EC-OPRF: Oblivious Pseudorandom Functions using Elliptic Curves

Jonathan Burns*, Daniel Moore, Katrina Ray, Ryan Speers and Brian Vohaska*

Ionic Research
math@ionicsecurity.com

Abstract

We introduce a secure elliptic curve oblivious pseudorandom function (EC-OPRF) which operates by hashing strings onto an elliptic curve to provide a simple and efficient mechanism for computing an oblivious pseudorandom function (OPRF). The EC-OPRF protocol enables a semi-trusted server to receive a set of cryptographically masked elliptic curve points from a client, secure those points with a private key, and return the resulting set to the client for unmasking. We also introduce extensions and generalizations to this scheme, including a novel mechanism that provides forward secrecy, and discuss the security and computation complexity for each variant. Benchmark tests for the implementations of the EC-OPRF protocol and one of its variants are provided, along with test vectors for the original protocol.

1 Introduction

In this paper, we propose a construction for a cryptographically secure oblivious pseudorandom function (OPRF) based on elliptic curve cryptography. As with other OPRF constructions, selected information is concealed from each of the two parties involved in the PRF. Colloquially speaking, a user Alice transfers an encoded input to a semi-trusted party Ted – who performs a calculation and sends the result to Alice – which Alice then uses to finalize the PRF evaluation. Throughout this process, two things hold: (a) Ted can neither determine Alice's original input nor compute her final output, and (b) Alice does not gain sufficient information to compute the PRF independently. More formally, we rely on the definition of an OPRF from [11], that it is a secure two-party protocol which for some pseudorandom function family f_r has the functionality $g(r, w) = (\lambda, f_r(w))$. The client holds the input w, the server holds a key r, the client outputs $f_r(w)$, and the server outputs nothing (designated by λ).

Such OPRF constructions can be utilized for a variety of applications including dynamic hashing, constructing deterministic yet memory-less authentication schemes, message-locked encryption key generation to enable de-duplication of encrypted files, and others as identified in [13], [11], and [2]. Other constructions proposed have been based on the Decisional Diffie-Hellman assumption (DDH) [11] and via Oblivious Polynomial Evaluation [18]. Similar techniques could be based on Chaum's blind signature scheme [8] similar to what we discuss in Section 5.2. We propose a method based on the security of elliptical curve discrete log problem.

In Section 2, we present the conventions used throughout the paper. The algorithm for the EC-OPRF protocol and its potential extensions are introduced in Section 3. Section 4 provides a detailed example of the EC-OPRF protocol, and includes test vectors for each step. Section 5 demonstrates how the EC-OPRF protocol can be generalized to arbitrary commutative groups – e.g., cylic groups – and offers a construction based on the RSA protocol, and Section 5.3 compares the performance metrics from the C++ implementations of the EC-ORPF protocol with one of its variants.

*These authors contributed equally.

2 Preliminaries

In this section, we introduce definitions which will be used throughout this paper along with suggestions for choosing secure functions and parameters.

2.1 Definitions and Notations

Parties

- Let *Alice* be a client that initiates the OPRF protocol.
- Let *Ted* be a semi-trusted third party that salts queries from Alice.
- Let *Claude* be a cloud-based service, which may be either a passive or active participant in the protocol.

Elliptic curves

- For a prime p, let $E(a, b, p)$ be an elliptic curve over GF(p) with order r defined by
$$y^2 = x^3 + ax + b$$
for integers a, b such that $4a^3 + 27b \neq 0$.[1]
- Let $c * X = \underbrace{X + X + \cdots + X}_{c\ times}$ denote the multiplication of a scalar integer c with a point X on an elliptic curve.

Hash and Pseudorandom functions

- Let $H : \{0, 1\}^* \to E$ be a secure hash function, i.e., the digest $H(w)$ of the binary string w is a point on the elliptic curve E.[2]
- Let $P : E \to \{0, 1\}^*$ be a pseudorandom function, i.e., $P(X)$ is the random binary string derived from the elliptic curve point $X \in E$. This is used in the **Random** step in Section 3 to convert an elliptic curve point to a fixed size block.

Client parameters

- Let $W = \{w_0, ..., w_{l-1}\}$ be the set of binary strings in a collection.
- Let $M = \{m_0, \ldots, m_{l-1}\}$ be a set of random integer scalars, called *masks*, corresponding to the input strings $\{w_0, \ldots, w_{l-1}\}$, which have multiplicative inverses $M^{-1} = \{m_0^{-1}, \ldots, m_{l-1}^{-1}\}$ modulo r, respectively.

Server parameters

- Let the scalar integer s be Ted's private key ("salt").[3]
- Let the scalar integer t be Claude's private key ("pepper").

[1] For the purpose of our implementation, we choose the curve NIST P-384 unless otherwise specified.
[2] Our implementation composes the SHA-256 hash with the 'Try-And-Increment' hash [6] (see Algorithm 1).
[3] This is not a public salt such as is used to confound table-based attacks, but instead this is a private key.

