposed a variant of Elligator, called "Elligator Squared," in which a point $P \in E(\mathbb{F}_q)$ is represented not by a preimage under an injective encoding ι, but by a randomly sampled preimage under an essentially surjective map $\mathbb{F}_q^2 \to E(\mathbb{F}_q)$ with good statistical properties, known as an *admissible encoding* following a terminology introduced by Brier et al. [10]. By results due to Farashahi et al. [14], such admissible encodings are known to exist for all isomorphism classes of elliptic curves, including curves of prime order and binary curves. Since admissible encodings are essentially surjective, the approach also eliminates the need for rejection sampling at the protocol level.

Our contributions. While the Elligator Squared approach is quite versatile, its efficiency is highly dependent on how fast the underlying admissible encoding can be computed and sampled, and the same can be said of Elligator in the settings were it can be used. Since, to the best of our knowledge, no detailed implementation results or concrete performance numbers have been published so far for the underlying encodings, one only has some rough estimates to go by. For Elligator, Bernstein et al. give ballpark Westmere cycle count figures based on earlier implementation results [7], and for Elligator Squared, Tibouchi provides some average operation counts in [27] for a few selected encoding functions. No performance-oriented implementation is available for either approach.

In this paper, we provide the first such implementation for Elligator Squared, and do so in the binary curve setting, which had not been considered by Tibouchi. Binary curves provide a major advantage for algorithms like Elligator Squared due to the existence of a point encoding function, the binary Shallue–van de Woestijne encoding [25], that can be computed without base field exponentiations. Using the framework of Farashahi et al. [14], one can obtain an admissible encoding from that function, and hence use it to implement Elligator Squared.

We propose various algorithmic improvements and computation tricks to obtain a fast evaluation of the binary Shallue–van de Woestijne encoding and of the associated Elligator Squared sampling algorithm. In particular, our description is much more efficient than the one given in [9, Appendix E].

Based on these algorithmic improvements, we performed software implementations of Elligator Squared on the record-setting binary GLS curve of Oliveira et al. , defined over $\mathbb{F}_{2^{254}}$ [24]. We dedicate special attention to optimizing the performance-critical operations and introduce corresponding novel techniques, namely a new point addition formula in λ-affine coordinates and a faster approach for constant-time half-trace computation over quadratic extensions of $\mathbb{F}_{2^m}$. Moreover, timings are presented for both variable-time and constant-time field arithmetic.[6] The resulting timings compare very favorably to previously suggested estimates.

Finally, as a side contribution, we also propose concrete cycle counts on Ivy Bridge and Haswell for both Elligator and Elligator Squared on the Edwards curve Curve25519 [4] based on the publicly available implementation of Ed25519 [5]. We find that, on this curve, the Elligator approach is roughly 35–40% faster than Elligator Squared for protocols that rely on fixed-base scalar multiplication, such as ECDH, but conversely, for protocols that rely on variable-base scalar multiplication like ElGamal encryption, Elligator Squared is 30–35% faster. Both approaches are significantly slower than what we achieve on the same CPU with our binary curve implementation.

2 Preliminaries

Let E be an elliptic curve over a finite field $\mathbb{F}_q$.

2.1 Well-bounded encodings

Definition 1. *A function $f : \mathbb{F}_q \to E(\mathbb{F}_q)$ is said to be B-well-distributed encoding for a certain constant $B > 0$ if for any nontrivial character χ of $E(\mathbb{F}_q)$, the following holds:*

$$\left| \sum_{u \in \mathbb{F}_q} \chi(f(u)) \right| \leq B\sqrt{q}.$$

Definition 2. *We call a function $f : \mathbb{F}_q \to E(\mathbb{F}_q)$ a (d, B)-well-bounded encoding, for positive constants d, B, when f is B-well-distributed and all points in $E(\mathbb{F}_q)$ have at most d preimages under f.*

2.2 Elligator Squared

Let $f : \mathbb{F}_q \to E(\mathbb{F}_q)$ be a (d, B)-well-bounded encoding and let $f^{\otimes 2}$ the tensor square defined by:

$$\begin{aligned} f^{\otimes 2} : \mathbb{F}_q^2 &\to E(\mathbb{F}_q) \\ (u, v) &\mapsto f(u) + f(v). \end{aligned}$$

Tibouchi shows in [27] that if we sample a uniformly random preimage under $f^{\otimes 2}$ of a uniformly random point P on the curve, we get a pair $(u, v) \in \mathbb{F}_q^2$ which is statistically close to uniform. Moreover he proves that sampling uniformly random preimages under $f^{\otimes 2}$ can be done efficiently for all points $P \in E(\mathbb{F}_q)$ except possibly a negligible fraction of them [27, Theorem 1]. The sampling algorithm Tibouchi proposed is described as Algorithm 1. The idea is to randomly pick a random u and then to compute a correct candidate v such that $P = f(u) + f(v)$. The last steps of the algorithm (step 5 to 7) are also needed in order to ensure the uniform distribution of the output (u, v).

[6] We point out that using constant-time arithmetic for Elligator Squared is not required in most realistic adversarial models, but it does offer protection against very powerful distinguishing attackers, so the paranoid may prefer that option nonetheless.

