Purify and Zero-Knowledge Proofs in `purify-cpp`

Judica, Inc.

March 2026

Abstract

This note explains how `purify-cpp` implements Purify and how the repository builds two proof systems around it. Purify is a provably secure, circuit-friendly pseudorandom function (PRF) [3, 1]: the design pushes the expensive message hashing outside the arithmetic circuit while keeping the in-circuit work close to low-degree field arithmetic. In this repository, the same native Purify verifier circuit supports two different proof encodings:

1. with the imported legacy BulletProofs circuit backend [4], and
2. with a reduction to the `secp256k1-zkp` BulletProofs++ norm-argument backend (the BulletProofs++ path) [6, 7, 8].

The important point is that both backends prove the same underlying Purify statement; they differ only in how each backend encodes that statement into a zero-knowledge proof.

1 Contribution

Purify itself, including the combine formula and the legacy circuit encoding used below, comes from MuSig-DN and the upstream reference implementation [3, 1]. The contribution of `purify-cpp` is a practical C/C++ implementation of that existing construction, together with the engineering required to realize it on both the legacy BulletProofs backend and the BulletProofs++ backend. The discussion below is a high-level explanation of that implementation and backend integration, not a formal proof of Purify or of the circuit encoding.

2 What Purify Is

Purify is a PRF over a prime field $\mathbb{F}_P$ chosen to have low multiplicative complexity in arithmetic circuits. The reference construction fixes curve coefficients $A, B \in \mathbb{F}_P$, a non-square $D \in \mathbb{F}_P$, and then works with two elliptic curves over the same field [3, 1]:

$$E_1 : y^2 = x^3 + Ax + B,$$
$$E_2 : y^2 = x^3 + AD^2x + D^3B.$$

The parameters A, B, and D are special because they do two jobs at once. First, A and B define the base curve E_1. Second, the construction chooses D as a non-square so that E_2 becomes the quadratic twist of E_1 rather than an unrelated second curve. Over a field where $\sqrt{D}$ exists, the two curves become isomorphic, so one can "untwist" a point on E_2 into a corresponding point on E_1. That is exactly why the later combine formula can mix one x-coordinate from E_1 with one normalized x-coordinate from E_2 and still satisfy a single clean algebraic identity.

This is the special relationship Purify exploits: most arbitrary pairs of curves would not give a low-degree formula tying the two branches together. Here the branch on E_2 is not independent of

the branch on E_1; the construction chooses it as the twist partner that shares the same underlying addition law after normalization by D. At the same time, the construction selects parameters so that both curves still have the required cryptographic properties, such as prime order and hard Diffie-Hellman-style problems.

It also fixes generators and hash-to-curve maps

$$G_i \in E_i(\mathbb{F}_P), \qquad H_i : \{0,1\}^* \to E_i(\mathbb{F}_P) \qquad (i \in \{1,2\}).$$

We write

$$X : E_i(\mathbb{F}_P) \setminus \{\mathcal{O}\} \to \mathbb{F}_P$$

for the x-coordinate projection, so $X(P)$ is a field element, not a curve point. We also write $[z]P$ for scalar multiplication of P by z.

The secret key is a pair (z_1, z_2), one scalar for each curve. The public key is the pair of x-coordinates in $\mathbb{F}_P$

$$X_1 = X([z_1]G_1), \qquad X_2 = X([z_2]G_2),$$

which `purify-cpp` packs into one public-key value.

For a message m, the Prover evaluates Purify as follows:

$$\begin{aligned} M_1 = H_1(m) \in E_1(\mathbb{F}_P), \qquad & M_2 = H_2(m) \in E_2(\mathbb{F}_P), \\ [z_1]M_1 \in E_1(\mathbb{F}_P), \qquad & [z_2]M_2 \in E_2(\mathbb{F}_P), \\ u = X([z_1]M_1), \qquad & v = X([z_2]M_2) \in \mathbb{F}_P. \end{aligned}$$

$$\begin{gathered} \text{combine} : \mathbb{F}_P \times \mathbb{F}_P \to \mathbb{F}_P, \\ \text{combine}(u,v) = \frac{\left(u + \frac{v}{D}\right)\left(A + \frac{uv}{D}\right) + 2B}{\left(u - \frac{v}{D}\right)^2}. \end{gathered}$$

