# Verifiable Random Function in the Scalar

Robin Linus

**Abstract**

This is a scheme for a verifiable random function in the scalar. The scheme can be used to generate verifiably deterministic nonces for stateless signing in multi-signature or threshold-signature schemes. It is inspired by *Purify* as described in the MuSig-DN paper[*] which requires about 2000 multiplication gates. A naive implementation of our VRF requires less than 750 gates and it can be further optimized down to about 230 gates. Furthermore, we demonstrate how to use this VRF to construct a verifiable encryption scheme for fair data exchange.

## The Implicit Representation

Let $\mathcal{G}$ be an elliptic curve in which the discrete logarithm is hard, and let $\mathcal{H}$ be a second secure curve, defined over the scalar field of $\mathcal{G}$. For example, in the case of $\mathcal{G} = \text{secp256k1}$ we can choose the curve $\mathcal{H} = \text{secq256k1}$, which is given by the equation

$$y^2 \equiv x^3 + 7 \pmod{n}$$

where $n$ is the order of secp256k1. The order of secq256k1 is the order of the base field of secp256k1 and vice versa. We can hide points of $\mathcal{H}$ in the scalar of group elements of $\mathcal{G}$. For a given point $P = (x, y) \in \mathcal{H}$ we define the *implicit representation*[1] of $P$ to be

$$\langle P \rangle = (xG, yG)$$

### Implicit Curve Point Operations

The crux of being able to perform curve point operations in the implicit representation is to prove *multiplications* in the scalar. This is possible using sigma protocols, such as a proof of multiplication, as described below.

[*] https://eprint.iacr.org/2020/1057.pdf
[1] https://eprint.iacr.org/2023/539.pdf

# Proof of Multiplication

Given the curve points $G$, $xG$, $yG$, and $xyG$ on $\mathcal{G}$, the prover wants to convince the verifier that $x$ and $y$ have indeed been multiplied. This is a classical sigma protocol known as *the Chaum-Pedersen protocol for DH-triples*[2].

## Prove

1. Choose random $r$
2. Compute $R_1 = rG$, and $R_2 = r(yG)$
3. Compute $c = \text{Hash}(R_1 \mid R_2 \mid xG \mid yG \mid xyG)$
4. Compute $s = r + c \cdot x$

## Verify

Given $R_1$, $R_2$, and $s$:

1. Compute $c = \text{Hash}(R_1 \mid R_2 \mid xG \mid yG \mid xyG)$
2. Check $sG = R_1 + c(xG)$
3. Check $s(yG) = R_2 + c(xyG)$

# Implicit Proof of Curve Point

Given a curve point in the implicit representation $\langle P \rangle = (xG, yG)$, the prover wants to convince the verifier that the point is indeed on the curve $\mathcal{H}$.

$$y^2G = x^3G + a \cdot xG + bG$$

To verify this we require:

- One proof of multiplication to show $y \cdot y = y^2$.
- Two proofs of multiplication to compute $x \cdot x \cdot x = x^3$.

[2] `https://toc.cryptobook.us/book.pdf#page=791`

# Implicit Proof of Curve Point Addition

Given three curve points in the implicit representation:

$$\langle P\rangle = (x_1G, y_1G)$$
$$\langle Q\rangle = (x_2G, y_2G)$$
$$\langle R\rangle = (x_3G, y_3G)$$

the prover wants to convince the verifier that the equation

$$\langle P\rangle + \langle Q\rangle = \langle R\rangle$$

holds. For two distinct points, we can apply the point addition formula:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$
$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda(x_1 - x_3) - y_1$$

To prove this:

- One proof of multiplication is required for each of the three equations.
- All additions in the equations can be easily verified in the scalar field by the verifier.

To efficiently prove the inversion in the first equation, the prover simply provides the slope

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

in form of a “hint”, $\lambda G$, and then proves that $\lambda$ is indeed the correct result by proving

$$\lambda \cdot (x_2 - x_1) = (y_2 - y_1)$$

with a proof of multiplication for $\lambda \cdot (x_2 - x_1)$.
Doubling a point works analogously, but the slope for doubling becomes

$$\lambda = \frac{3x_1^2}{2y_1}$$

which requires one additional multiplication for $x_1^2$, totaling 4 proofs of multiplication.