2.2 Hash function H

The EC-OPRF protocol requires a secure hash function H which consumes an input string and produces a point on the elliptic curve $E(a, b, p)$, where the mapping onto the curve provides confusion of the input string, and the function is non-invertible.

In practice, H can be constructed as the composition of a secure hash function $F_1 : W \to \mathrm{GF}(p)$ and an injection $F_2 : \mathrm{GF}(p) \to E(a, b, p)$, i.e., $H(w_i) = F_2(F_1(w_i))$ for each w_i in W. In this case, the security of the hash function H is equivalent to the security (including confusion and non-invertibility) of F_1, so a secure cryptographic hash function such as SHA-256, RIPEMD-160, or BLAKE 2 should be used. For F_2, there are several known methods for mapping an integer onto an elliptic curve [7, 14], e.g., the 'Try-and-Increment' method [6], the 'Twisted' curves method [9], the Boneh-Franklin admissible encoding for supersingular curves [5], the Shallue-Woestijne-Ulas algorithm [19], and the Brier method [7].

Of the methods mentioned above, the Boneh-Franklin admissible encoding, the 'Twisted' curves method, and the Brier method are indifferentiable from a random oracle [1, 14]. However, the Boneh-Franklin admissible encoding is a bijection over supersingular elliptic curves, which are susceptible to the MOV attack [17], and require substantially larger parameters to ensure the same level of security provided by ordinary elliptic curves. Similarly, the 'Twisted' curves method maps each input to either a point on the elliptic curve or one of its twisted curves, but this effectively doubles the computation time of the protocol since each computation must be carried on both curves [9]. In the example given in Section 4, H is defined as the composition of SHA-256 for F_1 and the 'Try-and-Increment' method for F_2 (see Algorithm 1 for details).

Algorithm 1 Secure hash function $H(w_i)$

Input: A string w_i and an elliptic curve $E(a, b, p)$
Output: A point G_i on $E(a, b, p)$

1: $x_i \leftarrow \text{SHA-256}(w_i) \pmod p$
2: **while** $x_i^3 + ax_i + b$ is not a quadratic residue modulo p **do**
3: $\quad x_i \leftarrow x_i + 1$
4: **end while**
5: $y_i \leftarrow (x_i^3 + ax_i + b)^{1/2}$
6: **return** $G_i = (x_i, y_i)$

Note that in Algorithm 1, if you exceed a security parameter (e.g., some number of iterations in the while loop) and the point is not a quadratic residue, then you re-start the algorithm with a different w_i (e.g., by incrementing the input value by one).

2.3 Elliptic curve E

The choice of the elliptic curve $E(a, b, p)$ used throughout this paper affects the security, speed, and utility of the overall protocol. For instance, the order r of $E(a, b, p)$ bounds the number of possible outputs. Choosing a curve where r is small increases the probability of collisions among the outputs, and weakens the security of the protocols that are dependent on the difficulty of the elliptic curve discrete logarithm problem.

For general purposes, we recommend that a secure and peer-reviewed elliptic curve should be chosen for use in this protocol. Examples include NIST curves as defined in [15] (e.g., NIST P-384) and [4] (e.g., Curve41417). When speed and memory are critical, several specialized elliptic curve families may also be appropriate (see [3, 12]), e.g., Edwards curves, Inverted Edwards curves, or Montgomery curves.

The parameters a, b, p, and r for the NIST P-384 curve are given in Table 1, and the P-384 curve is used throughout the example presented in Section 4.

3 Algorithm Description

This section defines the EC-OPRF protocol and offers optional extensions to it.

3.1 EC-OPRF

An oblivious pseudorandom function, such as the one described here, can be thought of as a keyed hash – with the properties that the first party holding the input to hash does not disclose the input to the second party, and the second party retains the key for the hash and does not disclose it to the first party.

1. **(Hash)** The client (i.e., Alice) first takes the collection of inputs $W = \{w_0, \ldots, w_{l-1}\}$ and computes the hash H of each (see Algorithm 1), namely

$$G_i = H(w_i), \tag{1}$$

   to form the set $G = (G_0, \ldots, G_{l-1})$.

2. **(Mask)** For each G_i in G, the client then generates a random integer m_i as a mask for G_i, and computes the scalar multiplication

$$M_i = m_i * G_i, \tag{2}$$

   and sends the list $M = (M_0, \ldots, M_{l-1})$ to the server (i.e., Ted).[4]

3. **(Salt)** The server then salts each M_i in M with its private key s as

$$S_i = s * M_i, \tag{3}$$

   and returns the list $S = (S_0, \ldots, S_{l-1})$ back to the client.

4. **(Unmask)** Upon receipt of S, the client applies each inverse mask $m_i^{-1} \pmod r$ to S_i and obtains

$$U_i = m_i^{-1} * S_i = m_i^{-1} * (s * M_i) = m_i^{-1} * (s * (m_i * G_i)) = s * G_i, \tag{4}$$

   the server's salted version of the hashed point G_i (corresponding to the input w_i).