Algorithm 1 Preimage sampling algorithm for $f^{\otimes 2}$.

```
1: function SamplePreimage($P$)
2:     repeat
3:         $u \xleftarrow{\$} \mathbb{F}_q$
4:         $Q \leftarrow P - f(u)$
5:         $i \leftarrow \#f^{-1}(Q)$
6:         $j \xleftarrow{\$} \{1, \cdots, d\}$
7:     until $j \leq i$
8:     $\{v_1, \cdots, v_t\} \leftarrow f^{-1}(Q)$
9:     return $(u, v_j)$
10: end function
```

2.3 Shallue–van de Woestijne in Characteristic 2

In this section, we recall the Shallue–van de Woestijne algorithm in characteristic 2 [25], following the more explicit presentation given in [9, Appendix E]. An elliptic curve over a field $\mathbb{F}_{2^n}$ is a set of points $(x, y) \in (\mathbb{F}_{2^n})^2$ verifying the equation:

$$E_{a,b} : Y^2 + X \cdot Y = X^3 + a \cdot X^2 + b$$

where $a, b \in (\mathbb{F}_{2^n})^2$. Let g be the rational function $x \mapsto x^{-2} \cdot (x^3 + a \cdot x^2 + b)$. Letting $Z = Y/X$, the equation for $E_{a,b}$ can be rewritten as $Z^2 + Z = g(X)$.

Theorem 1. *Let $g(x) = x^{-2} \cdot (x^3 + a \cdot x^2 + b)$ where $a, b \in (\mathbb{F}_{2^n})^2$. Let*

$$X_1(t, w) = \frac{t \cdot c}{1 + t + t^2} \qquad X_2(t, w) = t \cdot X_1(t, w) + c \qquad X_3(t, w) = \frac{X_1(t, w) \cdot X_2(t, w)}{X_1(t, w) + X_2(t, w)}$$

where $c = a + w + w^2$. Then $g(X_1(t, w)) + g(X_2(t, w)) + g(X_3(t, w)) \in h(\mathbb{F}_{2^n})$ where h is the map $h : z \mapsto z^2 + z$.

From Theorem 1, we have that at least one of the $g(X_i(t, w))$ must be in $h(\mathbb{F}_{2^n})$, which leads to a point in $E_{a,b}(\mathbb{F}_{2^n})$. Indeed, we have that $h(\mathbb{F}_{2^n}) = \{z \in \mathbb{F}_{2^n} \mid \mathrm{Tr}(z) = 0\}$, where Tr is the trace operator $\mathrm{Tr} : \mathbb{F}_{2^n} \to \mathbb{F}_2$ with:

$$\mathrm{Tr} = \sum_{i=0}^{n-1} z^{2^i}$$

(one inclusion is obvious and the other one follows from the fact that the kernel of the $\mathbb{F}_2$-linear map h is $\{0, 1\}$, hence its image is a hyperplane). As a result, $\sum_{i=1}^{3} \mathrm{Tr}(g(X_i)) = 0$ and therefore at least one of the X_i must satisfy $\mathrm{Tr}(g(X_i)) = 0$ since Tr is $\mathbb{F}_2$-valued. Such an X_i is indeed the abscissa of a point in $E_{a,b}(\mathbb{F}_{2^n})$, and we can find its y-coordinate by solving the quadratic equation $Z^2 + Z = g(X_i)$. That equation is $\mathbb{F}_2$-linear, so finding Z amounts to solving a linear system over $\mathbb{F}_2$. This yields the point-encoding function described in Algorithm 2.

In the description of that algorithm, the solution of the quadratic equation is expressed in terms of the map $\mathrm{QS} : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ ("quadratic solver"), which is the well-defined linear map such that, for all x, $\mathrm{QS}(x)$ is the trace zero solution of the quadratic equation $z^2 + z = x + \mathrm{Tr}(x)$. When n is odd, QS is straightforward to compute: it is the half-trace map HTr defined as:

$$\mathrm{HTr} : z \mapsto \sum_{i=0}^{(n-1)/2} z^{2^{2i}}.$$

Algorithm 2 Shallue–van de Woestijne algorithm in characteristic 2.

```
Require: $a, b \in \mathbb{F}_{2^n}$ and $t, w \in \mathbb{F}_{2^n}$
Ensure: $(x, y) \in E_{a,b}$
1: $c \leftarrow a + w + w^2$
2: $X_1 \leftarrow t \cdot c/(1 + t + t^2)$
3: $X_2 \leftarrow t \cdot X_1 + c$
4: $X_3 \leftarrow X_1 \cdot X_2/(X_1 + X_2)$
5: for $j = 1$ to $3$ do
6:     $h_j \leftarrow (X_j^3 + a \cdot X_j^2 + b)/X_j^2$
7:     if $\mathrm{Tr}(h_j) = 0$ then return $(X_j, \mathrm{QS}(h_j) \cdot X_j)$
8:     end if
9: end for
```

We discuss the efficient computation of QS in even degree extensions in §4.

Algorithm 2 actually maps two parameters t, w to a rational point on the curve $E_{a,b}$. One can obtain a map $f \colon \mathbb{F}_q \to E_{a,b}(\mathbb{F}_q)$ by picking one of the two parameters as a suitable constant and letting the other one vary. In what follows, for efficiency reasons, we fix t and use w as the variable parameter.