This gives the full PRF

$$\begin{aligned} o = \text{Purify}((z_1, z_2), m) &= \text{combine}(X([z_1]H_1(m)), X([z_2]H_2(m))) \\ &= \text{combine}(X([z_1]M_1), X([z_2]M_2)) \\ &= \text{combine}(u, v). \end{aligned}$$

Purify takes the average of the x-coordinate of the sum and the x-coordinate of the difference of two corresponding points on the base curve. The v/D term appears because we start the second point on the D-twist E_2 and then untwist it before applying the common addition law. The combine map is therefore a structured curve-derived identity rather than an arbitrary rational function. For the full derivation, see Appendix A.

The point-to-field projection is what makes the final expression a low-degree arithmetic relation over $\mathbb{F}_P$: the curve operations produce points, but the combine map consumes only their x-coordinates as field elements.

The mathematical output is therefore a field element $o \in \mathbb{F}_P$. Later proof systems do not always reveal that scalar directly. In the public-message setting studied in this repository, the message-derived points M_1, M_2 and the packed public key are public statement data, while z_1, z_2, the intermediate points Q_1, Q_2, and usually the scalar o remain hidden. The verifier checks consistency with the Purify relation through a proof and may learn only a derived public nonce point rather than the raw field element o itself.

2.1 From Purify Output to a Schnorr Nonce

This repository mainly uses that field output as a BIP340 Schnorr nonce. Higher-level protocols such as MuSig-DN consume this nonce shape too [2, 3]. To avoid overloading the earlier curve generators G_1 and G_2, write G_{secp} for the standard generator of secp256k1 [2].

For a nonce-bound input $\widetilde{m}$, first define the raw Purify output

$$r_0 = \text{Purify}((z_1, z_2), \widetilde{m}).$$

The PureSign and PureSign++ layers require the Purify field modulus to equal the secp256k1 scalar modulus:

$$P = n_{\text{secp256k1}}.$$

That means the protocol can read the same byte string directly as a secp256k1 scalar. Except for the single invalid value $r_0 = 0$, it can use the Purify output as a Schnorr nonce scalar without any extra reduction step. The implementation rejects the zero case, which occurs with negligible probability $1/n_{\text{secp256k1}}$.

BIP340 then replaces that raw scalar by the even-Y representative of the same x-only nonce point. Write

$$R_0 = [r_0]G_{\text{secp}}.$$

If R_0 already has even Y, set $r = r_0$. Otherwise set

$$r = n_{\text{secp256k1}} - r_0.$$

In other words,

$$r = \text{BIP340Canon}(r_0),$$

where BIP340Canon means "keep the scalar if its nonce point already has even Y, otherwise negate it modulo $n_{\text{secp256k1}}$." Then

$$R = [r]G_{\text{secp}}.$$

has even Y and the same x-only encoding as R_0. `purify_bip340_nonce_from_scalar(...)` performs exactly this normalization: it rewrites the nonce scalar in place to the even-Y representative corresponding to the returned x-only nonce. The API exposes the result in x-only form, because BIP340 uses that nonce encoding. In the signing layers, Purify therefore serves as:

$$(z_1, z_2), \widetilde{m} \;\longmapsto\; r_0 = \text{Purify}((z_1, z_2), \widetilde{m}) \;\longmapsto\; r = \text{BIP340Canon}(r_0) \;\longmapsto\; R = [r]G_{\text{secp}}.$$

The later proof systems show that the published x-only nonce matches the Purify-derived curve point after this BIP340 normalization step.

2.2 Why Purify Is Useful for Proof Systems

The construction is attractive for zero-knowledge proving because the message-dependent hashing is outside the circuit when the message is public. After the prover and verifier agree on the public curve points $M_1 = H_1(m)$ and $M_2 = H_2(m)$, the remaining work is elliptic-curve scalar multiplication logic and a field-rational expression over the resulting coordinates. That is the reason the upstream Purify materials describe the PRF as having low multiplicative complexity.