Note that for any input w the output $U_i = s * G_i$ is well defined in terms of s and G_i, and cannot be solely computed by either the client or server. While Alice has access to G_i and $s * G_i$, recovering Ted's s is the elliptic curve discrete log problem and is known to be difficult to compute [16]. The server only has access to s and $m * H(w_i)$ and thus can not compute either w_i or $H(w_i)$. Hence, the EC-OPRF protocol is indeed oblivious.

3.2 Extensions

The OPRF from the previous section results in a secure two-party computation that can be used in calculations by the original user or used as a component for an external application. Section 3.2.1 describes how the protocol can be extended to provide forward secrecy for a third party application, and Section 3.2.2 provides additional security alternatives such as using the signed hash as a seed for a pseudorandom function.

3.2.1 Key Rotation

There are often security or regulatory requirements for updating keys used in cryptographic services which can either be based on time or frequency of use. In this section, a scheme is presented which provides, via key rotation, forward secrecy while maintaining a minimal state. To illustrate this, we describe a scenario involving three parties: the client (Alice), the salting server (Ted), and a recipient "cloud" server (Claude). See Figure 1 for a schematic outline of the three party system. As in Section 3.1 above, Alice chooses an input w_i and coordinates with Ted to compute $U_i = s * G_i$. Alice will use the result of the protocol as input to an external application hosted by Claude, so she sends him the unmasked point U_i. Claude can either use this value directly or apply an additional signature using a private integer t:

[4]Although it is possible for the server to recompute y_i from x_i and $\mathrm{sgn}(y_i)$, sending both coordinates of M_i is more computationally efficient (at the expense of bandwidth).

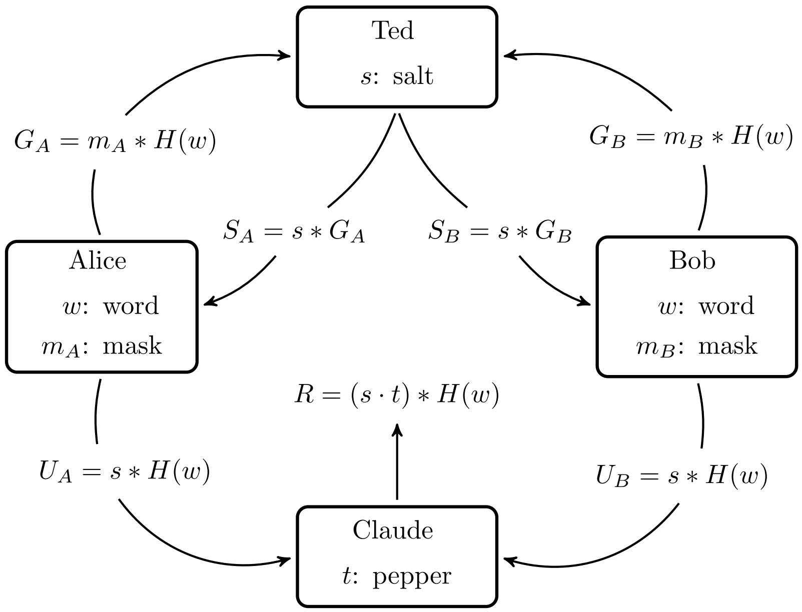

Figure 1: Example flow between the EC-OPRF parties

5. **(Pepper)** The cloud based server receives the unmasked points $U = (U_0, \ldots, U_{l-1})$ and calculates each

$$R_i = t * U_i = (t \cdot s) * G_i, \tag{5}$$

which are used as tokens for a hosted application.

If Claude actively participates in the scheme by performing the **Pepper** step, a key rotation can be initiated by either the salting server, Ted, or another trusted party. The key rotation authority can initiate an update by generating a random integer k, calculating, $k^{-1} \pmod r$, and securely distributing the values k and k^{-1} to Ted and Claude, respectively, who update their private constants accordingly:

$$\text{Ted:} \quad \hat{s} \leftarrow k \cdot s \qquad \text{and} \qquad \text{Claude:} \quad \hat{t} \leftarrow k^{-1} \cdot t\,.$$

It is straightforward to verify that the final keyed hash remains unchanged after performing the key rotation:

$$\hat{R}_i = (\hat{t} \cdot \hat{s}) * G_i = ((k^{-1} \cdot t) \cdot (k \cdot s)) * G_i = (t \cdot s) * G_i = R_i\,.$$

The keys s and t can be rolled repeatedly without storing the random constants k and k^{-1}. Ted learns nothing about Claude's computed key $\hat{t}$ when he receives k, and likewise Claude learns nothing about Ted's computed key $\hat{s}$ even though k^{-1} can be computed from k.

Messages sent across the channel between Alice and Ted are obsured by a random scalar mask, so the input w_i maps to a different elliptic curve point each time the protocol is run. However, the message becomes fixed after performing the **Unmask** step, making the channel between Alice and Claude susceptible to statistical active attacks. However, periodically changing Ted's key s to $\hat{s}$ causes the point $U_i = s * G_i$ to become $\hat{U}_i = \hat{s} * G_i$, potentially prohibiting the sample space to grow large enough to build a reliable statistical model.