# Implicit Proof of Scalar Multiplication

Given a scalar $s$ and a curve point in its implicit representation $\langle P \rangle$, the prover wants to convince the verifier that $\langle Q \rangle = s\langle P \rangle$. This is possible by applying the double-and-add algorithm using implicit additions and doublings. Multiplying a point by a $n$-bit scalar requires at most $n$ doublings and on average $\frac{n}{2}$ additions.

# Proof of Equal Scalar

Given two generators $H_1$ and $H_2$ of $\mathcal{H}$, the prover wants to convince the verifier that the two points $zH_1$ and $zH_2$ share the same scalar. This is also a classical sigma protocol known as *discrete log equivalency (DLEQ)*.

## Prove

1. Choose $r$ randomly
2. Compute $R_1 = rH_1$ and $R_2 = rH_2$
3. Compute $c = \text{Hash}(R_1 \mid R_2 \mid zH_1 \mid zH_2)$
4. Compute $s = r + c \cdot z$

## Verify

Given $R_1$, $R_2$, and $s$:

1. Compute $c = \text{Hash}(R_1 \mid R_2 \mid zH_1 \mid zH_2)$
2. Check $sH_1 = R_1 + c \cdot zH_1$
3. Check $sH_2 = R_2 + c \cdot zH_2$

# Implicit Proof of Equal Scalar

Given two generators $H_1$ and $H_2$ of $\mathcal{H}$, the prover wants to convince the verifier that the point $zH_1$ and the implicit point $\langle zH_2 \rangle$ share the same scalar.

## Prove

1. Choose $r$ randomly
2. Compute $R_1 = rH_1$ and $R_2 = rH_2$

3. Compute $c = \text{Hash}(R_1 \mid \langle R_2 \rangle \mid zH_1 \mid \langle zH_2 \rangle)$

4. Compute $s = r + c \cdot z$

5. Prove implicitly that $\langle sH_2 \rangle = \langle R_2 \rangle + c \cdot \langle zH_2 \rangle$

6. Prove implicitly that $\langle zH_2 \rangle$ and $\langle R_2 \rangle$ are on the curve

### Verify

Given $R_1$, $\langle R_2 \rangle$, and $s$:

1. Compute $c = \text{Hash}(R_1 \mid \langle R_2 \rangle \mid zH_1 \mid \langle zH_2 \rangle)$

2. Check $sH_1 = R_1 + c \cdot zH_1$

3. Check implicitly $\langle sH_2 \rangle = \langle R_2 \rangle + c \cdot \langle zH_2 \rangle$

4. Check implicitly that $\langle zH_2 \rangle$ and $\langle R_2 \rangle$ are on the curve

## Verifiable Random Function in the Scalar

The prover wants to convince the verifier that $xG$ is a deterministic random nonce. This is essentially a *DDH VRF*[3], but in the scalar.

### Setup

Some generator $H_1 \in \mathcal{H}$ is chosen. The prover generates a public key $zH_1$ and shares it with the verifier.

### Prove

1. For a given message $m$, the prover computes another generator by hashing onto the curve $H_2 = \text{Hash}(zH_1|m)$

2. The prover computes $zH_2$ and shares its implicit representation $\langle zH_2 \rangle$ with the verifier

3. The prover generates an implicit proof of equal scalar for $zH_1$ and $\langle zH_2 \rangle$

[3] `https://toc.cryptobook.us/book.pdf#page=860`

### Verify

Given $zH_1$, $\langle zH_2 \rangle$ and the implicit proof of equal scalar

1. Compute $H_2 = \text{Hash}(zH_1 \mid m)$
2. Verify the implicit proof of equal scalar for $zH_1$ and $\langle zH_2 \rangle$
3. The prover's random nonce is the $x$-coordinate of the point $\langle zH_2 \rangle = (xG, yG)$

## Performance

The proving costs are dominated by the implicit proof of equal scalars, which requires 4 multiplications for every doubling and 3 multiplications for every addition. Thus, using a 128-bit challenge and the conventional add-and-double algorithm, we need to prove on average $128 \cdot 4 + 64 \cdot 3 = 704$ multiplications to prove an implicit scalar multiplication. Additionally, we need 3 multiplications for the implicit addition and 3 for the implicit proof of curve point, totaling an average of 710 proofs of multiplication to generate a nonce.