One can check that the resulting function is well-bounded in the sense of §2.1. Indeed, the framework of Farashahi et al. [14] can be used to establish that it is a well-distributed encoding: the proof is easily adapted from the one given in [18] for the positive characteristic version of the Shallue–van de Woestijne algorithm. Moreover, each curve point has at most 6 preimages under the corresponding function: there are at most two values of w that yield a given value of X_1, and similarly for X_2, X_3. Thus, we obtain a (d, B)-well-bounded encoding for an explicitly computable constant B and $d = 6$.

2.4 Lambda affine coordinates

In order to have more efficient binary elliptic curve arithmetic, we will use lambda coordinates [24]. Given a point $P = (x, y) \in E_{a,b}(\mathbb{F}_{2^n})$, with $x \neq 0$, its λ-affine representation of P is defined as (x, λ) where $\lambda = x + y/x$. The λ-affine equation of the Weierstrass Equation of the curve $y^2 + xy = x^3 + ax^2 + b$ is $(\lambda^2 + \lambda + a)x^2 = x^4 + b$. Note that the condition $x \neq 0$ is not restrictive in practice since the only point $x = 0$ satisfying Weierstrass equation is $(0, \sqrt{b})$.

3 Algorithmic aspects

We focus on Algorithm 1 proposed by Tibouchi in [27], which we adapt for the specific characteristic 2 finite field. More precisely, we consider an elliptic curve over a field $\mathbb{F}_{2^n}$ that satisfies the equation in λ-coordinates:

$$E_{a,b} : (\lambda^2 + \lambda + a)X^2 = X^4 + b$$

where $a, b \in (\mathbb{F}_{2^n})^2$. The $(6, B)$-well-bounded encoding we consider for our efficient Elligator Squared implementation is the binary Shallue–van de Woestijne algorithm recalled in §2.3.

One of its properties is that among three candidates denoted X_1, X_2, X_3, either exactly one of them or all three are x-coordinate of a rational point over the binary elliptic curve $E_{a,b}$, and the algorithm outputs the first correct one. Owing to this property, some additionnal verifications during preimage computation, since it is not always true that $\mathrm{SWC{\scriptstyle HAR}2}_X(\mathrm{SWC{\scriptstyle HAR}2}_X^{-1}(X_i)) = X_i$ for $i = 2, 3$, where we denote by $\mathrm{SWC{\scriptstyle HAR}2}_X$ the x-coordinate of the binary Shallue–van de Woestijne algorithm, and by $\mathrm{SWC{\scriptstyle HAR}2}_X^{-1}$ an arbitrary preimage thereof (see the discussion on the subroutine PREIMAGESSW in §3.3 for more details). We also have to consider another property of this algorithm, concerning the output. Indeed the y-coordinate has a specific form and thus, before searching for some preimages of the point Q, one has to test whether this property is verified (see the discussion on the overall complexity in §3.3 for more details).

The details of our preimage sampling algorithm in characteristic 2 are described in Algorithm 3 with t fixed to a constant such that $t(t+1)(t^2+t+1) \neq 0$, *i.e.* $t \notin \mathbb{F}_4$. Note that we make the choice to use the λ-coordinates for efficiency reasons justified in §3.2. The rest of the section consists in describing the two subroutines SWCHAR2 and PREIMAGESSW, as well as in evaluating the overall complexity of Algorithm 3.

3.1 The subroutine SWCHAR2

The first subroutine represents the binary Shallue–van de Woestijne algorithm and its pseudocode for our case is given as Algorithm 4. Given a value $u \in \mathbb{F}_{2^n}$, it outputs the lambda coordinates of a point over the binary elliptic curve $E_{a,b}$.

Since the field inversion is by far the most expensive field operation (see [24] for experimental timings and Table 2 below), we have modified Algorithm 2 so that we have a single inversion of c to perform. Indeed Algorithm 2 requires at most 4 field inversions: the first one at step 4 and the three others at step 6. However the parameters X_i and $1/X_i$ for $j = 1, 2, 3$ can be expressed using c, $1/c$ and some constants depending on t which can be precomputed (see Table 1). Note that X_3 can be computed as $c \cdot t_3$, or more efficiently as $X_1 + X_2 + c$ but this requires to keep in memory X_1 and X_2. Finally this algorithm requires a single field inversion, a QS computation and some negligible field operations (multiplications, squarings and trace computations).

$X_1 \leftarrow t_1 \cdot c$	$1/X_1 \leftarrow 1/t_1 \cdot 1/c$
$X_2 \leftarrow t_2 \cdot c$	$1/X_2 \leftarrow 1/t_2 \cdot 1/c$
$X_3 \leftarrow X_1 + X_2 + c$	$1/X_3 \leftarrow 1/t_3 \cdot 1/c$

Table 1. Efficient computation of values X_i and $1/X_i$ for $i = 1, \cdots 3$. The values $t_1 = \frac{t}{1+t+t^2}, 1/t_1, t_2 = \frac{1+t}{1+t+t^2}, 1/t_2$ and $1/t_3 = \frac{1+t+t^2}{t(1+t)}$ can be precomputed, with t a constant such that $t \notin \mathbb{F}_4$.

3.2 The subroutine PREIMAGESSW

The second subroutine is useful to compute the number of preimages of the point $Q = (x_Q, \lambda_Q)$ by Algorithm 4. Its pseudocode is detailed as Algorithm 5 and refers to the steps 5 and 8 of Algorithm 1.

