Fast and simple constant-time hashing to the BLS12-381 elliptic curve

Riad S. Wahby and Dan Boneh

{rsw,dabo}@cs.stanford.edu
Stanford University

Abstract. Pairing-friendly elliptic curves in the Barreto-Lynn-Scott family have experienced a resurgence in popularity due to their use in a number of real-world projects. One particular Barreto-Lynn-Scott curve, called BLS12-381, is the locus of significant development and deployment effort, especially in blockchain applications. This effort has sparked interest in using BLS12-381 for BLS signatures, and in particular for aggregatable signatures, which requires hashing to one of the groups of the bilinear pairing defined by the BLS12-381 elliptic curve.

While there is a substantial body of literature on the problem of hashing to elliptic curves, much of this work does not apply to Barreto-Lynn-Scott curves. Moreover, the work that does apply has the unfortunate property that fast implementations are complex, while simple implementations are slow.

In this work, we address these issues. First, we show a straightforward way of adapting the "simplified SWU" map of Brier et al. to BLS12-381. Second, we describe optimizations to the SWU map that both simplify its implementation and improve its performance; these optimizations may be of interest in other contexts. Third, we implement and evaluate. We find that our work yields constant-time hash functions that are simple to implement, yet perform within 9% of the fastest, non–constant-time alternatives, which require much more complex implementations.

Keywords: pairing-friendly · elliptic curves · hashing · Barreto-Lynn-Scott · BLS12-381

1 Introduction

The Barreto-Lynn-Scott family of pairing-friendly elliptic curves [BLS03], and in particular the elliptic curve BLS12-381 [BLSb] (§2.1), has recently seen widespread adoption (e.g., in pairing-based SNARKs [GGPR13, PHGR13, BCTV14, Gro16]), largely because of the recent result of Kim and Barbulescu [KB16] that speeds up attacks on the discrete log problem in finite field extensions.

The availability of high-quality BLS12-381 implementations combined with the desire for aggregatable signatures [BGLS03] has sparked interest in using the BLS12-381 curve for BLS signatures [BLS01] (§2.2). The BLS signature scheme requires a hash function to points in a prime-order subgroup of a pairing-friendly curve. For this purpose, the authors suggest a method based on folklore that they call MapToGroup [BLS01, §3.3] (we call this method "hash-and-check"), which works roughly as follows: pick a random element in the elliptic curve's base field and check whether it is the x-coordinate of a rational point on the curve. If it is, return that point, otherwise try again.

While hash-and-check is simple to implement and fast in expectation, it is not without downsides. Most importantly, it is not possible to make hash-and-check run in *constant time*—that is, time independent of the hash input—with both good performance and low failure probability. For BLS signatures, a constant-time hash function is not strictly necessary for security. However, because hash-and-check on a random message takes k

checks with probability $\approx 2^{-k}$, it is relatively easy to (accidentally or adversarially) choose messages that are difficult to hash, which wastes verifiers' and signers' time. Moreover, in practice cryptographic primitives often see "mission creep," meaning that a constant-time hash function is desirable as a defense against future (mis)use.

Several lines of work in both the number theory and cryptography literature have considered the problem of deterministically mapping to rational points on elliptic curves; we briefly survey in Section 1.1. Unfortunately, most of these constructions do not apply to BLS12-381, because they are restricted to, e.g., elliptic curves of particular shapes or over base fields of specific characteristic.

One exception is the seminal work of Shallue and van de Woestijne [SvdW06] (§2.3), which applies to essentially any elliptic curve. This map can be used for BLS12-381, but fast implementations are complex, and simple ones are slow, especially for implementations with input-independent runtime (§6). At a high level, this is because evaluating the map requires evaluating Legendre symbols, for which the simple algorithm is an exponentiation (which is expensive) while the fast algorithm involves reductions modulo essentially random integers [Coh93, §1.4.2] (which entails substantial implementation complexity for good performance). Fouque and Tibouchi give an explicit construction tailored to the Barreto-Naehrig curve family, but it is undefined at several points when applied to BLS12-381, further increasing complexity (i.e., to detect and handle the undefined cases) and making implementations with input-independent runtime yet more difficult.

Ulas [Ula07] describes a simpler version of the Shallue–van de Woestijne map; Brier et al. [BCI+10, §7] give a further simplification and name it the "simplified SWU" map. This map is attractive because it has somewhat reduced computational cost and is easier to describe and implement compared to the original Shallue–van de Woestijne map. But it only applies to curves with j-invariant $\notin \{0, 1728\}$, almost surgically preventing its application to most pairing-friendly curve families (including Barreto-Lynn-Scott), which have j-invariant either 0 or 1728 for efficiency reasons [BN06, HSV06].

A second issue is that the description by Brier et al. is (somewhat artificially) restricted to curves over fields $\mathbb{F}$ where $\#\mathbb{F} \equiv 3 \bmod 4$, meaning that it does not apply to curves over extension fields of even degree (since $p^{2k} \equiv 1 \bmod 4$). This is a concern because, for Barreto-Lynn-Scott and other pairing-friendly curves, one group of the bilinear pairing is a subgroup of an elliptic curve over an even-order extension field (for BLS12-381, a quadratic extension; §2.1). Thus, the simplified SWU map as described does not work for this group.

Our contributions. We show that careful design choices and optimizations yield hash functions that admit fast, simple, and constant-time implementations. Our focus is on the BLS12-381 elliptic curve, but we describe our design methods and optimizations with an eye to straightforward application to other curves. Our specific contributions are:

- In Section 3, we give explicit Shallue and van de Woestijne maps tailored to the BLS12-381 curves. We also describe a simple method for designing exception-free maps of this type, which applies generically to other elliptic curves.

- In Section 4, we give "indirect" maps for BLS12-381 based on the simplified SWU map, which work by mapping to an isogenous curve with nonzero j-invariant, then evaluating the isogeny map. To do so, we extend the simplified SWU map to $\#\mathbb{F} \equiv 1 \bmod 4$, and thus to even-order extension fields. We also describe several optimizations that make the SWU map simpler to implement and faster to evaluate, including in constant time. Our optimizations apply generically, and can be used to speed up any implementation of the simplified SWU map.

- In Section 5 we describe explicit hash functions built on the above maps, based on known constructions. We briefly discuss security and efficiency in our context.

- In Section 6, we implement and evaluate.

  We find that, for implementations built on a rich multi-precision library like GMP [GMP] (in particular, one that supports fast reductions modulo arbitrary integers, and provides fast Legendre symbol and extended Euclidean algorithms), hashes using the map of Section 3 are up to ≈9% faster than hashes that use the map of Section 4.

  For implementations restricted to using only field operations—typical restrictions for small cryptographic libraries or for compatibility with hardware accelerators (§6)—hashes that use the map of Section 4 are ≈1.3–2× faster than hashes that use the map of Section 3. More surprisingly, our optimizations yield constant-time hashes based on the map of Section 4 that are at worst ≈9% slower than the fastest *non–constant-time* implementations of the Section 3 map.

1.1 Related work

Deterministic maps to rational points on elliptic curves. Schinzel and Skałba [SS04] give the first deterministic method of constructing rational points on elliptic curves. For curves $\mathrm{E}(\mathbb{F}) : y^2 = x^n + k$ with $x \in \{3, 4\}$ and $\mathbb{F}$ of characteristic greater than 3, their construction yields at most four points, which are parameterized by k. Skałba subsequently gives a more general construction for points on curves of the form $\mathrm{E}(\mathbb{F}) : y^2 = x^3 + ax + b$ where $a \neq 0$ and $\mathbb{F}$ has characteristic greater than 3.

Shallue and van de Woestijne [SvdW06] generalize and simplify Skałba's construction, giving concretely efficient rational maps to essentially any elliptic curve. Ulas [Ula07] further simplifies the construction, at the cost of restricting it to curves of the form $\mathrm{E}(\mathbb{F}_p) : y^2 = x^3 + ax + b, ab \neq 0$, i.e., curves with j-invariant $\notin \{0, 1728\}$; the author also generalizes the results to some hyperelliptic curves. Brier et al. [BCI+10] give yet a further simplification; and Fouque and Tibouchi [FT10] tweak this version to simplify their analysis of the size of its image. In Section 4, we further optimize this map. Fouque and Tibouchi [FT12] also analyze the original construction of Shallue and van de Woestijne specifically for the case of Barreto-Naehrig curves [BN06]; in Section 3, we give a very similar construction tailored to the BLS12-381 curve.

Icart [Ica09] describes a different approach for elliptic curves over fields of characteristic $p \equiv 2 \bmod 3$, for which Farashahi et al. [FSV09] and Fouque and Tibouchi [FT10] give improved analyses. Kammerer et al. [KLR10] generalize this approach to Hessian curves and certain hyperelliptic curves. Farashahi [Far11] also independently gives a map to Hessian curves based on Icart's approach. Couveignes and Kammerer [CK11] give a further generalization to an infinite family of such maps. Because of their restriction on field characteristic, none of these maps apply to BLS12-381.

Following the work of Farashahi [Far11], Fouque et al. [FJT13] study injective encodings to elliptic curves. Their results apply to curves $\mathrm{E}(\mathbb{F}_p), p \equiv 3 \bmod 4$ with $4 \mid \#\mathrm{E}(\mathbb{F}_p)$. Bernstein et al. [BHKL13] extend this work, giving a related encoding to all curves over fields of odd characteristic having a point of order 2. All of these restrictions rule out application to BLS12-381.

Hash functions to curves as random oracles. Brier et al. [BCI+10] study hash functions to elliptic curves in the indifferentiability framework of Maurer et al. [MRH04].[1] The authors build two indifferentiable hash functions by composing random oracles $\mathrm{H} : \{0, 1\}^\star \to \mathbb{F}$ with deterministic maps $\mathbb{F} \to \mathrm{E}(\mathbb{F})$. Their first construction is tailored to Icart's map; the second applies to essentially all known deterministic maps, but is roughly five times more costly to evaluate. We describe and evaluate both constructions in Sections 5 and 6.