3.2.2 Client Side PRF

This extension relies on another definition, a *pseudorandom function* (PRF) P.

The client-side application of the PRF to the server's salted, unmasked elliptic curve points can be added as the final step in EC-OPRF protocol. An implementation of the EC-OPRF protocol with this extension could apply HMAC-SHA-256 to the x-coordinate of the elliptic curve point. Figure 2 shows an example system which incorporates this extension.

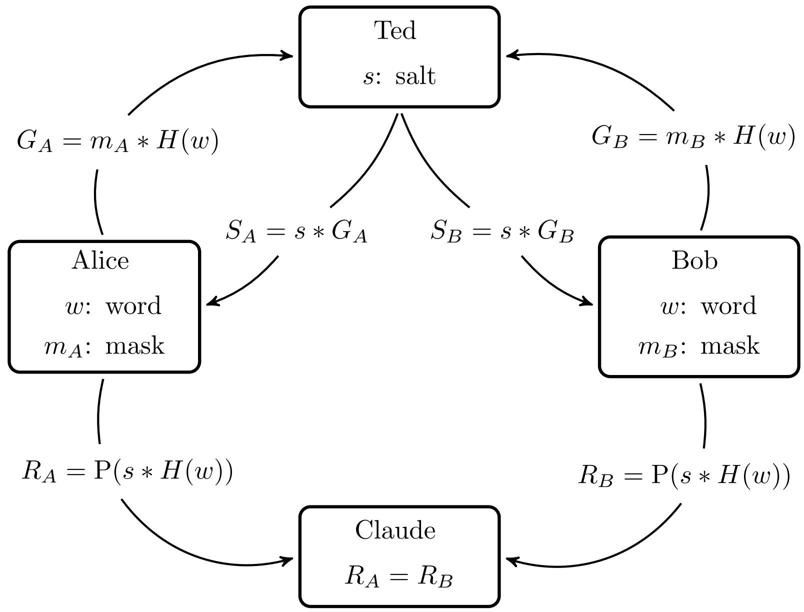

Figure 2: Example flow between components with a client-side PRF

To implement this, an additional step may be added after the **Unmask** step given in the algorithm. In this additional step, **Random**, the client applies the PRF P to each element of U, ensuring that a trusted one-way function has been utilized. The final output of the EC-OPRF is $R = (R_0, \ldots, R_{l-1})$ where:

$$R_i = P(U_i)$$

This extension is incompatible with the key rotation extension described above, as in this extension the client converts the point to a block, meaning that Claude can not preform the operations required for the key rotation extension.

4 Example

Alice the client wishes to randomize the contents of his collection of words using Ted the server's private key s to salt the randomization. The contents of Alice's collection should be kept secret from Ted, while keeping Ted's key s hidden from Alice. This can be accomplished using the EC-OPRF protocol.

Client hashing and masking

Hash. Before the protocol begins, Ted and Alice agree to base their computations on the NIST approved P-384 elliptic curve $E(a, b, p)$ with the parameters found in Table 1.

Name	Description	Hex Value
a	Coefficient	`ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff` `ffffffff fffffffe ffffffff 00000000 00000000 fffffffc`
b	Coefficient	`b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112` `0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef`
p	Prime	`ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff` `ffffffff fffffffe ffffffff 00000000 00000000 ffffffff`
r	Order	`ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff` `c7634d81 f4372ddf 581a0db2 48b0a77a ecec196a ccc52973`

Table 1: Parameters of the NIST P-384 elliptic curve $y^2 = x^3 + ax + b$ over GF(p).[15]

If Alice wants to randomize the collection

$$W = (\texttt{"see"}, \texttt{"spot"}, \texttt{"run"}).$$

she will first hash the inputs with H and mask them with M. To compute H, Alice starts by hashing the inputs W with F_1, which in this case is SHA-256 reduced modulo p. The output of $F_1(w_i)$ is shown in Table 2.

Input	$F_1(w_i) = \text{SHA-256}(w_i) \pmod p$
see $\rightarrow w_0$	00000000 00000000 00000000 00000000 aa9e9b5c 907d50fe b410f2a8 4e81ab72 c5ae6724 d57ccc53 650f9361 d33dc734
spot $\rightarrow w_1$	00000000 00000000 00000000 00000000 be2bdbb3 100e4119 1874b192 057ae741 8f549e9d 4ec8b3eb 1a3ada7b 9453d365
run $\rightarrow w_2$	00000000 00000000 00000000 00000000 acba2551 2100f80b 56fc3ccd 14c65be5 5d94800c da77585c 5f41a887 e398f9be

Table 2: Computation of F_1 for the inputs W.

Now that each input of W has been assigned to an element of GF(p), Alice can finish the computation of $H(w_i)$ using the 'Try-and-Increment' method for F_2, which associates each element with a point on the elliptic curve $E(a, b, p)$. The output of composing F_2 with F_1 is $G_i = H(w_i)$, and is shown in Table 3. Note that for each w_i, the x coordinate of G_i is almost identical to the values of $F_1(w_i)$, which indicates that Algorithm 1 terminates after only a few 'tries', indicating that a high number of the elements in GF(p) are quadratic residues of the chosen curve.