## Optimizations

1. The digest of the hash function producing the challenge can be truncated to 128 bits to reduce the number of steps required for the scalar multiplication. This is because we rely only on second-preimage resistance and not on collision resistance. Maybe even a 80-bit challenge is sufficient for most use cases, which reduces the number of multiplications down to 440.
2. We can use Bulletproofs instead of sigma protocols to reduce the proof size significantly, because their size is logarithmic in the number of multiplication gates.
3. We can use a windowed non-adjacent form to speed up the 128-bit scalar multiplication, which saves about 50 multiplications.
4. A single implicit proof of curve point for $\langle zH_2 \rangle$ suffices, if we verify $\langle sH_2 \rangle - c \cdot \langle zH_2 \rangle = \langle R_2 \rangle$ instead, to get the proof for $\langle R_2 \rangle$ for free, which saves 3 multiplications.
5. We do not have to compute $yG$, which saves a multiplication.
6. Can we batch proofs of implicit scalars to efficiently generate multiple nonces?
7. To destroy any potential structure in the nonce $xG$, we could use instead $xyG$ proven by another proof of multiplication

# Further Applications: Fair Data Exchange

Another application of our VRF is a *Fair Data Exchange* protocol, in which a server sells a file to a client such that client receives the file only if they paid the server and the server is only paid if they sent the correct file. This is possible by using verifiable encryption. The payment is facilitated via a blockchain. The encryption algorithm is essentially a one-time pad.

## Setup

1. The author of the *file* chunks the file into $n$ many 255-bit chunks $c_1, \ldots c_n$
2. The author computes the *file_id* $= \text{Hash}(c_1G|\ldots|c_nG)$ and publishes it such that it is available to any client
3. The author sends the file to the server

## Encrypt

1. The server generates some random $z$ and sends $zH$ to the client
2. The server computes $n$ many random generators $H_i = \text{Hash}(zH|i)$ and computes $n$ verifiably random nonces $r_1G, \ldots, r_nG$ and sends them to the client. They also send the proofs to verify the nonces
3. The server encrypts all the chunks using $e_i = c_i + r_i$ and sends them to the client

## Verify

1. The client verifies the random nonces
2. The client computes for all encrypted chunks $e_i$ the point $e_iG$ and then uses the nonces to compute the chunk commitments $c_iG = e_iG - r_iG$
3. The client checks *file_id* $= \text{Hash}(c_1G|\ldots|c_nG)$

## Decrypt

1. The client purchases the scalar $z$ of the point $zH$ atomically in the blockchain. (E.g., via a PTLC or some smart contract)
2. The client uses $z$ to compute the nonces in plaintext $r_i = [zH_i]_x$
3. The client decrypts all chunks to obtain the file $c_i = e_i - r_i$

# *Appendix*

## Optimizing the Scalar Multiplication

We can significantly optimize the scalar multiplication by leveraging the endomorphism of secp256k1. Since $p \equiv 1 \pmod 3$ there exists a $\omega_p \in \mathbb{F}_p$ such that $\omega_p^3 \equiv 1$. Also there exists a $\omega_q \in \mathbb{F}_q$ such that $\omega_q^3 \equiv 1$. This allows us to define an endomorphism

$$\phi : E \to E \quad (x, y) \mapsto (\omega_p \cdot x, y)$$

which has the interesting property that for some point $P = (x, y)$

$$\phi(P) = \omega_q P$$

which essentially reduces the expensive scalar multiplication by $\omega_q$ to a cheap field multiplication by $\omega_p$. This is the key idea behind our optimization.

### Complex Multiplication

We claim that for every $a \in \mathbb{F}_q$ we can find some $b, c \in \mathbb{F}_q$ such that $a = b + \omega_q \cdot c$ where $b$ and $c$ are about the size of $\sqrt{q}$. We will show later that why this is true. Now we want to show succinctly for some arbitrary $a \in F_q$ and $P \in E$ that $aP = Q$. Given the corresponding $b$ and $c$ we have

$$a \cdot P = (b + \omega_q c) \cdot P = b \cdot P + c \cdot \phi(P)$$

which reduces the scalar multiplication by $a$ to two scalar multiplications of half the size. We can apply Strauss-Shamir's trick to perform these two 128-bit scalar multiplications more efficiently than a regular 256-bit scalar multiplication: Let's define $b_i$ to be the bits of $b$ and $c_i$ to be the bits of $c$. Furthermore, we precompute $\phi(P)$ and $P + \phi(P)$. We iterate over all bits of $b_i$ and $c_i$ using the following algorithm:

1. Set $A_{128} = 0$.

2. For $i$ from 128 down to 0:

    - Determine the case for $(b_i, c_i)$:

$$\begin{aligned} (0,0) &\to T_i = 0 \\ (0,1) &\to T_i = \phi(P) \\ (1,0) &\to T_i = P \\ (1,1) &\to T_i = P + \phi(P) \end{aligned}$$

- $A_i = 2 \cdot A_{i+1} + T_i$

3. Return $A_0$

On average, this requires 128 doublings and $\frac{3}{4} \cdot 128$ additions.