[1] Informally, an implementation is indifferentiable from an ideal primitive if no probabilistic polynomial-time Turing machine can distinguish between the two, except with negligible probability.

More recently, Farashahi et al. [FFS+13] give a new analysis showing that the more efficient hash construction of Brier et al. is indifferentiable from a random oracle for essentially any deterministic map. Fouque and Tibouchi [FT12] give a different analysis for a particular version of the map of Shallue and van de Woestijne tailored to Barreto-Naehrig curves. Both of the above analyses apply to the maps of Sections 3 and 4.

Kim and Tibouchi [KT15] further improve the analysis of Farashahi et al. Most importantly for us, the authors show that replacing the random oracle H in the constructions of Brier et al. with $\hat{\mathrm{H}} : \{0,1\}^\star \to \{0,1\}^{\lfloor \log \#\mathbb{F} \rfloor}$ preserves indifferentiability. Since practical hash functions (e.g., SHA-256) produce bit strings, using $\hat{\mathrm{H}}$ results in an efficiency improvement because it only requires the hash function to produce $\lfloor \log \#\mathbb{F} \rfloor$ bits, whereas hashing to an unbiased element of $\mathbb{F}$ generally requires more.

Fast cofactor multiplication when hashing to G_2. Barreto-Lynn-Scott curves define two groups G_1 and G_2, where $\#G_1 = \#G_2$, G_1 is a subgroup of $\mathrm{E}_1(\mathbb{F}_p)$, and G_2 is a subgroup of $\mathrm{E}_2(\mathbb{F}_{p^{2k}})$ ($k = 1$ for BLS12-381). Hasse's theorem [Has] dictates that the cofactor is much larger for $\mathrm{E}_2(\mathbb{F}_{p^{2k}})$ than for $\mathrm{E}_1(\mathbb{F}_p)$, so exponentiating by this cofactor to obtain an element of G_2 is much costlier than the corresponding computation for G_1.

Scott et al. [SBC+09] show how to reduce this cost by exploiting efficiently-computable endomorphisms, building on prior methods [GLV01, IMCT02, GS08]. Fuentes-Castañeda et al. [FKR12] describe another approach to exploiting the same endomorphism, giving lower cost for some curves. Budroni and Pintore [BP17] study both methods for the Barreto-Lynn-Scott family and give explicit constructions; our hashes to G_2 (§5) use these.

2 Background

Notation. We write $\mathrm{E}(\mathbb{F})$ for the group (in multiplicative notation) of rational points on elliptic curve E over field $\mathbb{F}$ of order $\#\mathrm{E}(\mathbb{F})$. $a \,||\, b$ is the concatenation of a and b.

$\chi(\cdot)$ is a quadratic character. For $\alpha \in \mathbb{F}_p$, this is the Legendre symbol, which can be computed as $\alpha^{(p-1)/2}$. For $\delta \in \mathbb{F}_{p^2}$, this can be computed as $\delta^{(p^2-1)/2} = (\delta^p \delta)^{(p-1)/2}$, i.e., the Legendre symbol of the norm $\|\delta\|$.

$\mathrm{Sgn}_0(\beta)$ is a function that returns the "sign" of β. For $\beta \in \mathbb{F}_p$, let $\mathrm{Sgn}_0(\beta) = -1$ if $\beta > (p-1)/2$, and 1 otherwise. For $\gamma \in \mathbb{F}_{p^2}$, let $\mathrm{Sgn}_0(\gamma) = \mathrm{Sgn}_0((\gamma^p + \gamma)/2 \in \mathbb{F}_p)$, i.e., the sign of the "first coordinate" of γ written in the canonical power basis.

We regard the square root in $\mathbb{F}$ as a function, so we fix a canonical representation. For $\mathbb{F}_p, p \equiv 3 \bmod 4$, $\sqrt{\alpha} \triangleq \alpha^{(p+1)/4} \in \mathbb{F}_p$. Otherwise, $\beta \triangleq \sqrt{\alpha} \in \mathbb{F}$ such that $\mathrm{Sgn}_0(\beta) = 1$.

Jacobian coordinates. It is often convenient to represent points (x, y) on $\mathrm{E}(\mathbb{F})$ in Jacobian projective coordinates $(X : Y : Z)$, which are defined as follows:

$$(x, y) \mapsto (x : y : 1) \qquad (X : Y : Z) \mapsto (X/Z^2, Y/Z^3)$$

Projective coordinates are generally useful to avoid computing inversions, and Jacobian coordinates give some of the fastest group operations for this curve shape [EFD]. Moreover, the point addition law in this representation is independent of the coefficients of the curve equation. Looking ahead, this fact will be useful for SWU-based hash functions (§4, §5).

2.1 The BLS12-381 elliptic curve

BLS12-381 [BLSb] is a pairing-friendly elliptic curve in the Barreto-Lynn-Scott family [BLS03], which defines a bilinear group pair G_1, G_2 [BLS01, Definition 2.2]. G_1 is the order-q subgroup of $\mathrm{E}_1(\mathbb{F}_p) : y^2 = x^3 + 4$, $\#\mathrm{E}_1(\mathbb{F}_p) = h_1 q$, where

$$p = \texttt{0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f624}$$
$$\texttt{1eabfffeb153ffffb9feffffffffaaab}$$
$$q = \texttt{0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001}$$
$$h_1 = \texttt{0x396c8c005555e1568c00aaab0000aaab}$$

G_2 is the order-q subgroup of $\mathrm{E}_2(\mathbb{F}_{p^2}) : y^2 = x^3 + 4(1+\sqrt{-1})$, where $\mathbb{F}_{p^2} \triangleq \mathbb{F}_p[\sqrt{-1}]/(x^2+1)$. $\#\mathrm{E}_2(\mathbb{F}_{p^2}) = h_2 q$, where

$$h_2 = \texttt{0x5d543a95414e7f1091d50792876a202cd91de4547085abaa68a205b2e5a7ddfa}$$
$$\texttt{628f1cb4d9e82ef21537e293a6691ae1616ec6e786f0c70cf1c38e31c7238e5}$$

2.2 BLS signatures

This description follows the one due to Boneh et al. [BLS01]; it uses the following primitives:

- G_1 and G_2 are cyclic groups of prime order q with generators g_1 and g_2, respectively.
- $\psi : G_2 \to G_1$ is an efficiently computable isomorphism such that $\psi(g_2) = g_1$. ψ may not exist, in which case a stronger assumption is necessary for security [BDN18].
- $e : G_1 \times G_2 \to G_T$ is a bilinear map.
- The groups G_1, G_2 and the maps ψ, e comprise a bilinear group pair.
- $\mathrm{H} : \{0,1\}^\star \to G_1$ is a hash function modeled as a random oracle.

The BLS signature scheme is the triple of algorithms $(\text{gen}^{\text{BLS}}, \text{sign}^{\text{BLS}}, \text{verify}^{\text{BLS}})$ defined as

$$\text{gen}^{\mathsf{BLS}}() \to (pk, sk) \text{ : Sample } x \xleftarrow{\text{R}} \{0, \ldots, q-1\} \text{ and output } (g_2^x, x).$$
$$\text{sign}^{\mathsf{BLS}}(sk, msg) \to sig \text{ : Output } \mathrm{H}(msg)^{sk} \in G_1.$$
$$\text{verify}^{\mathsf{BLS}}(pk, msg, sig) \to \{\mathsf{OK}, \bot\} \text{ : } \mathsf{OK} \text{ if } e(\mathrm{H}(msg), pk) = e(sig, g_2) \text{, else } \bot.$$

It is also possible to swap the functions of G_1 and G_2 in the above. This reduces the size of public keys at the cost of longer signatures; in practice, when using aggregatable signatures this tradeoff may be desirable. In this case, signing and verifying requires a hash function $\mathrm{H} : \{0,1\}^\star \to G_2$. This work considers hashing to both G_1 and G_2.

2.3 The Shallue–van de Woestijne map

For any elliptic curve $\mathrm{E}(\mathbb{F}) : y^2 = f(x) = x^3 + ax + b$, $\#\mathbb{F} > 5$, Shallue and van de Woestijne give a map from $L \subseteq \mathbb{F}$ to the curve $\mathrm{E}(\mathbb{F})$ [SvdW06]. They observe, generalizing and simplifying the result of Skałba [Ska05], that for any rational point on the threefold

$$V(\mathbb{F}) : f(x_1) f(x_2) f(x_3) = x_4^2$$

such that $x_4 \neq 0$, at least one of $f(x_j), j \in \{1,2,3\}$ must be a square. This implies that one of the x_j is the x-coordinate of a rational point on $\mathrm{E}(\mathbb{F})$.

To construct a rational point on $V(\mathbb{F})$, the authors define the surface $S(\mathbb{F})$ and the rational map $\phi_1 : S(\mathbb{F}) \mapsto V(\mathbb{F})$, which is invertible on its image [SvdW06, Lemma 6]:

$$S(\mathbb{F}) : y^2 \left(u^2 + uv + v^2 + a\right) = -f(u)$$
$$\phi_1 : (u, v, y) \mapsto \left(v, -u-v, u+y^2, f(u+y^2) \cdot \frac{y^2 + uv + v^2 + a}{y}\right).$$

Next, the authors observe [SvdW06, Lemma 7] that fixing $u = u_0$ satisfying $f(u_0) \neq 0$ and $3u_0^2 + 4a \neq 0$ specializes $S(\mathbb{F})$ to a curve that is birational to a conic with a rational parameterization. This gives a rational map $\phi_2 : \mathbb{A}^1 \mapsto S(\mathbb{F})$ that is invertible on its image.

Putting it all together: define $L = \{t \in \mathbb{F} : \phi_1(\phi_2(t)) \text{ is defined}\}$. Then, to map $t \in L$ to $\mathrm{E}(\mathbb{F})$, first compute $\phi_1(\phi_2(t))$, which is a rational point (x_1, x_2, x_3, x_4) on $V(\mathbb{F})$, so at least one $f(x_j), j \in \{1, 2, 3\}$ is square. Choose the smallest i where this is the case, compute the corresponding y-coordinate, and return (x_j, y).