EC Point		$G_i = H(w_i) = F_2(F_1(w_i))$
G_0	x	00000000 00000000 00000000 00000000 aa9e9b5c 907d50fe b410f2a8 4e81ab72 c5ae6724 d57ccc53 650f9361 d33dc736
	y	cf810efb 16aa6de4 7e8a4532 c23f3b2b 4961fca2 943f3f41 15ab492c c3ae278e 4d8626fb 11c8078c c859291f e45b708d
G_1	x	00000000 00000000 00000000 00000000 be2bdbb3 100e4119 1874b192 057ae741 8f549e9d 4ec8b3eb 1a3ada7b 9453d36a
	y	04a82d48 a75fa977 9a69b6bb 09558892 e448ef12 a4763d73 1173de19 3392b3f9 a2d4a9a3 eb76b6f5 47fc25e1 1369dc58
G_2	x	00000000 00000000 00000000 00000000 acba2551 2100f80b 56fc3ccd 14c65be5 5d94800c da77585c 5f41a887 e398f9c3
	y	747d6333 cfd7724a 6dd810dc 7c845200 d18a8200 2aca7ae3 db9f4140 401778d1 c4a964d7 aa49d30a c527158c 67bb44a1

Table 3: Composition of F_2 with F_1 to compute H for the input W.

Mask. Since Alice wants to conceal the contents of her collection, she will mask each of her points before sending them to Ted. In particular, Alice multiplies each point G_i with a randomly generated scalar integer m_i (Table 4), and sends the resulting list M (Table 5) to Ted.

Scalar	Random Mask (m_i)
m_0	ee1c8974 956313c3 14379d6e 714ce2af 3aa3fabe 105f7da6 70d8ad2c 97148b08 b7da6e93 02d6d251 169bc8b8 472e9ad0
m_1	9b7491c6 ceb651cd a592fd50 5c10aaa0 cbe4dfc4 662fb223 67b46627 3ad1af51 886e4006 0c08c09f 7fe8ce6f 4c181fe7
m_2	e259c5b6 9e130b72 251e926e 815ba6c7 8343b3f2 cc044ec5 b7c928c6 a8d2445c d40c254c d351ebc7 ba4bcccf be79d63d

Table 4: The list m of scalar integer masks, chosen randomly by Alice for each input of W.

EC Point		$M_i = m_i * G_i$
M_0	x	`5e231c9b 0b685fbc 9cdb11b1 48009ba d08d7da88 b2a4419a`
		`472ad7ff 2bbdbd63 ad955361 814b5b1 3681d9f2f 5c4cb73c`
	y	`d91c2360 27ce556c 6dfec798 65870078 127a97f4 0ceffe4f`
		`d6ea1b39 4abbf944 d86cb381 4f173fbc 4978c36d c2b1432f`
M_1	x	`3dd6a1c7 7624897a 4376c0ae b939432a 9f64f479 b51f2c89`
		`8f0f30cc 2d2c0df8 88ca48a6 807cb66d d6a2b209 54056a54`
	y	`4a3ec4c0 35c3fb83 988b48d7 e341b8c1 02d762e7 623a2928`
		`04661903 81e19e4d c774bd04 447224db 3c9b9864 b1a06965`
M_2	x	`27bdf67b f939c04a 294be47f 00be8a5d 2ac735d5 3ceeed38`
		`0d93cd59 681371d7 177e0297 6f7b08e3 cfc502c1 5077df99`
	y	`5ddc13d9 3ed4705a 8c185c04 690e70d7 73649ce1 fb7ed356`
		`9c68d925 4023abbb dd2a0b28 2f16d479 e2d89acf c918606f`

Table 5: Alice's hashed and masked elliptic curve points M.

Server salting

Salt. Ted receives the list M, salts the points with his private key s (Table 6) by computing $S_i = s * M_i$ (Table 7), and replies to Alice with the list S.

Scalar	Random Salt (s)
s	`379c5eaf bd99f838 23fa59e6 cfe61a73 785fdcc5 7cceb654`
	`b35ed9f8 3d996f18 6a03d019 304dc3ce 9caf73c1 587b3e94`

Table 6: Ted's private key s used to salt M.