**Fraction Representation**

The above trick can be combined with another trick which is very similar. Given some $a \in \mathbb{F}_q$ we can find some $u, v, s, t$ such that

$$a = \frac{u + \omega_q v}{s + \omega_q t}$$

where $u, v, s$ and $t$ are only 64 bits. Now for some $P$ we want to show that $aP = Q$

$$a \cdot P = \frac{u + \omega_q v}{s + \omega_q t} \cdot P = Q$$

which we can prove by showing that

$$(u + \omega_q v) \cdot P - (s + \omega_q t) \cdot Q = 0$$

By applying the endomorphism this simplifies to

$$u \cdot P + v \cdot \phi(P) - s \cdot Q - t \cdot \phi(Q) = 0$$

Analogously to the algorithm above, we can apply Strauss-Shamir's trick to make the multi-scalar multiplication more efficient.
On average, this requires 64 doublings and $\frac{15}{16} \cdot 64$ additions for a 256-bit scalar. We can apply the same trick for a 128-bit scalar, which reduces the cost to 32 doublings and $\frac{15}{16} \cdot 32$ additions. Additionally, we need 3 additions for the precomputation. Since a doubling costs 4 field multiplications and an addition costs 3 field multiplications, this totals to $32 \cdot 4 + \frac{15}{16} \cdot 32 \cdot 3 + 3 \cdot 3 = 227$ field multiplications instead of 704 for the conventional algorithm.

First, we precompute $\phi(P), -\phi(Q), P+\phi(P) = -\phi(\phi(P)), -Q-\phi(Q) = \phi(\phi(Q)), P-Q, \phi(P)-\phi(Q) = \phi(P-Q), P-\phi(\phi(Q)), \phi(\phi(P))-Q$. This can be done using only 3 curve point additions. Then we iterate over the bits of $u, v, s, t$.

1. Set $A_{64} = 0$.
2. For $i$ from 64 down to 0:
   - Determine the case for $(u_i, v_i, s_i, t_i)$:

$$
\begin{aligned}
&(0,0,0,0) \to T_i = 0 \\
&(0,0,1,0) \to T_i = -Q \\
&(0,0,0,1) \to T_i = -\phi(Q) \\
&(0,0,1,1) \to T_i = -Q-\phi(Q) \\
&(1,0,0,0) \to T_i = P \\
&(1,0,1,0) \to T_i = P-Q \\
&(1,0,0,1) \to T_i = P-\phi(Q) \\
&(1,0,1,1) \to T_i = P-Q-\phi(Q) \\
&(0,1,0,0) \to T_i = \phi(P) \\
&(0,1,1,0) \to T_i = \phi(P)-Q \\
&(0,1,0,1) \to T_i = \phi(P)-\phi(Q) \\
&(0,1,1,1) \to T_i = \phi(P)-Q-\phi(Q) \\
&(1,1,0,0) \to T_i = P+\phi(P) \\
&(1,1,1,0) \to T_i = P+\phi(P)-Q \\
&(1,1,0,1) \to T_i = P+\phi(P)-\phi(Q) \\
&(1,1,1,1) \to T_i = P+\phi(P)-Q-\phi(Q)
\end{aligned}
$$

   - $A_i = 2 \cdot A_{i+1} + T_i$
3. Check that $A_0 = 0$

## Scalar Decomposition

We haven't discussed yet how to find $u, v, s, t$. In fact, we can simply split our 128-bit challenge into 4 32-bit chunks and assign those to $u, v, s, t$. We only have to show that

$$a = \frac{u + \omega_q v}{s + \omega_q t}$$

actually maps into a 128-bit space for the value of $a$. [TODO: proof]

## ***Acknowledgments***

Thanks to Liam Eagen for explaining this optimization of the scalar multiplication to me.