2.4 The simplified Shallue–van de Woestijne–Ulas map of Brier et al.

Brier et al. [BCI+10] define the simplified SWU map (with a small modification due to Fouque and Tibouchi [FT10]) as follows. Let $\mathrm{E}(\mathbb{F}_p) : y^2 = g(x) = x^3 + ax + b$, $ab \neq 0$, $p \equiv 3 \bmod 4$, and

$$X_0(t) = -\frac{b}{a}\left(1 + \frac{1}{t^4 - t^2}\right) \qquad X_1(t) = -t^2 X_0(t)$$

Then the simplified SWU map is given by

$$t \mapsto \begin{cases} \infty & \text{if } t \in \{-1, 0, 1\} \\ \left(X_0(t), \sqrt{g(X_0(t))}\right) & \text{if } \chi(g(X_0(t))) = 1 \\ \left(X_1(t), -\sqrt{g(X_1(t))}\right) & \text{otherwise} \end{cases}$$

To see why this map works, assume u is non-square and assume we have x such that $g(ux) = u^3 g(x)$. Since u is non-square, either $g(x)$ or $g(ux)$ is square, and thus either x or ux is the x-coordinate of a rational point on $\mathrm{E}(\mathbb{F}_p)$. Expanding and solving for x,

$$\begin{aligned} u^3x^3 + aux + b &= u^3(x^3 + ax + b) \\ aux + b &= u^3(ax + b) \\ x &= -\frac{b}{a}\frac{u^3 - 1}{u^3 - u} = -\frac{b}{a}\left(\frac{u^3 - u - 1}{u^3 - u} + \frac{u}{u^3 - u}\right) = -\frac{b}{a}\left(1 + \frac{u - 1}{u\,(u^2 - 1)}\right) \\ &= -\frac{b}{a}\left(1 + \frac{1}{u^2 + u}\right) \end{aligned}$$

Since $p \equiv 3 \bmod 4$, -1 is non-square, so $u = -t^2$ is, too. Substituting yields X_0 and X_1.

3 A Shallue–van de Woestijne map for BLS12-381

We construct an explicit Shallue–van de Woestijne map (§2.3) for the BLS12-381 curves $\mathrm{E}_1(\mathbb{F}_p)$ and $\mathrm{E}_2(\mathbb{F}_{p^2})$ (§2.1), Our description follows the one by Fouque and Tibouchi [FT12].

For both of the BLS12-381 curves, we have $\mathrm{E}(\mathbb{F}) : y^2 = f_i(x) = x^3 + b_i$ (§2.1), so we restrict our analysis to the case $a = 0$. For now, we work with $S(\mathbb{F})$ generically in terms of $u = u_0$; we discuss convenient choices of u_0 below. Rewriting [SvdW06, Lemma 7]:

$$y^2\left(\frac{3}{4}u_0^2 + \left(v + \frac{u_0}{2}\right)^2\right) = -f_i(u_0)$$

$$z^2 + f_i(u_0)w^2 = -\frac{3}{4}u_0^2 \qquad \text{where } z = v + \frac{u_0}{2}, \quad w = \frac{1}{y}$$

-3 is square in $\mathbb{F}_p$ (and thus $\mathbb{F}_{p^2}$), so $(z_0, w_0) = (\sqrt{-3u_0^2/4}, 0)$ is a rational point on this conic. Parameterizing in t by setting $z = z_0 + tw$ and substituting gives

$$t\sqrt{-3u_0^2} + (t^2 + f_i(u_0))w = 0 \qquad w \neq 0$$

Solving for y and v,

$$y = \frac{1}{w} = -\frac{t^2 + f_i(u_0)}{t\sqrt{-3u_0^2}}$$

$$v = z_0 + tw - \frac{u_0}{2} = \frac{\sqrt{-3u_0^2} - u_0}{2} - \frac{t^2\sqrt{-3u_0^2}}{t^2 + f_i(u_0)}$$

Finally, from the map ϕ_1 (§2.3), we have

$$x_1 = v = \frac{\sqrt{-3u_0^2} - u_0}{2} - \frac{t^2\sqrt{-3u_0^2}}{t^2 + f_i(u_0)}$$

$$x_2 = -u_0 - v = \frac{t^2\sqrt{-3u_0^2}}{t^2 + f_i(u_0)} - \frac{\sqrt{-3u_0^2} + u_0}{2}$$

$$x_3 = u_0 + y^2 = u_0 - \frac{\left(t^2 + f_i(u_0)\right)^2}{3u_0^2 t^2}$$

This map is undefined when $t = 0$ or $t^2 + f_i(u_0) = 0$. To avoid this issue, Fouque and Tibouchi [FT12] choose u_0 such that $-f_i(u_0)$ is non-square (ensuring that $t^2 + f_i(u_0) \neq 0$) and add a special case for $t = 0$. We take a slightly different approach.

First, notice that, once we have fixed u_0, we can evaluate the map using only one inversion by applying Montgomery's trick [Mon87], i.e., inverting the product $t^2\left(t^2 + f_i(u_0)\right)$. Evaluating the x_j then entails a few inexpensive arithmetic operations involving t^2 and precomputed constants. Computing Legendre symbols and a square root yields y.

Now we can handle the exceptional cases. Notice that when applying Montgomery's trick, the map is undefined just when the value being inverted is zero. If we use an inversion algorithm that returns zero on input zero (which is true, e.g., for inversion by Fermat's little theorem), the resulting value of x_1 will be $x_0 \triangleq (\sqrt{-3u_0^2} - u_0)/2$. Choosing u_0 such that $f_i(x_0)$ is square then yields an exception-free map. For $\mathrm{E}_1(\mathbb{F}_p)$, the smallest u_0 in absolute value for which this holds is $u_0 = -3$; for $\mathrm{E}_2(\mathbb{F}_{p^2})$, it is $u_0 = -1$.

Finally, Fouque and Tibouchi observe that the x_j values depend only on t^2, i.e., t and $-t$ map to the same x-coordinate. They suggest choosing the sign of y to be the sign of $\chi(t)$, but this costs an extra Legendre symbol evaluation and does not work in $\mathbb{F}_{p^2}$ (because -1 is square). A more easily computed choice that works for both $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ is $y = \mathrm{Sgn}_0(t) \cdot \sqrt{f_i(x_j)}$. This fully specifies both maps.

Putting it all together. Precompute $f_i(u_0)$, $(\sqrt{-3u_0^2} \pm u_0)/2$, $\sqrt{-3u_0^2}$, and $1/3u_0^2$. On input t, compute $\alpha = 1/(t^2\left(t^2 + f_i(u_0)\right)$ setting $\alpha = 0$ if $t^2\left(t^2 + f_i(u_0)\right) = 0$. Use α to compute x_1, x_2, and x_3 (e.g., $x_1 = (\sqrt{-3u_0^2} - u_0)/2 - \alpha t^4\sqrt{-3u_0^2}$). Choose the smallest $j \in \{1, 2, 3\}$ such that $\chi(f_i(x_j)) = 1$, compute $y = \mathrm{Sgn}_0(t) \cdot \sqrt{f_i(x_j)}$, and return (x_j, y).

Computing this map in constant time requires evaluating all x_j and all $\chi(f_i(x_j))$, all in constant time. Note that fast Legendre symbol and inversion algorithms [Coh93, §1.3.2, §1.4.2] are *not* constant time. Fouque and Tibouchi suggest blinding the Legendre symbol by choosing random r_j and computing $\chi(r_j^2 f_i(x_j))$ [FT12, §6]; a similar trick for inversion is standard (to invert β, choose random r, invert $r\beta$ and then multiply by r). Of course, computing Legendre symbols or inversions by exponentiation is easily made constant time, but is also far more costly. Our constant-time implementations using only field operations take an approach that we describe in the Section 4; we discuss specifics in Section 6.

4 An optimized SWU map for BLS12-381

The simplified SWU map of Brier et al. [BCI+10] (§2.4) applies only to curves of the form $\mathrm{E}(\mathbb{F}) : y^2 = g(x) = x^3 + ax + b$ where $ab \neq 0$ and $\#\mathbb{F} \equiv 3 \bmod 4$. For BLS12-381 (§2.1), $\mathrm{E}_1(\mathbb{F}_p)$ meets the second requirement but not the first; $\mathrm{E}_2(\mathbb{F}_{p^2})$ meets neither requirement. As a result, this map cannot be applied directly to either curve.

In this section we show how to solve these problems. To avoid the requirement that $\#\mathbb{F} \equiv 3 \bmod 4$, we tweak the map's definition. To sidestep the issue that $a = 0$ for both curves, we construct an "indirect" map, with two steps: first, map to some $\mathrm{E}'(\mathbb{F})$ isogenous to $\mathrm{E}(\mathbb{F})$ with $ab \neq 0$, and second, apply the isogeny map to obtain a point on $\mathrm{E}(\mathbb{F})$.

One potential issue with the indirect approach is that, for an isogeny of degree d, the resulting map is to the points $\{P^d : P \in \mathrm{E}(\mathbb{F})\}$—that is, it maps only to a *subset* of $\mathrm{E}(\mathbb{F})$. Our concern is twofold: first, recall from Section 2.2 that our goal is to hash to a subgroup of $\mathrm{E}(\mathbb{F})$, in particular the order-q subgroup whose elements are $\{P^h : P \in \mathrm{E}(\mathbb{F})\}$, where $\#\mathrm{E}(\mathbb{F}) = hq$. Second, in Section 5 we will construct hash functions that rely on the statistical properties of the SWU map, so we must be sure that the indirect approach does not alter those properties.