This subroutine is more complex due to the properties of the binary Shallue–van de Woestijne algorithm. More precisely, there is an order relation in Algorithm 4: if X_1 corresponds to a x-coordinate of a point over

Algorithm 3 Preimage Sampling Algorithm in Characteristic 2

```
 1: Precomputed: $t_1 = \frac{t}{1+t+t^2}, t_2 = \frac{1+t}{1+t+t^2}, t_3 = \frac{t(1+t)}{1+t+t^2}$
 2: function SAMPLEPREIMAGE($E_{a,b}, P = (x_P, \lambda_P)$)
 3:     repeat
 4:         repeat
 5:             $u \overset{\$}{\leftarrow} \mathbb{F}_{2^n}$
 6:             $R \leftarrow$ SWCHAR2($E_{a,b}, u, t_1, t_2, t_3$)
 7:             $Q \leftarrow P - R$
 8:         until $\mathrm{Tr}(\lambda_Q - x_Q) = 0$
 9:         $k, S = \{v_1, \cdots, v_k\} \leftarrow$ PREIMAGESSW($E_{a,b}, Q, t_1, t_2, t_3$)
10:         $j \overset{\$}{\leftarrow} \{1, \cdots, 6\}$
11:     until $j \leq k$
12:     return $(u, v_j)$
13: end function
```

Algorithm 4 Efficient Binary Shallue–van de Woestijne Algorithm

```
1: function SWChar2($E_{a,b}, u, t_1, t_2, t_3$)
2:     $c \leftarrow u^2 + u + a$
3:     $c_{-1} \leftarrow 1/c$
4:     for $j = 1$ to 3 do                          ▷ Compute $h_j$ and perform a trace test
5:         $X_j \leftarrow t_j \cdot c$                          ▷ or $X_3 \leftarrow X_1 + X_2 + c$
6:         $X_{-j} \leftarrow 1/t_j \cdot c_{-1}$                 ▷ $1/t_j$ can also be precomputed
7:         $h_j \leftarrow (X_{-j})^2 \cdot b + X_j + a$
8:         if $\mathrm{Tr}(h_j) = 0$ then           ▷ At least one of the three potential tests will succeed
9:             $x \leftarrow X_j$
10:            $\lambda \leftarrow \mathrm{QS}(h_j) + x$
11:            break                          ▷ Only take into account the first correct solution
12:        end if
13:    end for
14:    return $(x, \lambda)$                   ▷ Lambda coordinates of a point over $E_{a,b}$
15: end function
```

the elliptic curve, then it will output this point, even if X_2 and X_3 also correspond to a possible x-coordinate. Thus, the equality $\mathrm{SWChar2}(\mathrm{SWChar2}^{-1}(X_j)) = X_j$ is true for $j = 1$ but not necessary for $j = 2, 3$. In others words, for $j = 2, 3$ a solution of $\mathrm{SWChar2}^{-1}(X_j)$ is not necessarily a preimage of X_j by SWChar2.

Starting from the equations $x_Q = X_j(t, w) = c(w) \cdot t_j$ for $j = 1, 2, 3$, with $c(w) = w^2 + w + a$, the main idea of Algorithm 5 consists in testing if there exists some values of w which satisfy these equations. If one founds some candidates for w, one also has to verify if they really correspond to preimages by Algorithm 4. From an equation $x_Q = X_j(t, w)$ we can obtain an equation $w + w^2 = x_Q/t_j - a = \alpha_j(a, t)$ which has 2 solutions if $\mathrm{Tr}(\alpha_j(a, t)) = 0$ and no solution otherwise. As an example $\alpha_1(a, t)$ is equal to $x_Q \cdot (1 + t + t^2)/t - a$. The solutions are then $w_0^1 = \mathrm{QS}(\alpha_j(a, t))$ and $w_1^1 = w_0^1 + 1$. There are thus at most 6 possible solutions for all values of j. Now for the cases $x_Q = X_2(t, w)$ and $x_Q = X_3(t, w)$, it remains to perform a verification. Actually, denoting w_0^2 one of both solutions of the equation $x_Q = X_2(t, w)$ if it exists, the computation of $\mathrm{SWChar2}(w_0^2)$ can result in $X_1(t, w_0^2)$ instead of $X_2(t, w_0^2)$, and this happens with probability $1/2$ which is the probability that $\mathrm{Tr}(h_1) = 0$. The same result holds for $x_Q = X_3(t, w)$, however note that if X_3 is solution but not X_1 then X_2 cannot be a solution since $\sum_{i=1}^{3} \mathrm{Tr}(g(X_i)) = 0$ according to Theorem 1. Thus the verification can focus only on X_1.

Naive implementation of the verification. A simple way for implementing the verification would consist in computing $\mathrm{QS}(\alpha_j(a, t))$ for $j = 2, 3$ and then calling twice the subroutine SWChar2 (without the steps referring to X_2 and X_3) for testing if the test on the trace is true or not. However this would require an additional inversion per call to compute SWChar2. Moreover, with this naive implementation we have to compute the half trace before testing if the result will be a preimage.

Efficient implementation of the verification. Since the verification focus only on X_1 as explained above, we propose an efficient way to compute b/X_1^2, which is required in order to performing the test $\mathrm{Tr}(h_1) = \mathrm{Tr}(X_1 + a + b/X_1^2)$, without any field inversion. This trick is valuable when we are working in lambda coordinates. Our proposal has another advantage: we do not need to compute the solutions, *i.e.* $w_0 = \mathrm{QS}(\alpha_j(a, t))$ and $w_1 = w_0 + 1$, before to be sure that we will get two preimages. We thus save some quite expensive half trace computations.