EC Point		$S_i = s * M_i$
S_0	x	`1b11424c a9777bde 4f16010d 94665c1f 154d2514 42a8d64b`
		`3c0eca92 bfe21c24 12c4ac56 330edb49 3f3bfcee ca79b9f8`
	y	`cc178f90 807f2507 165ccd2f a508c6ba dde1abbd 374c6956`
		`261c34b5 1a9de437 4d5d021c 731f15f6 dd4d3712 2c6def90`
S_1	x	`39f476df 09ca72f9 d45befa7 8eb68279 b2a06325 6e3f1569`
		`d4bd2e1b bd2add9f 399d8c16 e4d96a08 fbf07780 55b5a8de`
	y	`5f7405ff 1ff521cc 19275b95 48b8e9a9 86c0643e 93b8c6b9`
		`577edbfc 64dcb67d e755c1ab b1c87c61 47d971b1 2f59bacf`
S_2	x	`484b1519 cc83bf60 42f843d7 dc9853f8 9904eebc da3ac2a1`
		`9ec5e4f5 11e27f33 ffe8bcb6 88fe746a 5e5310e9 b8a6ef9e`
	y	`eb302d0a bbd68e07 8e0e9bc3 389d1d8b 909030e5 292def15`
		`7ce021c1 79fc86b6 d481a6b1 fb0e9972 a523f0c6 a21c0dbb`

Table 7: Resulting points S from salting points M with Ted's private key s.

Client unmasking

Unmask. Now that Ted has salted the masked points, Alice can unmask each point. For each random scalar m_i, Alice computes the inverse mask $m_i^{-1} \pmod r$ shown in Table 8, and multiplies the resulting scalar with its respective point S_i to obtain $U_i = m_i^{-1} * S_i$ shown in Table 9.

Scalar	$m_i^{-1} \pmod r$
m_0^{-1}	d0216c93 1290f984 5596a916 ca6208be 8dfbaa02 311112d4 dd4f726c 95665d07 3f11e3d7 6cd2f2a0 4a1c6584 163ebecb
m_1^{-1}	dfb76a82 4d721faa 28670a21 be9246ef 32ee3dde e4fe8ecc 4e2c4d91 110c6e15 40a2bc4d 17153221 46107a7d fb3e85ec
m_2^{-1}	fca7d3b4 cbd0a157 b761fa3f b05aa618 7bb02d4a f4d375dc c6d2b06b a42437c3 51cd7a75 9ca387f6 a81dddb9 5a2edd4a

Table 8: Multiplicative inverses m^{-1} of Alice's mask set m modulo r.

EC Point		$U_i = m_i^{-1} * S_i$
U_0	x	6723853e b2079a61 144f7ccc 96d2a29c 73db81a8 f454defb e85fee0d a92cb33a ab52ca23 51e01a9a a965793e 41c9cfe1
	y	6899ae01 8468f7dd 26f9f0cd 29545a87 a27dd036 51136609 1e7cca7c 00637994 64b7c6a5 fa4dc5c9 c82cecfc 976dbdcc
U_1	x	5403ca6a 59b8ca57 ccf2f8b9 c3eb8bb7 7443d1b3 2692a0cf 9eea1a80 0e8744bb 3cde0e3e 6ec27d0c 49bcfb77 b8a14f74
	y	ce7c86b2 e4b9d6ad 512adb4e b9fc283d 2ee0b00e 8a8247bb 61e7a77b be874990 24bb21a8 e3e1c818 4e3ad89c 114e7e34
U_2	x	a51b0851 0be9eebc c13665e9 971bad76 36a96353 41daf84b cc1a9202 27ef156b 53c51eb3 8287675f 0fbc3840 168973dd
	y	35d17c05 42238c0c 0d350598 ec09efa6 3349f4a8 21dcf98a ba875d3b fb915c6c d3c81a5e 92999f45 b831ffcd 30225c72

Table 9: Points U unmasked by the Alice's mask m^{-1}.

5 Generalizations

The EC-OPRF protocol described in Section 3 is based on elliptic curve arithmetic, but the underlying protocol naturally extends to other abelian groups. For instance, the elliptic curve group $E(a, b, p)$ over $\mathrm{GF}(p)$ can be replaced by the cyclic group $(\mathbb{Z}/n\mathbb{Z})^\times$ to give a protocol similar to the RSA algorithm. See Section 5.2 for details. It should be noted that the EC-OPRF protocol makes full use of the abelian group axioms. In particular, both commutativity and invertibility are essential to the **Unmask** step.

5.1 Masking Alternatives

The **Mask** and **Unmask** steps in the EC-OPRF involve scalar multiplication of an elliptic curve point, which can be computationally expensive. As an alternative, Alice and Ted agree upon a base point $B \in E$ as part of the initial elliptic curve parameters, and Ted uses B to compute $T = s * B$ and then publicizes B and T. The alternative algorithm proceeds as before, but with the **Mask** and **Unmask** steps defined as follows:

2. **(Mask)** For each G_i in G, Alice generates a random integer m_i as a mask for G_i, and computes the **point addition**

$$M_i = (G_i) + (m_i * B)$$

and sends the list $M = (M_0, \ldots, M_{l-1})$ to Ted.

Ted proceeds to salt each point in M as before (see Section 3.1), but the S values are computed as

$$S_i = s * M_i = s * (G_i + m_i * B) = s * G_i + (s \cdot m_i) * B = s * G_i + m_i * T.$$

While Ted is computing S, Alice can pre-compute the values $(-m_i) * T$ to accelerate the unmasking step:

4. **(Unmask)** Upon receipt of S, Alice removes the mask using the **point addition**

$$U_i = S_i + (-m_i) * T = s * G_i + (m_i - m_i) * T = s * G_i$$

to once again arrive at the secured version (U_i) of the hashed point G_i.