Fortunately, both concerns are easily dispensed with. For the first, we choose an isogeny of degree d coprime to q; exponentiation by h gives $\{P^{hd} : P \in \mathrm{E}(\mathbb{F})\} \simeq \{P^h : P \in \mathrm{E}(\mathbb{F})\}$. For the second, Boneh and Franklin [BF01, Lemma 5.1] and, in the same vein, Brier et al. [BCI+10, Lemma 13] show that, as long as $(hq/d) \nmid d$, the relevant statistical properties of the map are preserved. In practice, both requirements are easily met.

In the remainder of this section, we generalize the SWU map to $\#\mathbb{F} \not\equiv 3 \bmod 4$; describe two optimizations that make the map simpler to implement and faster to evaluate; give explicit curves $\mathrm{E}'_1(\mathbb{F}_p)$ 11-isogenous to $\mathrm{E}_1(\mathbb{F}_p)$ and $\mathrm{E}'_2(\mathbb{F}_{p^2})$ 3-isogenous to $\mathrm{E}_2(\mathbb{F}_{p^2})$; and give one further small hint for evaluating the isogeny maps quickly. Finally, we put all of the above together, yielding the SWU maps to $\mathrm{E}_1(\mathbb{F}_p)$ and $\mathrm{E}_2(\mathbb{F}_{p^2})$.

4.1 Generalizing the map to $\mathrm{E}(\mathbb{F})$, $\#\mathbb{F} \not\equiv 3 \bmod 4$

Recall from Section 2.4 that for an elliptic curve $\mathrm{E}'(\mathbb{F}) : y^2 = g(x) = x^3 + ax + b$, the SWU map uses a parameterization of x in terms of u such that $g(ux) = u^3 g(x)$, for u a non-square. When $\#\mathbb{F} \equiv 3 \bmod 4$ (as in §2.4), -1 is non-square, so $u = -t^2$ is a convenient choice. More generally, for some non-square $\xi \in \mathbb{F}$, let $u = \xi t^2$. Then we have

$$g(X_0(t)) \cdot g(X_1(t)) = (g(X_0(t)))^2 \, \xi^3 t^6 = \left(t^3 g(X_0(t))\right)^2 \xi^3$$

Since ξ^3 and thus the right-hand side are non-square, exactly one of $g(X_0(t))$ and $g(X_1(t))$ must be square, and thus either $X_0(t)$ or $X_1(t) = \xi t^2 X_0(t)$ is the x-coordinate of a rational point on $\mathrm{E}'(\mathbb{F})$. In the next section, it will be convenient for $g(b/\xi a)$ to be square. For $\mathrm{E}'_1(\mathbb{F}_p)$, $\xi_1 = -1 \in \mathbb{F}_p$ satisfies this requirement; for $\mathrm{E}'_2(\mathbb{F}_{p^2})$, $\xi_2 = 1 + \sqrt{-1} \in \mathbb{F}_{p^2}$ does.

4.2 Optimizing the map

As described in Section 2.4, the SWU map requires computing a field inversion, a quadratic character, and a square root. We now describe how to avoid computing both the inversion and the quadratic character in a way that is amenable to constant-time implementation. We describe each optimization for $\mathrm{E}'_1(\mathbb{F}_p)$, then show how they apply to $\mathrm{E}'_2(\mathbb{F}_{p^2})$.

Notation. In this section we use the generic $g(\cdot)$, a, b, and ξ in expressions that apply to both $\mathrm{E}'_1(\mathbb{F}_p)$ and $\mathrm{E}'_2(\mathbb{F}_{p^2})$. These correspond to the curve equation $y^2 = g(x) = x^3 + ax + b$ and the map's distinguished non-square. We give the ξ_i, a_i, and b_i in Sections 4.1 and 4.3.

Eliminating the quadratic character computation. For $\mathrm{E}_1'(\mathbb{F}_p)$, $p \equiv 3 \bmod 4$, so one can evaluate the y-coordinate as follows: compute $\alpha = g_1(X_0(t))^{(p+1)/4}$. If $g_1(X_0(t))$ is square, α is its square root; this is easily checked by comparing α^2 with $g_1(X_0(t))$. If they are equal, α is the y-coordinate (up to sign; §4.4). Otherwise, compute the y-coordinate as $\sqrt{g_1(X_1(t))} = t^3\alpha$. To see why this works, assume $g_1(X_0(t))$ is non-square in $\mathbb{F}_p$; then

$$\left(t^3 \cdot g_1(X_0(t))^{(p+1)/4}\right)^2 = t^6 \cdot g_1(X_0(t)) \cdot g_1(X_0(t))^{(p-1)/2} = -t^6 \cdot g_1(X_0(t))$$

The final equality comes from the fact that $g_1(X_0(t))^{(p-1)/2} = \chi(g_1(X_0(t))) = -1$. Recall (§2.4) that $g(ux) = u^3 g(x)$, $X_1(t) = uX_0(t)$, and $u = -t^2$, establishing the claim.

Avoiding inversions. We borrow and adapt a trick from Bernstein et al. [BDL+12, §5]:

$$(U/V)^{(p+1)/4} = U^{(p+1)/4} \cdot V^{p-1-(p+1)/4} = U^{(p+1)/4} \cdot V^{(3p-5)/4} = UV\left(UV^3\right)^{(p-3)/4}$$

We can rewrite $g(X_0(t))$ into the required form as follows:

$$X_0(t) \triangleq \frac{N}{D} = \frac{b\left(\xi^2 t^4 + \xi t^2 + 1\right)}{-a\left(\xi^2 t^4 + \xi t^2\right)} \qquad g(X_0(t)) \triangleq \frac{U}{V} = \frac{N^3 + aND^2 + bD^3}{D^3}$$

The above is undefined just when $D = 0$. To facilitate constant-time evaluation in this case, set N and D to known good values and continue. Recall (§4.1) that we chose ξ such that $g(b/\xi a)$ is square, and notice that if $D = 0$, then $N = b$ (since $\xi^2 t^4 + \xi t^2 = 0$). Thus, all that is required to recover from the exception is to set $D = \xi a$.

Finally, the x-coordinate is either N/D or $\xi t^2 N/D$. To avoid inverting D, return a point in Jacobian projective coordinates (§2); we give specifics in Section 4.4.

The $\mathrm{E}_2'(\mathbb{F}_{p^2})$ case is only slightly more complicated. In particular, we need to show both how to take one square root in $\mathbb{F}_{p^2}$ in constant time, and how to avoid taking a second. For $p^2 \equiv 9 \bmod 16$, we recall a standard trick for computing $\sqrt{\delta} \in \mathbb{F}_{p^2}$. For square δ, define

$$\gamma = \delta^{(p^2+7)/16} = \left(\delta \cdot \delta^{(p^2-1)/8}\right)^{1/2} = \delta^{1/2}\left(\delta^{(p^2-1)/2}\right)^{1/8} = \delta^{1/2} \cdot 1^{1/8}$$

The final equality comes from the fact that $\delta^{(p^2-1)/2} = \chi(\delta) = 1$. This implies that $\sqrt{\delta}$ must be one of $\pm\gamma$, $\pm\gamma\sqrt{-1}$, or $\pm\gamma\sqrt{\pm\sqrt{-1}}$ (all exist). Exponentiating and checking in constant time is straightforward if the three constants are precomputed.[2]

This gives us an analogous approach to the $\mathrm{E}_1'(\mathbb{F}_p)$ case: compute $\gamma = g_2(X_0(t))^{(p^2+7)/16}$ and check the four possible square roots. If none is found, $g_2(X_0(t))$ is non-square, and one of the four possible values $t^3\gamma\eta = \sqrt{g_2(X_1(t))}$, where $\eta^2 = \xi_2^3\,(-1)^{-1/4}$. This is because

$$\left(\eta t^3 \cdot g_2(X_0(t))^{(p^2+7)/16}\right)^2 = \eta^2 t^6 \cdot g_2(X_0(t)) \cdot g_2(X_0(t))^{(p-1)/8} = \eta^2 t^6 \cdot g_2(X_0(t)) \cdot (-1)^{1/4}$$

Recalling (§4.1) that $g(X_1(t)) = \xi^3 t^6 g(X_0(t))$, we have $\eta^2 = \xi_2^3\,(-1)^{-1/4}$ as claimed. Note that $(-1)^{-1/4}$ and ξ^3 are non-square in $\mathbb{F}_{p^2}$, so four unique values of η exist (along with their negations). Once again, for efficiency these values should be precomputed.

Finally, the above requires computing $\gamma = (U/V)^{(p^2+7)/16}$ for U and V as above. In this case, the trick of Bernstein et al. yields $\gamma = UV^7\left(UV^{15}\right)^{(p^2-9)/16}$, avoiding an inversion.

[2] Adj and Rodríguez-Henríquez describe a different algorithm for computing square roots in $\mathbb{F}_{p^2}$, $p \equiv 3 \bmod 4$ [AR12], which essentially always requires two exponentiations in $\mathbb{F}_{p^2}$ with exponents of size $\approx \log p$. The method we describe uses one exponentiation in $\mathbb{F}_{p^2}$ with an exponent of size $\approx 2\log p$. The costs are almost the same; the method we describe is slightly easier to implement in constant time.

4.3 The isogeny maps

Recall from above that we want an isogeny of degree d coprime to q. Small d is best for efficiency. For $\mathrm{E}_1(\mathbb{F}_p)$, the smallest prime d giving a curve in the base field for which $a \neq 0$ is 11; thus, we use the 11-isogenous curve[3] $\mathrm{E}'_1(\mathbb{F}_p) : y^2 = x^3 + a_1 x + b_1$ where

$$a_1 = \texttt{0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd881ac98}$$
$$\texttt{936f8da0e0f97f5cf428082d584c1d}$$
$$b_1 = \texttt{0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fcef35ef5}$$
$$\texttt{5a23215a316ceaa5d1cc48e98e172be0}$$

The 11-isogeny from $\mathrm{E}'_1(\mathbb{F}_p)$ to $\mathrm{E}_1(\mathbb{F}_p)$ is given by the following map:

$$(x, y) \mapsto \left(\frac{\sum_{i=0}^{11} k_{1,i} x^i}{\sum_{i=0}^{10} k_{2,i} x^i}, \quad y \frac{\sum_{i=0}^{15} k_{3,i} x^i}{\sum_{i=0}^{15} k_{4,i} x^i} \right)$$

The $\mathrm{E}_2(\mathbb{F}_{p^2})$ case is much nicer: there is a 3-isogenous curve $\mathrm{E}'_2(\mathbb{F}_{p^2}) : y^2 = x^3 + a_2 x + b_2$ where $a_2 = 240\sqrt{-1}$ and $b_2 = 1012 + 1012\sqrt{-1}$. The 3-isogeny from $\mathrm{E}'_2(\mathbb{F}_{p^2})$ to $\mathrm{E}_2(\mathbb{F}_{p^2})$ is given by a map similar to the map for $\mathrm{E}'_1(\mathbb{F}_p)$, but with polynomials of degree at most 3. Appendix A gives a Sage [SM] script that constructs both isogeny maps.