Consider the equation:

$$x_Q = X_2 = t_2 \cdot c = t_2 \cdot X_1/t_1 \quad \text{with} \quad c = \mathrm{QS}(\alpha_2(a, t))^2 + \mathrm{QS}(\alpha_2(a, t)) + a.$$

X_1 can be expressed as $t_1/t_2 \cdot x_Q$, whose computation is negligible for t_1/t_2 a precomputed value. Now starting from the equation of the elliptic curve in affine coordinates, *i.e* $E_{a,b} : Y^2 + X \cdot Y = X^3 + a \cdot X^2 + b$,

Algorithm 5 Preimages Computation by Algorithm 4

```
1: function PreimagesSW($E_{a,b}, Q = (x_Q, \lambda_Q), t_1, t_2, t_3$)
2:     $k \leftarrow 0$
3:     $S \leftarrow \{\}$
4:     for $j = 1$ to 3 do                                          ▷ From $x_Q = X_j(t, w)$...
5:         $\alpha_j \leftarrow x_Q \cdot 1/t_j - a$
6:         if $\mathrm{Tr}(\alpha_j) = 0$ then                       ▷ ...Test if there are some solutions
7:             if $j = 1$ then                                      ▷ For $X_1$, a solution is a preimage
8:                 $w_0 \leftarrow \mathrm{QS}(\alpha_j)$
9:                 $w_1 \leftarrow w_0 + 1$
10:                $k \leftarrow 2$
11:                $S \leftarrow \{w_0, w_1\}$
12:            else                              ▷ For $X_2, X_3$, a solution is not necessarly a preimage
13:                $X_1 \leftarrow t_1/t_j \cdot x_Q$
14:                $tmp \leftarrow [(\lambda_Q - x_Q)^2 + (\lambda_Q - x_Q) - x_Q - a] \cdot (t_j/t_1)^2$      ▷ $tmp = b/X_1^2$
15:                $h_1 \leftarrow tmp + X_1 + a$
16:                if $\mathrm{Tr}(h_1) \neq 0$ then              ▷ Test if $X_1$ would also be a correct $x$-coordinate
17:                    $w_0 \leftarrow \mathrm{QS}(\alpha_j)$
18:                    $w_1 \leftarrow w_0 + 1$
19:                    $k \leftarrow k + 2$
20:                    $S \leftarrow S \cup \{w_0, w_1\}$
21:                end if
22:            end if
23:        end if
24:    end for
25:    return $k, S$                                  ▷ $k$: number of preimages, $S$: set of preimages
26: end function
```

we divide each term by X^2 and we evaluate the equation in the point Q. We then obtain:

$$\left(\frac{y_Q}{x_Q}\right)^2 + \frac{y_Q}{x_Q} = x_Q + a + \frac{b}{x_Q^2},$$

and finally:

$$\frac{b}{X_1^2} = \left(\frac{t_2}{t_1}\right)^2 \cdot \left[\left(\frac{y_Q}{x_Q}\right)^2 + \frac{y_Q}{x_Q} - x_Q - a\right].$$

Assuming that $(t_2/t_1)^2$ is a precomputed constant, the computation of b/X_1^2 is not costly if y_Q/x_Q does not require an expensive operation. That is the case when we are working in λ-coordinates since $\lambda_Q = y_Q/x_Q + x_Q$. The same result obviously holds for the equation $x_Q = X_3$ by replacing t_2 with t_3.

To conclude, Algorithm 5 requires at most 3 QS computations and some negligible field operations (multiplications, squarings and trace computations).

3.3 Operation counts

We conclude this section by evaluating the average number of operations needed to evaluate Algorithm 3.

Proposition 1. *An evaluation of Algorithm 3 on a uniformly random curve points requires, on average and up to $O(2^{-n/2})$ variations, 6 field inversions, 6 point additions, 9 quadratic solver computations and some negligible operations such as field multiplications, field squares and trace computations.*

Proof. The proof consists in evaluating the probability for exiting the two loops. First note that the output (x, λ) of Algorithm 4 has a specific property, namely $\lambda - x$ is in the image of QS. Since we want to retrieve the preimages of a point Q, we have to be sure that $\lambda_Q - x_Q$ is indeed in that image, which we test for by verifying whether $\mathrm{Tr}(\lambda_Q - x_Q) = 0$. Indeed, all elements of the form $\mathrm{QS}(z)$ have zero trace by definition, and the converse is true for reasons of dimensions. The success probability of this test is exactly $1/2$ since Q is a uniformly random curve point. We thus have on average 2 field inversions, 2 point additions and 2 quadratic solver computations for the internal loop (steps 4 to 8).