The final **Pepper** step, performed by Claude, uses scalar multiplication as before.

Avoiding the two scalar multiplications and the multiplicative inverse calculations in the **Unmask** step yields a near 100x speed increase (excluding pre-computation time), which can be measured as the difference between the computation time listed for the client unmask and server salt steps in Table 10.

In this alternative, Ted and Claude can still perform a key rotation, but Ted will also have to update his public point T – at which time an eavesdropper will be able to detect that a key rotation has taken place.

5.2 OPRFs based on cyclic groups

Without loss of generality, we consider arithmetic over the cyclic group $(\mathbb{Z}/n\mathbb{Z})^\times$ for some integer n, known to Alice, Ted, and Claude. Initially, Ted and Claude choose private integers s and t, respectively.

5.2.1 DH-OPRF

The security of the DH-OPRF protocol relies upon the order of $(\mathbb{Z}/n\mathbb{Z})^\times$, so we take n to be a large prime.

This construction can then follow:

1. **(Hash)** For an input w, Alice hashes w with a cryptographically secure hash function $H : \{0,1\}^* \to \mathbb{Z}_n$ according to $w \mapsto H(w) \not\equiv 0 \pmod{n}$.
2. **(Mask)** Alice chooses a random $m \in \mathbb{Z}$ relatively prime to $\varphi(n)$, and sends Ted $G \equiv H(w)^m \pmod{n}$.
3. **(Salt)** Ted uses his private s to reply to Alice with $S \equiv G^s = H(w)^{ms} \pmod{n}$.
4. **(Unmask)** Alice calculates $d \equiv m^{-1} \pmod{\varphi(n)}$ from her mask m, and sends Claude $U \equiv S^d \pmod{n}$. Note that $U = H(w)^{(m^{-1}m)s} \equiv H(w)^s \pmod{n}$.
5. **(Pepper)** Claude computes $R \equiv U^t = H(w)^{st} \pmod{n}$.

Similarly, a key rotation between Ted and Claude can be accomplished if Ted chooses a random integer k relatively prime to $\varphi(n)$ and sends Claude $k^{-1} \pmod{\varphi(n)}$. Ted and Claude update their respective private keys as

$$\hat{s} \leftarrow s \cdot k \pmod{n} \qquad \text{and} \qquad \hat{t} \leftarrow t \cdot k^{-1} \pmod{n},$$

and the final keyed hash $\hat{R} = H(w)^{(s\,k)(t\,k^{-1})} \equiv H(w)^{st} \equiv R \pmod{n}$ remains unchanged.

5.2.2 RSA-OPRF

Just as in Section 5.1, the **Mask** and **Unmask** steps can be accelerated by replacing several costly exponentiation steps with integer multiplication, at the cost of Ted publicizing an addition public key.

Before the RSA-OPRF begins, Ted generates large primes p and q, computes $n = p \cdot q$ and $\varphi(n) = (p-1)(q-1)$, chooses a key $s \in \mathbb{Z}$ relatively prime with $\varphi(n)$, and computes the value e such that $e \equiv s^{-1} \pmod{\varphi(n)}$. Ted then shares the public values n and e with Alice (and optionally n with Claude). For completeness, the analogous updates to the **Mask**, **Salt**, and **Unmask** steps within the RSA context are as described below:

2. **(Mask)** Alice chooses a random $m \in \mathbb{Z}$ that is relatively prime to n, and sends Ted

$$G \equiv m^e \cdot H(w) \pmod{n},$$

where e is Ted's public exponent.

3. **(Salt)** Ted uses his private exponent s to sign Alice's message with

$$S \equiv G^s = (m^e \cdot H(w))^s = m^{e \cdot s} \cdot H(w)^s \equiv m \cdot H(w)^s \pmod{n},$$

which follows the fact that $e \cdot s \equiv 1 \pmod{\varphi(n)}$.

4. **(Unmask)** Alice calculates $m^{-1} \pmod{n}$ from her mask m, and sends Claude

$$U \equiv m^{-1} \cdot S = (m^{-1} \cdot m) \cdot H(w)^s \equiv H(w)^s \pmod{n}.$$

Note that steps 2-4 describe Chaum's blind signature scheme [8]. After the **Unmask** step, Alice can send U to Claude, who can enable key rotation by applying the **Pepper** step and computing $R \equiv U^t = H(w)^{st} \pmod{n}$. As in Section 5.2.1, the key rotation takes place when Ted – or a trusted party with knowledge of the secret $\varphi(n)$ – generates a $k \in \mathbb{Z}$ relatively prime to $\varphi(n)$, computes $k^{-1} \pmod{\varphi(n)}$, and distributes k and k^{-1} to Ted and Claude, respectively. Once again, Ted and Claude update their respective private keys as

$$\hat{s} \leftarrow s \cdot k \qquad \text{and} \qquad \hat{t} \leftarrow t \cdot k^{-1},$$

and Ted additional updates his public exponent to $\hat{e} \leftarrow \hat{s}^{-1} \pmod{\varphi(n)}$.