Above, we mentioned that returning a point in Jacobian projective coordinates saves an inversion when computing the SWU map. Fortunately, it is easy to evaluate the isogeny maps on points in Jacobian coordinates without an inversion. For example, given $(X : Y : Z)$ on $\mathrm{E}'_1(\mathbb{F}_p)$, where $x = X/Z^2$ and $y = Y/Z^3$, we evaluate the isogeny map to give a point $(X_o : Y_o : Z_o)$ in Jacobian coordinates on $\mathrm{E}_1(\mathbb{F}_p)$, as follows.

Rewriting the x-coordinate map given above in terms of the projective coordinates:

$$\frac{X}{Z^2} \mapsto \frac{N_x}{D_x} \triangleq \frac{\sum_{i=0}^{11} k_{1,i} \left(\frac{X}{Z^2}\right)^i}{\sum_{i=0}^{10} k_{2,i} \left(\frac{X}{Z^2}\right)^i} = \frac{\sum_{i=0}^{11} k_{1,i} \left(Z^2\right)^{11-i} X^i}{Z^2 \sum_{i=0}^{10} k_{2,i} \left(Z^2\right)^{10-i} X^i}$$

Similarly, for the y-coordinate map:

$$\frac{Y}{Z^3} \mapsto \frac{N_y}{D_y} \triangleq \frac{Y}{Z^3} \frac{\sum_{i=0}^{15} k_{3,i} \left(\frac{X}{Z^2}\right)^i}{\sum_{i=0}^{15} k_{4,i} \left(\frac{X}{Z^2}\right)^i} = \frac{Y \sum_{i=0}^{15} k_{3,i} \left(Z^2\right)^{15-i} X^i}{Z^3 \sum_{i=0}^{15} k_{4,i} \left(Z^2\right)^{15-i} X^i}$$

To evaluate the above maps, first compute $Z^{2i}, i \in \{1, \ldots, 15\}$. Then, to evaluate, for example, the numerator of the X map, compute the products $k_{1,i} \left(Z^2\right)^{11-i}, i \in \{1, \ldots, 11\}$ and then evaluate the polynomial using Horner's method [Knu97].

Finally, after evaluating the numerator and denominator of each map, compute

$$Z_o = D_x D_y \qquad X_o = N_x D_y Z_o \qquad Y_o = N_y D_x Z_o^2$$

Then $y_o = Y_o / Z_o^3 = N_y / D_y$ and $x_o = X_o / Z_o^2 = N_x / D_x$, as required.

---

[3] Alternatively, we might map to a curve isogenous to E_1 over an extension field in which there are isogenies of lower degree. But this entails costlier arithmetic (in the extension), so it is likely no cheaper.

4.4 Putting it all together

We slightly modify the SWU map of Section 2.4. In particular, since $X_0(t)$ and $X_1(t)$ only depend on t^2, we can arbitrarily choose the sign of the resulting y-coordinate; we set $y = \mathrm{Sgn}_0(t) \cdot \sqrt{g(X_j(t))}$, which we justified in Section 3.

Thus, the map to a point on $\mathrm{E}_1(\mathbb{F}_p)$ is computed as follows on input $t \in \mathbb{F}_p$:

(1) Compute N, D, U, V (§4.2) and $\alpha \triangleq UV\left(UV^3\right)^{(p-3)/4}$.

(2) If $\alpha^2 V - U = 0$, then $g_1(X_0(t))$ is square in $\mathbb{F}_p$, so $(x, y) = (N/D, \mathrm{Sgn}_0(t) \cdot \alpha) \in \mathrm{E}'_1(\mathbb{F}_p)$.

Set $(X : Y : Z) = (ND : \mathrm{Sgn}_0(t) \cdot \alpha D^3 : D)$.

(3) Otherwise, $g_1(X_1(t))$ is square in $\mathbb{F}_p$, so $(x, y) = (\xi t^2 N/D, t^3\alpha) \in \mathrm{E}'_1(\mathbb{F}_p)$. (No $\mathrm{Sgn}_0(\cdot)$ is necessary in this case, because multiplication by t^3 preserves the sign of t.)

Set $(X : Y : Z) = (\xi t^2 ND : t^3 \alpha D^3 : D)$.

(4) Evaluate the 11-isogeny map on $(X : Y : Z)$ and return the resulting point.

The map to $\mathrm{E}_2(\mathbb{F}_{p^2})$ is analogous.

As a final optimization, if one is computing this map several times and summing the result (§5), one can avoid repeatedly evaluating the isogeny map by summing the points on $\mathrm{E}'_1(\mathbb{F}_p)$ or $\mathrm{E}'_2(\mathbb{F}_{p^2})$ and then applying the isogeny map to the sum. As mentioned in Section 2, the point addition law in Jacobian coordinates is independent of the coefficients in the curve equation, meaning that one can use the same point addition (but not doubling) routine for $\mathrm{E}_1(\mathbb{F}_p)$ and $\mathrm{E}'_1(\mathbb{F}_p)$, and similarly for $\mathrm{E}_2(\mathbb{F}_{p^2})$ and $\mathrm{E}'_2(\mathbb{F}_{p^2})$ [EFD].

5 Hashing to the groups G_1 and G_2 of BLS12-381

The following five hash functions use the maps of Sections 3 and 4 to output points in G_1 or G_2 of BLS12-381 (§2.1). The first and fourth are trivial. The second, third, and fifth are based on the work of Brier et al. [BCI+10] and Farashahi et al. [FFS+13], who show that all three are indifferentiable from a random oracle (§1.1).

The hash functions build on these primitives:[4]

- $\mathrm{H}_p : \{0,1\}^\star \to \mathbb{F}_p$, $\mathrm{H}_{p^2} : \{0,1\}^\star \to \mathbb{F}_{p^2}$, and $\mathrm{H}_q : \{0,1\}^\star \to \mathbb{F}_q$ are random oracles.
- $\mathrm{Map}_1 : \mathbb{F}_p \to \mathrm{E}_1(\mathbb{F}_p)$ and $\mathrm{Map}_2 : \mathbb{F}_{p^2} \to \mathrm{E}_2(\mathbb{F}_{p^2})$ can be either of the maps of Sections 3 and 4, to G_1 and G_2 respectively.
- h_1 is the cofactor of the BLS12-381 $\mathrm{E}_1(\mathbb{F}_p)$ elliptic curve group (§2.1).
- $\Psi : \mathrm{E}_2(\mathbb{F}_{p^2}) \to \mathrm{E}_2(\mathbb{F}_{p^2})$ is the endomorphism of Budroni and Pintore [BP17, §4.1], based on the work of Scott et al. [SBC+09] and Fuentes-Castañeda et al. [FKR12]. This is effectively a fast method of multiplying by the cofactor h_2 of $\mathrm{E}_2(\mathbb{F}_{p^2})$.

[4]For constant-time hash functions, these primitives must be constant time, too. Reducing a $2\log p$–bit integer modulo p gives an element of $\mathbb{F}_p$ with negligible bias (and analogously for H_q). Alternatively, Kim and Tibouchi [KT15] show that H_p can be replaced with $\mathrm{H}_{\hat{p}} : \{0,1\}^\star \to \{0,1\}^{\lfloor \log p \rfloor}$. H_{p^2} can be implemented via two evaluations of H_p or $\mathrm{H}_{\hat{p}}$. All of these are easily implemented by seeding a PRG with a hash of the input; AES-CTR and ChaCha20 [Ber08] are good choices.

Ψ is a straight-line computation, so it is constant time provided that the underlying field operations are.

Construction #1. The hash function $\mathrm{H}_1 : \{0,1\}^\star \to G_1$ is given by

$$\mathrm{H}_1(msg) \triangleq \mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1}$$

Remark 1. Since the maps of Sections 3 and 4 are to to a subset of $\mathrm{E}_1(\mathbb{F}_p)$, and since both maps are invertible on their image, this function is easily distinguished from a random oracle. We note that a random oracle is not necessary for BLS signatures—hashing to a constant fraction of the points in G_1 suffices (see also [BCI+10, Section 5.2]). Still, we recommend using one of the other hash functions in this section, for three reasons. First, a random oracle simplifies the BLS security proof. Second, an indifferentiable hash function is suitable for uses beyond signatures (another defense against "mission creep"; §1). And third, since one exponentiation (by h_1) is anyway necessary to obtain a point in G_1, the overheads of the other hash functions in this section are small in practice (§6).

Construction #2. The hash function $\mathrm{H}_2 : \{0,1\}^\star \to G_1$ is given by

$$\mathrm{H}_2(msg) \triangleq \left(\mathrm{Map}_1(\mathrm{H}_p(msg \,||\, 0)) \cdot \mathrm{Map}_1(\mathrm{H}_p(msg \,||\, 1))\right)^{h_1}$$

Remark 2. As mentioned in Section 4.4, when using that section's map it is most efficient to evaluate the map twice, sum the results on $\mathrm{E}_1'(\mathbb{F}_p)$, and then apply the isogeny map.