The complexity of the external loop demands to evaluate the probabilities for having 0, 2, 4 or 6 preimages of Q. Since all tests on the trace in Algorithm 5 succeed, independently, with probability $1/2$ up to $O(2^{-n/2})$ variations[7], these probabilities are then, again up to $O(2^{-n/2})$ variations, $9/32$ for 0 preimage, $15/32$ for 2 preimages, $7/32$ for 4 preimages, and $1/32$ for 6 preimages. Thus, the probability for exiting the external loop is equal to $0 \cdot 9/32 + 1/3 \cdot 15/32 + 2/3 \cdot 7/32 + 1 \cdot 1/32 = 1/3$. These probabilities also hold for evaluating the average cost of an iteration of PreimagesSW in term of quadratic computations. With probability $15/32$ one such computation will be performed and so on. As a consequence, one iteration of PreimagesSW cost on average $\frac{15\cdot1+7\cdot2+1\cdot3}{32} = 1$ quadratic solver computation.

To sum up, Algorithm 3 requires on average $3 \cdot 2$ field inversions, $3 \cdot 2$ additions of points and $3 \cdot (2+1)$ quadratic solver computations, up to $O(2^{-n/2}$ variations. □

Note that the efficiency of this algorithm can be improved further by choosing a sparse value of b and a value of t that yields sparse precomputed constants. Many of the field multiplications will then be computed faster.

4 Implementation aspects

Our software implementation targets modern Intel Desktop-based processors, making extensive use of the recently introduced AVX instruction set [16] accessible through compiler intrinsics. The curve choice is the GLS binary curve $(\lambda^2 + \lambda + a)x^2 = x^4 + b$ represented in λ-coordinates and defined over the quadratic extension $\mathbb{F}_{2^{254}}$. The extension is built by choosing the irreducible trinomial $g(u) = u^2 + u + 1$ over the base field $\mathbb{F}_{2^{127}}$ defined with the irreducible trinomial $f(z) = z^{127} + z^{63} + 1$. In this set of parameters, a field element a is represented as $a = a_0 + a_1 u$, with $a_0, a_1 \in \mathbb{F}_{2^{127}}$. For simplicity, parameter t is chosen to be a random subfield element, allowing the computational savings by sparse multiplications described in the previous section.

Squaring and multiplication. Field squaring closely mirrors the vector formulation proposed in [3], with coefficient expansion implemented by table lookups performed through byte-shuffling instructions. The table lookups operate on registers only, allowing a very efficient constant-time implementation. Field multiplication is natively supported by the carry-less multiplier (PCLMULQDQ instruction), with the number of word multiplications reduced through application of Karatsuba formulae, as described in [26]. Modular reduction is implemented with a shift-and-add approach, with careful choice of aligning vector word shifts on multiples of 8, to explore the faster memory alignment instructions available in the target platform.

Half-trace computation. For an odd extension degree m, the half-trace function $\mathrm{HTr} : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ is defined by $\mathrm{HTr}(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}$ and computes a solution $c \in \mathbb{F}_{2^m}$ to the quadratic equation $\lambda^2 + \lambda = c + \mathrm{Tr}(c)$. In a quadratic extension, the equation $\lambda^2 + \lambda = c + \mathrm{Tr}(c)$ can be solved for $c = c_0 + c_1 u \in \mathbb{F}_{2^m}^2$ by computing two half-traces in $\mathbb{F}_{2^m}$, as described in [20]. First, solve $\lambda_1^2 + \lambda_1 = c_1$ to obtain λ_1, and then solve $\lambda_0^2 + \lambda_0 = c_0 + c_1 + \lambda_1 + \mathrm{Tr}(c_0 + c_1 + \lambda_1)$ to obtain the solution $\lambda = \lambda_0 + (\lambda_1 + \mathrm{Tr}(c_0 + c_1 + \lambda_1))u$. This approach is very efficient for variable-time implementations and only requires two half-trace computations in the base field, where each half-trace computation employs a large precomputed table of $2^8 \cdot \lceil \frac{m}{8} \rceil$ field elements [24].

[7] This can be justified rigorously using the fact that the corresponding function field extensions are pairwise linearly disjoint, exactly as in the image size computations of [18, §4]. For simplicity, we do not include the tedious Galois extension computations involved.

A more naive approach evaluates the function by alternating $m-1$ consecutive squarings and $(m-1)/2$ additions, with the advantage of taking constant-time (if squaring and addition are also constant-time, as in the case here). We derive a faster way to compute the half-trace function in constant-time over quadratic extension fields. Applying the naive approach to a quadratic extension allows a significant speedup due to the linear property of half-trace, by reducing the cost to essentially one constant-time half-trace computation over the base field. By considering that $\lambda_1^2 + \lambda_1 = c_1$ has a solution $\lambda_1 \in \mathbb{F}_{2^m}$, we always have $\mathrm{Tr}(\lambda_1) = \mathrm{Tr}(c_1) = 0$. This simplifies the expression above to $\lambda_0^2 + \lambda_0 = c_0 + c_1 + \lambda_1 + \mathrm{Tr}(c_0)$. Substituting $d = c_0 + \mathrm{Tr}(c_0)$, the expression for λ_0 becomes:

$$\lambda_0 = \sum_{i=0}^{(m-1)/2} (d + c_1 + \lambda_1)^{2^{2i}} = \sum_{i=0}^{(m-1)/2} \left(d + c_1 + \sum_{j=0}^{(m-1)/2} c_1^{2^{2j}} \right)^{2^{2i}} .$$

The expansion of the inner sum allows the interleaving of the consecutive squarings. The analysis can be split in two cases, depending on the format of the extension degree m:

$$\lambda_0 = \begin{cases} c_0 + \displaystyle\sum_{i=0}^{\lfloor m/4 \rfloor - 1} (c_0^{16} + d^4 + c_1^4 + c_1^8)^{2^{4i}} & \text{if } m \equiv 1 \pmod 4 \\ \displaystyle\sum_{i=0}^{\lfloor m/4 \rfloor} (c_0 + d^4 + c_1^2 + c_1^4)^{2^{4i}} & \text{if } m \equiv 3 \pmod 4. \end{cases}$$

The value λ_1 can then be computed as $\lambda_1 = \lambda_0^2 + \lambda_0 + d + c_1$, for a total of approximately m squarings and $m/4$ additions, a cost comparable to a single constant-time half-trace in the base field.