5.3 Comparison of EC-OPRF and RSA-OPRF

We implemented the EC-OPRF using the alternative masking scheme introduced in Section 5.1 and compared this with the RSA-OPRF algorithm introduced in Section 5.2.2 by running the following performance comparison.

Benchmark tests were conducted using implementations written in C++, utilizing core cryptographic components from Crypto++ [10], and run on a Windows 7 virtual machine with 2 GB of RAM and 2 CPUs. Our RSA-OPRF (based on Section 5.2.2) was implemented using a 2048-bit key, and the EC-OPRF (based on Section 5.1) was implemented using the NIST curve `secp256k1`. The code was compiled using Microsoft Visual Studio in the Win32 (x86) release configuration with the `/O2` flag to optimize for speed. The results from these computational experiments are shown in Table 10.

Party	Step	EC-OPRF (sec)	RSA-OPRF (sec)
Client	Hash + Mask	0.831	0.687
Server	Salt	10.850	26.908
Client	Unmask	0.160	0.206

Table 10: Speeds for 10,000 repetitions of the EC-OPRF (alternative masking) and RSA-OPRF protocols.

We observe that this implementation of the EC-OPRF offers a substantial overall speed reduction as compared to the RSA-OPRF, and only offers a slight time increase on the client side, which is beneficial when optimizing for many clients and a few servers, as it distributes the computational load to the clients.

We also see a significant improvement in the network bandwidth usage by the EC-OPRF construction as compared to other constructions. For example, Table 11 demonstrates that EC-OPRF uses 25% of the network bandwidth compared to RSA-OPRF.

	Public Key (PEM)	Phrase (raw)	Phrase (Base64)
EC-OPRF	174 bytes	512 bytes	684 bytes
RSA-OPRF	451 bytes	2048 bytes	2731 bytes

Table 11: Network Bandwidth Comparison of EC-OPRF and RSA-OPRF

6 Summary

In this paper, we have built a cryptographically secure OPRF based on elliptic curve arithmetic inspired by an existing blind signature scheme. The EC-OPRF output is a value that is signed by one or more semi-trusted parties, and can be used as a primitive within other applications. We have provided a detailed construction for the EC-OPRF protocol, an example with test vectors, and a brief statistical analysis of the benchmark tests for an implementation written in C++. Further, an extension to the protocol was introduced that provides forward secrecy – while maintaining a minimal storage overhead. Several additional variants to EC-OPRF protocol were considered, which include a method that speeds up the client side calculations via modifications to the masking steps, and other generalizations based on commutative groups. In particular, we explored the analogous DH-OPRF and RSA-OPRF constructions. The computational complexity, precomputation requirements, and security conditions were discussed for each of the protocol variations, and we showed that the C++ implementation of the EC-OPRF is faster and more space efficient than the RSA-OPRF protocol.

Acknowledgements

The authors thank Dan Boneh for several helpful conversations.

References

[1] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella-Beguelin. Verified indifferentiable hashing into elliptic curves. In *1st International Conference on Principles of Security and Trust, POST 2012*, volume 7215, pages 209–228. Springer, January 2012.

[2] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 179–194, Berkeley, CA, USA, 2013. USENIX Association.

[3] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007: 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007. Proceedings*, pages 29–50, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] Daniel J. Bernstein and Tanja Lange. Security dangers of the nist curves, 2013.

[5] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*, pages 213–229, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[6] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[7] 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, *Advances in Cryptology – CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 237–254, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[9] Olivier Chevassut, Pierre-Alain Fouque, Pierrick Gaudry, and David Pointcheval. The twist-augmented technique for key exchange. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings*, pages 410–426. Springer Berlin Heidelberg, 2006.

[10] Wei Dai. Crypto++ library. `https://www.cryptopp.com`.

[11] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference*, pages 303–324. Springer, 2005.

[12] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*, pages 190–200, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[13] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions (extended abstract). In George Robert Blakley and David Chaum, editors, *Advances in Cryptology: Proceedings of CRYPTO 84*, pages 276–288, Berlin, Heidelberg, 1985. Springer.

[14] Thomas Icart. How to hash into elliptic curves. In *Advances in cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 303–316, 2009.

[15] Cameron F. Kerry. Fips pub 186-4 federal information processing standards publication digital signature standard (dss), 2013.

[16] Kristin E. Lauter and Katherine E. Stange. The elliptic curve discrete logarithm problem and equivalent hard problems for elliptic divisibility sequences. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography: 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers*, pages 309–327, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[17] A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, 1993.

[18] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 245–254, New York, NY, USA, 1999. ACM.

[19] Andrew Shallue and Christiaan E. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Algorithmic Number Theory: 7th International Symposium, ANTS-VII, Berlin, Germany, July 23-28, 2006. Proceedings*, pages 510–524, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.