Construction #3. Let g be a generator of G_1 with unknown discrete logarithm relation to the base point g_1 of the BLS signature scheme (§2.2). The hash function $\mathrm{H}_3 : \{0,1\}^\star \to G_1$ is given by

$$\mathrm{H}_3(msg) \triangleq \mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1} \cdot g^{\mathrm{H}_q(msg)}$$

Remark 3. For BLS12-381, h is a 126-bit number, while q is 255 bits. This means that directly using multi-exponentiation [Möl01] to compute H_3 has limited efficacy in reducing cost, since computing $g^{\mathrm{H}_q(msg)}$ requires about twice as many squarings as computing $\mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1}$. This can be addressed straightforwardly as follows. First, compute r_1 and r_2 at most 128 bits such that $\mathrm{H}_q(msg) = r_2 \cdot 2^{128} + r_1$. Then, letting $g_{128} = g^{2^{128}}$ (which can be precomputed because g is fixed),

$$\mathrm{H}_3(msg) \triangleq \mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1} \cdot g^{r_1} \cdot g_{128}^{r_2}$$

This reduces the number of squarings because h_1, r_1, and r_2 are nearly the same bit length.

Remark 4. It is very important that the discrete log relation between the generator g and the base point g_1 for the BLS signature scheme is unknown. To see why, assume the discrete log of g base g_1 is ℓ, and recall (§2.2) that for key $(pk, sk) = (g_2^x, x)$, a signature on msg is given by $\mathrm{H}_3(msg)^x$. Then

$$\begin{aligned}\mathrm{H}_3(msg)^x &= \mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1 \cdot x} \cdot g^{\mathrm{H}_q(msg) \cdot x} \\ &= \mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1 \cdot x} \cdot ((g_1^\ell)^x)^{\mathrm{H}_q(msg)} \\ &= \mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1 \cdot x} \cdot (\psi(pk)^\ell)^{\mathrm{H}_q(msg)}\end{aligned}$$

Knowing ℓ renders moot the multiplication by $g^{\mathrm{H}_q(msg)}$ in H_3: in this case $(\psi(pk)^\ell)^{\mathrm{H}_q(msg)}$ is public, so forging a signature for $\mathrm{H}_3(msg)$ reduces to forging one for $\mathrm{Map}_1(\mathrm{H}_p(msg))^{h_1}$.

Construction #4. The hash function $\mathrm{H}_4 : \{0,1\}^\star \to G_2$ is given by

$$\mathrm{H}_4(msg) \triangleq \Psi\left(\mathrm{Map}_2(\mathrm{H}_{p^2}(msg))\right)$$

This is the equivalent of construction #1 for G_2, and the same caveats apply.

Construction #5. The hash function $\mathrm{H}_5 : \{0,1\}^\star \to G_2$ is given by

$$\mathrm{H}_5(msg) \triangleq \Psi\left(\mathrm{Map}_2(\mathrm{H}_{p^2}(msg \,||\, 0)) \cdot \mathrm{Map}_2(\mathrm{H}_{p^2}(msg \,||\, 1))\right)$$

This is the equivalent of construction #2 for G_2. As in that case, when using the map of Section 4, it is slightly faster to sum the result on $\mathrm{E}_2'(\mathbb{F}_{p^2})$ and then apply the isogeny.

Remark 5. It is possible to use a construction analogous to #3 for G_2. Making such a construction efficient appears to require non-black-box use of Ψ, specifically, integrating a multi-exponentiation in a way similar to the one we describe in Remark 3. Looking ahead (§6), the performance overhead of #3 is already large, and the overhead of an analogous scheme for G_2 would be even larger; we do not consider it further.

6 Implementation and evaluation

We implement and evaluate hash-and-check, plus the hashes of Section 5 using the maps of Sections 3 and 4. We compare performance for three implementation styles of varying complexity. The most complex style uses rich functionality from a full-featured multi-precision library, and is non–constant time. The other two styles are simpler: they are restricted to using only field operations, i.e., fixed-modulus arithmetic. They differ in that one is constant time and one is not. We justify our interest in simplicity below.

In sum, we find that our optimizations to the map of Section 4 yield hash functions that are at worst only ≈9% slower than the fastest alternatives, yet are considerably simpler to implement. Moreover, when comparing implementations restricted to field operations and constant-time execution, the map of Section 4 is faster by ≈1.3–2× than the map of Section 3. Our experiments show that this speed advantage is due to our optimizations.

Implementation complexity: why restrict to field operations? We are interested in implementations restricted to field operations because this is the bare minimum functionality required for elliptic curve operations—so these primitives are guaranteed to be available and likely to be highly optimized. This is especially germane in hardware implementations and embedded cryptographic co-processors (e.g., [Gui10, CDF+11, SLA]), which are usually restricted to field arithmetic because general multi-precision arithmetic is too expensive in terms of area or energy. Such restrictions are also typical of small software libraries (e.g., [NaC, BLSa]); reasons include ease of optimization and constant-time implementation, and a simpler codebase, which generally leads to easier maintenance and fewer bugs.

Special-purpose arithmetic libraries like GMP [GMP] and large cryptographic libraries like Botan [Bota], Crypto++ [Cry], and OpenSSL [Opea] *do* implement full-featured multi-precision arithmetic—but at best only the very basic operations are constant time [Opeb, Botb]. This means that implementations aiming for input-independent runtime that use rich functionality from these libraries must resort to techniques like blinding [Koc96, FT12, Bos14]. But this is no silver bullet: such techniques increase complexity, require a good source of randomness, and must be carefully analyzed to ensure that all leaks are plugged.

Still, it is reasonable to wonder about the performance of implementations with input-independent runtime built on full-featured multi-precision libraries. Since the main cost of blinding is in complexity rather than execution time, a good first-order estimate is that such implementations would be similar in speed to their unblinded counterparts. A performance comparison with a blinded variant of the map of Section 3 is future work.

As a rough comparison of complexity, our routines for constant-time arithmetic in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ (briefly described below) comprise 342 lines of C code plus 614 lines of headers, constants, and automatically-generated addition chains. In contrast, the bare-bones mini-GMP library [GMP] (a subset of GMP's integer arithmetic that does not include the

Legendre symbol function) comprises about 3600 lines of C.[5] The multi-precision arithmetic implementations of Botan, Crypto++, and OpenSSL weigh in at about 10800 lines of C++, 5500 lines of C++, and 11700 lines of C, respectively (not including necessary support code, e.g., for memory management). Obviously, these libraries provide *much* richer functionality than ours! But our interest is in hash functions that are fast *without* rich functionality, so that they can be implemented in simple, compact code or hardware.

Implementation. We implement four hash-and-check variants and 23 variants of the hash functions of Section 5 (detailed below) in 3660 lines of C, including an implementation of constant-time field and curve operations for BLS12-381 (§2.1). Our field operations use reduced-radix Montgomery arithmetic geared to 64-bit processors. Our curve operations use the "slothful reduction" approach due to Scott [Sco17]. For exponentiations by fixed exponents (e.g., h_1, square roots, etc.) we automatically generate addition chains by trying all of the methods of Bos and Coster [BC90], Bergeron et al., [BBBD89, BBB94], and Yacobi [Yac91, Yac98], and selecting the best result.[6] For hashing and PRNG we use the OpenSSL version 1.1.1.b implementations of SHA-256 and AES-CTR [Opea]. We use GMP version 6.1.2 [GMP] as our full-featured multi-precision library.

We have released our implementation under an open-source license [BH].

Benchmarks and baselines. For each of the hash functions of Section 5 plus the hash-and-check method, we evaluate up to three variants: The first uses GMP for modular arithmetic, Legendre symbols, and field inversions. The second also uses GMP, but is restricted to using only field operations. The third uses our constant-time field arithmetic library, and executes in time independent of the message being hashed. We discuss implementation specifics immediately below.

- For hash-and-check using full GMP functionality, we use `mpz_legendre` for $\chi(\cdot)$; in microbenchmarks, this is orders of magnitude faster than an exponentiation in $\mathbb{F}_p$.

- For the Shallue–van de Woestijne maps (§3) using full GMP functionality, we implement as suggested in Section 3 (i.e., using Montgomery's trick). We use `mpz_legendre` for $\chi(\cdot)$, and compute inversions with the `mpz_invert` function; in our microbenchmarks, this costs an order of magnitude less than an exponentiation in $\mathbb{F}_p$. Since it is not constant time, this implementation computes each x_j only if necessary.

- For the hash-and-check and Shallue–van de Woestijne implementations that use only field operations, we avoid inversions and Legendre symbol computations using the trick of Bernstein et al. [BDL+12] described in Section 4. For example, to check some x_j we compute the numerator U_j and denominator V_j of $f(x_j)$, compute $U_j V_j \left(U_j V_j^3\right)^{(p-3)/4}$, and check whether we have found a square root. The constant-time Shallue–van de Woestijne map thus requires three exponentiations.

- Our SWU map (§4) needs neither inversions nor Legendre symbol computations, so there is no implementation that uses full GMP functionality. Even the constant-time versions require only one exponentiation because of our optimizations (§4).

- All hash functions return points in Jacobian projective coordinates (§2), meaning they do not need to compute an inversion after clearing the cofactor. Our implementation of Ψ (§5) also works in projective coordinates to avoid inversions. This is reasonable

[5] The full GMP is faster because it includes optimized assembly, but it is also almost two orders of magnitude more code and includes unneeded functionality, e.g., rational and floating-point support.

[6] Addition-subtraction chains (e.g. [TS13]) may improve performance; exploring this is future work.