Inversion. Field inversion is implemented by two different approaches based on the Itoh-Tsuji algorithm [21]. This algorithm computes $a^{-1} = a^{(2^{m-1}-1)2}$, as proposed in [19], with the cost of $m-1$ squarings and a number of multiplications determined by the length of an addition chain for $m-1$. For a variable-time implementation, the squarings for each 2^i-power involved can be converted into a multi-squaring [8], implemented as a trade-off between space consumption and execution time. Each multi-squaring table requires the storage of $2^4 \cdot \lceil \frac{m}{4} \rceil$ field elements. A constant-time implementation must perform consecutive squarings and cannot benefit considerably from a precomputed table of field elements without introducing variance in the memory hierarchy latency potentially exploitable by an intrusive attacker.

Point addition. The last performance-critical operation to be described is the point addition in λ-affine coordinates. A formula for adding points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ on the curve is proposed in [24], with associated cost of 2 inversions, 4 multiplications and 2 squarings :

$$x_{P+Q} = \frac{x_P \cdot x_Q(\lambda_P + \lambda_Q)}{(x_P + x_Q)^2}, \qquad \lambda_{P+Q} = \frac{x_Q \cdot (x_{P+Q} + x_P)^2}{x_{P+Q} \cdot x_P} + \lambda_P + 1.$$

Simple substitution of x_{P+Q} in the computation of λ_{P+Q} gives faster new formulas. By unifying the denominators, one field inversion can be traded for 2 multiplications in the formulas below, with associated cost of 1 inversion, 6 multiplications and 2 squarings:

$$x_{P+Q} = \frac{x_P \cdot x_Q(\lambda_P + \lambda_Q)^2}{(x_P + x_Q)^2(\lambda_P + \lambda_Q)}, \qquad \lambda_{P+Q} = \frac{\left[(x_P + x_Q)^2 + x_Q \cdot (\lambda_P + \lambda_Q)\right]^2}{(x_P + x_Q)^2(\lambda_P + \lambda_Q)} + \lambda_P + 1.$$

Operation	Ivy Bridge	Haswell
Field squaring	13	15
Sparse multiplication	80	44
Multiplication	94	48
Inversion	959	734
Constant-time inversion	1,783	1,610
Half-trace	55	50
Constant-time half-trace	1,213	1,245
Point addition	1,500	1,026
Constant-time point addition	2,367	2,137
Elligator Squared	23,680	22,850
Constant-time Elligator Squared	52,850	51,750
DH with Elligator Squared	127,430	80,180
EG with Elligator Squared	138,480	83,680

Table 2. Timings for Elligator Squared and underlying field arithmetic in two Intel platforms. Results are in clock cycles and were taken as the average of 10^4 executions with random inputs. DH/EG results refer to generating a random point for ECDH (fixed-base) or ElGamal encryption (variable-base) using the constant-time, timing-attack protected scalar multiplication from [24], and computing its Elligator Squared representation with variable-time arithmetic.

5 Experimental results

The implementation was realized with help from the latest version of the RELIC toolkit [2]. Random number generation was implemented with the recently introduced `RDRAND` instruction [12]. Software was compiled with a prerelease version of GCC 4.9 available in the Arch Linux distribution with flags for loop unrolling, aggressive optimization (`-O3` level) and specific tuning for the Sandy/Ivy Bridge microarchitectures. Table 2 presents timings in clock cycles for field arithmetic and Elligator Squared in two different platforms – an Intel Ivy Bridge Core i5 3317U 1.7GHz and a Haswell Core i7 4770K 3.5GHz. The timings were taken as the average of 10^4 executions, with TurboBoost and HyperThreading disabled to reduce randomness in the results.

The constant-time implementation results are mostly for reference: indeed, since the Elligator Squared operation is efficiently invertible, there is no strong reason to compute it in constant time: timing information does not leak secret key data like in the case of a scalar multiplication. However, timing information could conceivably help an active distinguishing attacker; the corresponding attack scenarios are far-fetched, but the paranoid may prefer to choose constant-time arithmetic as a matter of principle.

6 Comparison of Elligator 2 and Elligator Squared on Prime Finite Fields

We have implemented Elligator 2 [6] and the corresponding Elligator Squared construction on Curve25519 [4] using the fast arithmetic provided by Bernstein et al. as part of the publicly available implementation of Curve25519 and Ed25519 [5] in SUPERCOP, in order to compare the two proposed methods on Edwards curves in large characteristic (and to see how they both perform compared to our binary implementation).