Group	Hash function	Map	Full MP lib	Field ops only	Constant time
G_1	Hash-and-check	—	140	152	—
	(worst 10%)	—	141	242	—
	Construction #1	§3	137	148	203
		§4	—	142	149
	Construction #2	§3	171	193	293
		§4	—	169	179
	Construction #3	§3	243	254	—
		§4	—	248	—
G_2	Hash-and-check	—	562	773	—
	(worst 10%)	—	573	1423	—
	Construction #4	§3	566	712	979
		§4	—	581	570
	Construction #5	§3	798	1082	1583
		§4	—	792	787

Figure 1: Evaluation times in microseconds for hash-and-check and the constructions of Section 5 applied to the maps of Sections 3 and 4. For hash-and-check, "worst 10%" is the mean of the slowest 10% of runtimes for each experiment. "Full MP lib" implementations use complex functionality from GMP (§6), "Field ops only" implementations are restricted to field operations, and "Constant time" implementations are further restricted to executing in time independent of the hash input.

for BLS signatures because another exponentiation would immediately follow, and its input would be in Jacobian projective form in any case.[7]

- We do not implement constant-time hash-and-check or hash #3 (§5). Hash-and-check would either be very slow or have high failure probability. Looking ahead, even for non–constant time, construction #3 is slow, so we do not belabor the comparison.

Setup and method. We measure each hash function's execution time by evaluating it 10^6 times. We give all hash functions the same sequence of inputs, which we generate by hashing a seed. Each evaluation invokes the SHA-256 compression function once, uses the result to seed an AES-CTR PRNG, extracts one or more field elements as required by the construction, and executes the hash function on the extracted field element(s).

We report execution times on an Intel Xeon E3-1535M v6 running Arch Linux, (current as of April 11, 2019) with kernel version 5.0. We compile our benchmarks with GCC 8.2.1.

Results. Figure 1 tabulates our benchmark results.

- *Hash-and-check*, for both G_1 and G_2, is reasonably fast in the average case, even when restricted to field operations. As discussed in Section 1, however, it is relatively easy to find messages that take many iterations to hash. To illustrate this, we report the mean of the worst 10% of runtimes for each hash-and-check experiment. When

[7] In principle, exponentiating a point P is faster when $Z = 1$, because it allows using a mixed addition law [EFD], which is less expensive. In practice there is almost no difference. The reason is that fast exponentiation routines precompute small powers of P, and the resulting points have $Z \neq 1$ whether or not P does. This means that an exponentiation routine using mixed addition anyway requires one inversion (to clear denominators of the precomputed points via Montgomery's trick). As a result, P's denominator can be cleared for free. The difference in cost is only in the precomputation, which is negligible.

hash-and-check is implemented using a full-featured multi-precision library, the worst decile is within ≈1–2% of the average case, because additional iterations (i.e., Legendre symbol computations) are inexpensive.

When hash-and-check is restricted to field operations, worst-decile performance is considerably worse than the average case: ≈59% worse for G_1, ≈84% for G_2. In other words, about 10% of the time on a random message hash-and-check performs nearly as badly as the slowest of the alternatives that we consider. For adversarially-chosen messages, the performance could easily be even worse.

- *Construction #1*, when built on either of the maps of Sections 3 and 4 (hereafter "SW" and "SWU", respectively), gives roughly similar performance to average-case hash-and-check, whether implemented either using a full-featured multi-precision library or restricted to field operations. As noted in Section 1, however, hash-and-check has much worse worst-case performance (i.e., some messages take many iterations).

  For constant-time implementations of construction #1, SW is ≈36% slower than SWU; this is because the SW map's cost is dominated by three exponentiations, whereas the SWU map requires only one because of our optimizations (§4). Without our optimizations, the constant-time SWU map also requires three exponentiations (one inversion and two square roots), so we expect its performance to be almost identical to the constant-time SW map's. We confirmed that this is the case by implementing and measuring an unoptimized constant-time SWU map.

  Importantly, the gap in performance between the constant-time SWU-based hash and the fastest non–constant-time SW-based hash is only ≈9%. In other words, even if blinding had no cost, a constant-time SW-based hash built on a full-featured multi-precision library would be only slightly faster than our SWU-based hash, even though the SW-based hash would entail *significantly* more implementation complexity.

- *Construction #2* is slower than construction #1 by ≈20–44%, depending on the map and implementation style. Ignoring the constant-time SW map, which is an outlier, the range is ≈20–30%; roughly speaking, this is the cost of making the hash function indifferentiable from a random oracle without the downsides of hash-and-check.

  The SW map is about the same speed as the SWU map when the SW map's implementation uses `mpz_invert` and `mpz_legendre`. When restricted to field operations, however, the SW map is ≈14% slower than the SWU map. The constant-time implementation of the SW map is dramatically worse: ≈64% slower than the constant-time SWU map. (The gap is about twice as big as in construction #1 because this construction entails two map evaluations rather than one.)

  Similarly to construction #1, the constant-time SWU-based hash is only ≈5% slower than the fastest non–constant-time SW-based hash.

- *Construction #3* is the slowest hash for G_1, ≈32–47% slower than construction #2. Execution time is dominated by a multi-exponentiation (§5), so the differences between the SW and SWU map and between full-featured and restricted multi-precision operations are rather small. As mentioned above, because of the poor performance, we do not evaluate constant-time implementations, but we note that a constant-time multi-exponentiation would be even slower.

- *Construction #4*, like #1, is roughly competitive with hash-and-check in G_2 when both are implemented with the full-featured multi-precision library. And like construction #1, #4 is easily distinguished from a random oracle (§5).

  For implementations using only field arithmetic, however, the much higher cost of square root computations (about 4×: compared to $\mathbb{F}_p$, a square-root computation in $\mathbb{F}_{p^2}$ is an exponentiation of twice the length, where each step is about twice as expensive) means that SW and hash-and-check are ≈23–33% slower than SWU. This difference is even more pronounced in the constant-time case: SW is ≈72% slower.

- *Construction #5* continues the trend: the constant-time SWU map is about as fast as the fastest SW map, ≈37% faster than the SW map restricted to field operations, and ≈2× faster than the constant-time SW map.

Discussion. In sum, our optimizations yield a constant-time SWU map that is at worst ≈9% slower than the *fastest* SW map implementation, even though our optimized SWU map's implementation is very simple—it uses only field operations—while the SW map's implementation requires a much richer multi-precision arithmetic implementation.

We also find that our optimizations are effective: in their absence, a constant-time SWU implementation using only field operations requires three exponentiations (an inversion and two square roots). Our experiments show that this puts its cost on par with the cost of our constant-time SW map implementation, whose execution time is also dominated by three exponentiations. In other words, our optimizations give roughly a 1.3–2× speed-up.

Finally, we argue that a few percent performance overhead is a worthwhile trade for the simplicity of our optimized SWU map. In fact, even implementations that already build or link against a full-featured multi-precision integer library might prefer the SWU map to a blinded SW map: as previously discussed, blinded implementations require a good source of randomness, or the blinding may be rendered ineffective; and they are intrinsically more complex to maintain and debug, e.g., because blinded execution is nondeterministic.

7 Conclusion

We tackled the problem of hashing to Barreto-Lynn-Scott pairing-friendly elliptic curves [BLS03], focusing on BLS12-381 [BLSb]. To do so, we revisited the Shallue–van de Woestijne [SvdW06] and "simplified" Shallue–van de Woestijne–Ulas [Ula07, BCI+10] maps.

We proposed an "indirect" SWU map to Barreto-Lynn-Scott curves. Specifically, we showed a simple way of extending the SWU map to curves with j-invariant $\in \{0, 1728\}$, by mapping to an isogenous curve and then evaluating the isogeny map. This is important in our context because Barreto-Lynn-Scott curves always have zero j-invariant. We also proposed a small change that extends the SWU map to curves over fields where $\#\mathbb{F} \not\equiv 3 \bmod 4$.

We then described several optimizations that make the SWU map simpler to implement and faster to evaluate, including in constant time. Specifically, our optimizations eliminate field inversions and quadratic character computations in the SWU map, and make it possible to evaluate the map in constant time by computing only one modular square root.

Finally, we implemented and evaluated over two dozen different hash function variants built on the Shallue–van de Woestijne and optimized SWU maps. All told, we found that our optimizations to the SWU map yield hash functions that are fast, simple to implement, and constant time. Specifically, constant-time hash functions based on this map implemented using only field arithmetic are within 9% of the best-performing non–constant-time hash functions, which require significantly more complex implementations.

Acknowledgments

This work was supported in part by the NSF, the ONR, the Simons Foundation, the Stanford Center for Blockchain Research, and the Ripple Foundation.

References

[AR12] Gora Adj and Francisco Rodríguez-Henríquez. Square root computation over even extension fields. Cryptology ePrint Archive, Report 2012/685, 2012. http://eprint.iacr.org/2012/685.

[BBB94] F. Bergeron, J. Berstel, and S. Brlek. Efficient computation of addition chains. *Journal de Théorie des Nombres de Bordeaux*, 6(1):21–38, 1994.

[BBBD89] F. Bergeron, J. Berstel, S. Brlek, and C. Duboc. Addition chains using continued fractions. *J. Algorithms*, 10(3):403–412, September 1989.

[BC90] Jurjen N. Bos and Matthijs J. Coster. Addition chain heuristics. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 400–407. Springer, Heidelberg, August 1990.

[BCI+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 2010*, volume 6223 of *LNCS*, pages 237–254. Springer, Heidelberg, August 2010.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 435–464. Springer, Heidelberg, December 2018.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. Technical report, January 2008.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 213–229. Springer, Heidelberg, August 2001.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 416–432. Springer, Heidelberg, May 2003.

[BH] bls12-381_hash: Fast and constant-time hashing to the BLS12-381 elliptic curve. https://github.com/kwantam/bls12-381_hash.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013.

[BLSa] GitHub: zkcrypto/pairing BLS12-381 implementation. https://github.com/zkcrypto/pairing/tree/master/src/bls12_381.

[BLSb] BLS12-381: New zk-SNARK elliptic curve construction. https://z.cash/blog/new-snark-curve/.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, December 2001.

[BLS03] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 257–267. Springer, Heidelberg, September 2003.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 319–331. Springer, Heidelberg, August 2006.

[Bos14] Joppe W. Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, 4(4):275–281, November 2014.

[Bota] Botan: Crypto and TLS for modern C++. https://botan.randombit.net.

[Botb] Botan: Side channels. https://botan.randombit.net/manual/side_channels.html.

[BP17] Alessandro Budroni and Federico Pintore. Efficient hash maps to $\mathbb{G}_2$ on BLS curves. Cryptology ePrint Archive, Report 2017/419, 2017. http://eprint.iacr.org/2017/419.

[CDF+11] Ray C. C. Cheung, Sylvain Duquesne, Junfeng Fan, Nicolas Guillermin, Ingrid Verbauwhede, and Gavin Xiaoxu Yao. FPGA implementation of pairings using residue number system and lazy reduction. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 421–441. Springer, Heidelberg, September / October 2011.

[CK11] Jean-Marc Couveignes and Jean-Gabriel Kammerer. The geometry of flex tangents to a cubic curve and its parameterizations. Cryptology ePrint Archive, Report 2011/033, 2011. http://eprint.iacr.org/2011/033.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993.

[Cry] Crypto++ Library. https://www.cryptopp.com/.

[EFD] Explicit formulas database: Genus-1 large-characteristic short weierstrass curves. http://www.hyperelliptic.org/EFD/g1p/auto-shortw.html.

[Far11] Reza Rezaeian Farashahi. Hashing into hessian curves. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT 11*, volume 6737 of *LNCS*, pages 278–289. Springer, Heidelberg, July 2011.

[FFS+13] Reza R. Farashahi, Pierre-Alain Fouque, Igor E. Shparlinski, Mehdi Tibouchi, and J. Felipe Voloch. Indifferentiable deterministic hashing to elliptic and hyperelliptic curves. *AMS Mathematics of Computation*, 82(281):491–512, 2013.

[FJT13] Pierre-Alain Fouque, Antoine Joux, and Mehdi Tibouchi. Injective encodings to elliptic curves. In Colin Boyd and Leonie Simpson, editors, *ACISP 13*, volume 7959 of *LNCS*, pages 203–218. Springer, Heidelberg, July 2013.

[FKR12] Laura Fuentes-Castañeda, Edward Knapp, and Francisco Rodríguez-Henríquez. Faster hashing to $\mathbb{G}_2$. In Ali Miri and Serge Vaudenay, editors, *SAC 2011*, volume 7118 of *LNCS*, pages 412–430. Springer, Heidelberg, August 2012.

[FSV09] Reza R. Farashahi, Igor E. Shparlinski, and Jos'e Felipe Voloch. On hashing into elliptic curves. *Journal of Mathematical Cryptology*, 3(4):353–360, 2009.

[FT10] Pierre-Alain Fouque and Mehdi Tibouchi. Estimating the size of the image of deterministic hash functions to elliptic curves. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 81–91. Springer, Heidelberg, August 2010.

[FT12] Pierre-Alain Fouque and Mehdi Tibouchi. Indifferentiable hashing to Barreto-Naehrig curves. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 1–17. Springer, Heidelberg, October 2012.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[GLV01] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 190–200. Springer, Heidelberg, August 2001.

[GMP] The GNU Multi-Precision arithmetic library. https://gmplib.org/.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[GS08] Steven D. Galbraith and Michael Scott. Exponentiation in pairing-friendly groups using homomorphisms. In Steven D. Galbraith and Kenneth G. Paterson, editors, *PAIRING 2008*, volume 5209 of *LNCS*, pages 211–224. Springer, Heidelberg, September 2008.

[Gui10] Nicolas Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 48–64. Springer, Heidelberg, August 2010.

[Has] Helmut Hasse. Zur Theorie der abstrakten elliptischen Funktionenkörper, I–III. *Journal für die reine und angewandte Mathematik*, 1936(175).

[HSV06] F. Hess, N.P. Smart, and F. Vercauteren. The eta pairing revisited. Cryptology ePrint Archive, Report 2006/110, 2006. http://eprint.iacr.org/2006/110.

[Ica09] Thomas Icart. How to hash into elliptic curves. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 303–316. Springer, Heidelberg, August 2009.

[IMCT02] Tsutomu Iijima, Kazuto Matsuo, Jinhui Chao, and Shigeo Tsujii. Construction of Frobenius maps of twists [of] elliptic curves and its application to elliptic scalar multiplication. In *Proc. SCIS*, January 2002.

[KB16] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 543–571. Springer, Heidelberg, August 2016.

[KLR10] Jean-Gabriel Kammerer, Reynald Lercier, and Guénaël Renault. Encoding points on hyperelliptic curves over finite fields in deterministic polynomial time. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *PAIRING 2010*, volume 6487 of *LNCS*, pages 278–297. Springer, Heidelberg, December 2010.

[Knu97] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*, chapter 4.6.4. Addison-Wesley, 3$^{\text{rd}}$ edition, 1997.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, August 1996.

[KT15] Taechan Kim and Mehdi Tibouchi. Improved elliptic curve hashing and point representation. In *Proc. Workshop on Coding and Cryptography*, April 2015.

[Möl01] Bodo Möller. Algorithms for multi-exponentiation. In Serge Vaudenay and Amr M. Youssef, editors, *SAC 2001*, volume 2259 of *LNCS*, pages 165–180. Springer, Heidelberg, August 2001.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, Heidelberg, February 2004.

[NaC] NaCL. https://nacl.cr.yp.to.

[Opea] OpenSSL: Cryptography and SSL/TLS toolkit. https://www.openssl.org/.

[Opeb] BIGNUM code is not constant-time due to bn_correct_top. https://github.com/openssl/openssl/issues/6640.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[SBC$^+$09] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. Fast hashing to G_2 on pairing-friendly curves. In Hovav Shacham and Brent Waters, editors, *PAIRING 2009*, volume 5671 of *LNCS*, pages 102–113. Springer, Heidelberg, August 2009.

[Sco17] Michael Scott. Slothful reduction. Cryptology ePrint Archive, Report 2017/437, 2017. http://eprint.iacr.org/2017/437.

[Ska05] Mariusz Skałba. Points on elliptic curves over finite fields. *Acta Arithmetica*, 117(3):293–301, 2005.

[SLA] Silicon Labs EFM32PG12 Pearl Gecko Microcontroller Reference Manual. https://www.silabs.com/documents/public/reference-manuals/efm32pg12-rm.pdf.

[SM] SageMath. http://www.sagemath.org/.

[SS04] Andrzej Schinzel and Mariusz Skałba. On equations $y^2 = x^n + k$ in a finite field. *Bulletin of the Polish Academy of Sciences Mathematics*, 52(3):223–226, 2004.

[SvdW06] Andrew Shallue and Christiaan E. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *Algorithmic Number Theory Symposium*, July 2006.

[TS13] Amadou Tall and Ali Yassin Sanghare. Efficient computation of addition-subtraction chains using generalized continued fractions. Cryptology ePrint Archive, Report 2013/466, 2013. http://eprint.iacr.org/2013/466.

[Ula07] Maciej Ulas. Rational points on certain hyperelliptic curves over finite fields. *Bulletin of the Polish Academy of Sciences Mathematics*, 55(2):97–104, 2007.

[Yac91] Yacov Yacobi. Exponentiating faster with addition chains. In Ivan Damgård, editor, *EUROCRYPT'90*, volume 473 of *LNCS*, pages 222–229. Springer, Heidelberg, May 1991.

[Yac98] Yacov Yacobi. Fast exponentiation using data compression. *SIAM J. Comput.*, 28(2):700–703, 1998.

A The isogeny maps

The following Sage script computes and outputs the rational maps $E_1'(\mathbb{F}_p) \mapsto E_1(\mathbb{F}_p)$ and $E_2'(\mathbb{F}_{p^2}) \mapsto E_2(\mathbb{F}_{p^2})$.

```
#!/usr/bin/env sage
# vim: syntax=python
#
# This script prints out the rational maps for the isogenies used in the indirect SWU map for BLS12-381

p = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab
F = GF(p)
F2.<X> = GF(p^2, modulus=[1,0,1])

##
## Ell1' -> Ell1
##
a1 = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd881ac98936f8da0e0f97f5cf428082d584c1d
b1 = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fcef35ef55a23215a316ceaa5d1cc48e98e172be0
k1 = [ 0x133341fb0962a34cb0504a9c4fada0a5090d38679b4c040d5d1c3afb023a3409fcc0815fea66d8b02bbef9c8b5a66e07
     , 0x264908af037bcede00d054cf5d4775e83eb6cf63c76b969f8ed174fb59fcff78d201f46f6cfc4ed6552e59ce75177b0
     , 0x1335c502c1f54c49aceea65e87fd7203ba0f626f305fc0cfd606a5dae9f3c8e81a4b3b69600129fabd307c69bf319d39
     , 0x94440f65f408a6e930e16e3e92dd17bf60d6e9679a8d3d58593de55ac23703042d609537eb3549aac234d896ca82944
     , 0x4afe09d5cf4956a23b6b71f59d2b3407b415a774b7be81bbb6fa99cbc798e0ac98ba725a5bc328016b1c268b4766e85
     , 0x1
     ]
Ell = EllipticCurve(F, [0, 4])
EllP = EllipticCurve(F, [a1, b1])
iso11 = EllipticCurveIsogeny(EllP, k1, codomain=Ell, degree=11)
iso11.switch_sign()  # we use the isogeny with opposite sign for y; this was an arbitrary choice

##
## Ell2' -> Ell2
##
Ell2 = EllipticCurve(F2, [0, 4 * (X + 1)])
Ell2p = EllipticCurve(F2, [240 * X, 1012 * (X + 1)])
iso3 = EllipticCurveIsogeny(Ell2p, [6 * (1 - X), 1], codomain=Ell2)

# print rational maps
print iso11.rational_maps()
print iso3.rational_maps()

## you can also access the rational maps separately
# xmap = iso.rational_maps[0]
# ymap = iso.rational_maps[1]

## and also their numerators and denominators
# xmap_num = xmap.numerator()
# xmap_den = xmap.denominator()
# ymap_num = ymap.numerator()
# ymap_den = ymap.denominator()
```