To generate a random point and compute the corresponding bitstring representation, the Elligator method requires, on average, 2 scalar multiplications, 2 tests for the existence of preimages and 1 preimage computation. On the other hand, for the same computation, Elligator Squared requires, on average, 1 scalar multiplication, 2 tests for the existence of preimages, 1 preimage computation and 2 computations of the Elligator 2 map function. As a result, compared to the Elligator approach, the Elligator Squared approach requires 1 scalar multiplication less, but 2 map function computations more. Therefore, Elligator will be faster than Elligator Squared in contexts where a scalar multiplication is cheaper than 2 map function evaluations and

Operation	Ivy Bridge	Haswell
Scalar multiplication (fixed-base)	42,570	42,180
Scalar multiplication (variable-base, est.)	182,490	162,460
Map function	38,420	36,590
DH with Elligator Squared	157,500	141,200
DH with Elligator 2	114,800	100,200
EG with Elligator Squared (est.)	297,420	261,480
EG with Elligator 2 (est.)	394,640	340,760

Table 3. Timings for Elligator Squared and Elligator 2 on Curve25519. Results are in clock cycles and were taken as the average of 10^4 executions with random inputs. DH/EG are as in Table 2.

conversely. Elligator will thus tend to have an edge for protocols using fixed base point scalar multiplication, such as ECDH key generation or ECDSA signatures, whereas Elligator Squared will perform better for protocols using variable base point scalar multiplication, like ElGamal encryption.

This is confirmed by our implementation results, as reported in Table 3, which are 35–40% in favor of Elligator in the fixed-base case (DH) but 30–35% in favor of Elligator Squared in the variable-base case (EG). Note that the variable-base scalar multiplication results are estimates based on the SUPERCOP performance numbers on `haswell` and `hydra2`. A comparison with Table 2 shows that the binary curve approach is 25% to 200% times faster than the fastest Curve25519 implementation.

References

1. ANSSI. Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française. http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/publication-d-un-parametrage-de-courbe-elliptique-visant-des-applications-de.html, November 2011.
2. D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/.
3. Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient software implementation of binary field arithmetic using vector instruction sets. In *LATINCRYPT*, pages 144–161, 2010.
4. Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.
5. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.
6. Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In Virgil Gligor and Moti Yung, editors, *ACM CCS*, 2013.
7. Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Software. http://elligator.cr.yp.to/software.html, August 2013.
8. Joppe W. Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. ECC2K-130 on Cell CPUs. In *AFRICACRYPT*, pages 225–242, 2010.
9. Eric Brier, Jean-Sebastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. Cryptology ePrint Archive, Report 2009/340, 2009. http://eprint.iacr.org/. Full version of [10].
10. Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2010.
11. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, Version 2.0, January 2010.
12. Intel Corporation. Intel Digital Random Number Generator (DRNG). https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf.
13. Reza Rezaeian Farashahi. Hashing into Hessian curves. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT*, volume 6737 of *Lecture Notes in Computer Science*, pages 278–289. Springer, 2011.

14. Reza Rezaeian Farashahi, Pierre-Alain Fouque, Igor Shparlinski, Mehdi Tibouchi, and José Felipe Voloch. Indifferentiable deterministic hashing to elliptic and hyperelliptic curves. *Math. Comp.*, 82(281), 2013.
15. FIPS PUB 186-3. *Digital Signature Standard (DSS)*. NIST, USA, 2009.
16. N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvement and energy efficiency. White paper. http://software.intel.com/.
17. Pierre-Alain Fouque, Antoine Joux, and Mehdi Tibouchi. Injective encodings to elliptic curves. In Colin Boyd and Leonie Simpson, editors, *ACISP*, volume 7959 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2013.
18. Pierre-Alain Fouque and Mehdi Tibouchi. Indifferentiable hashing to Barreto-Naehrig curves. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012.
19. Jorge Guajardo and Christof Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. *Des. Codes Cryptography*, 25(2):207–216, 2002.
20. Darrel Hankerson, Koray Karabina, and Alfred Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. *IEEE Trans. Computers*, 58(10):1411–1420, 2009.
21. Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in $\mathrm{GF}(2^m)$ using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.
22. M. Lochter and J. Merkle. Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation. RFC 5639 (Informational), March 2010.
23. Bodo Möller. A public-key encryption scheme with pseudo-random ciphertexts. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS*, volume 3193 of *Lecture Notes in Computer Science*, pages 335–351. Springer, 2004.
24. Thomaz Oliveira, Julio López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *J. Cryptographic Engineering*, 4(1):3–17, 2014.
25. Andrew Shallue and Christiaan van de Woestijne. Construction of rational points on elliptic curves over finite fields. In Florian Hess, Sebastian Pauli, and Michael E. Pohst, editors, *ANTS*, volume 4076 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2006.
26. Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. *J. Cryptographic Engineering*, 1(3):187–199, 2011.
27. Mehdi Tibouchi. Elligator Squared: Uniform points on elliptic curves of prime order as uniform random strings. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography*, Lecture Notes in Computer Science. Springer, 2014. To appear.
28. Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the Tor anonymity system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS*, pages 109–120. ACM, 2012.
29. Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security Symposium*. USENIX Association, 2011.
30. Adam L. Young and Moti Yung. Space-efficient kleptography without random oracles. In Teddy Furon, François Cayre, Gwenaël J. Doërr, and Patrick Bas, editors, *Information Hiding*, volume 4567 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2007.
31. Adam L. Young and Moti Yung. Kleptography from standard assumptions and applications. In Juan A. Garay and Roberto De Prisco, editors, *SCN*, volume 6280 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2010.