In practical terms, this repository exposes Purify through three layers:

Concept	Main API surface
Direct PRF evaluation	`purify::eval`
Message-specific verifier circuit template	`purify::verifier_circuit_template`
Concrete witness assignment for proving	`purify::prove_assignment_data`

3 How `purify-cpp` Maps Purify into a Circuit

The important implementation point is that the repository does *not* build a generic "elliptic-curve circuit" and then plug Purify into it. Instead, it constructs one message-specific symbolic transcript for the exact Purify relation and then lowers that transcript into a native sparse arithmetic circuit. As noted in Section 1, that basic encoding strategy already appears in the upstream Purify reference code; the focus here is how `purify-cpp` implements and reuses it across both the legacy BulletProofs backend and the BulletProofs++ path.

3.1 Step 1: Fix the Public Message Points

For a public message m, the construction first fixes

$$M_1 = H_1(m), \qquad M_2 = H_2(m)$$

outside the circuit. Both the template builder and the witness builder call `hash_to_curve` before they create any circuit logic. Hash-to-curve therefore sits outside the proved statement. The proved relation begins after the builder has already turned the message into the public curve points M_1 and M_2.

That is why the remaining statement is a fixed-base statement:

$$\begin{aligned} P_{1,x} &= X([z_1]G_1), & P_{2,x} &= X([z_2]G_2), \\ Q_{1,x} &= X([z_1]M_1), & Q_{2,x} &= X([z_2]M_2), \\ o &= \mathrm{combine}(Q_{1,x}, Q_{2,x}). \end{aligned}$$

Here the points G_1, G_2, M_1, M_2 are all public constants for the statement. Only the scalars z_1, z_2 are secret. If we write

$$\begin{aligned} P_1 &= [z_1]G_1, \quad P_2 = [z_2]G_2, \\ Q_1 &= [z_1]M_1, \quad Q_2 = [z_2]M_2, \end{aligned}$$

then the public key exposes only the x-coordinates

$$P_{1,x} = X(P_1), \qquad P_{2,x} = X(P_2).$$

The full points P_1, P_2 are therefore not themselves public objects in the API; only their x-only encodings are. By contrast, the points Q_1, Q_2 and their x-coordinates $Q_{1,x}, Q_{2,x}$ are internal witness-derived values. The final relation uses those hidden values to compute

$$o = \mathrm{combine}(Q_{1,x}, Q_{2,x}).$$

Depending on the surrounding protocol, a higher-level layer may later reveal, commit, or convert o into a public nonce point, but the individual Q_i values never enter the statement as public inputs.

3.2 Step 2: Turn the Secret Scalars into Boolean Witnesses

The helper `circuit_main(...)` allocates the secret key as witness bits, not as one giant field element. Concretely:

1. it recodes each secret scalar with `key_to_bits(...)`,
2. it allocates one witness variable per recoded bit with `transcript.secret(...)`, and

3. it constrains each of those variables to be boolean with `transcript.boolean(...)`.

At the symbolic level, "boolean" means the transcript records the multiplicative constraint

$$b(b-1) = 0.$$

Bitness is therefore an explicit proved constraint, not an unchecked assumption.

Here "recoding" means that `key_to_bits(...)` does not keep the scalar in ordinary binary form. Instead, it rewrites it into a signed-window encoding tailored to the later fixed-base multiplier. Write B for the public fixed base point given to one call of `circuit_ec_multiply_x(...)`; depending on the call, B is one of G_1, G_2, M_1, M_2. In the main 3-bit windows, two bits select one of the four odd multiples

$$\{1, 3, 5, 7\} \cdot 2^{3i} B$$

of that fixed base point, while a third bit chooses whether to use that point or its negation. On a short Weierstrass curve, negation is

$$(x, y) \mapsto (x, -y),$$

so this sign choice is cheap in-circuit: the x-coordinate stays the same and only the y-coordinate changes sign. Smaller 1-bit or 2-bit lookup gadgets handle the remaining boundary bits. This is much cheaper than proving a naive bit-by-bit double-and-add computation.

3.3 Step 3: Build Four Fixed-Base Multipliers

The heart of the mapping is `circuit_ec_multiply_x(...)`. It takes

- a *public* base point, such as G_1 or M_1, and
- a *secret* bit-vector witness for z_1 or z_2,

and returns an expression for the final x-coordinate only.

This is where the message-specific structure matters. Since the base point is public, the builder can precompute the required point tables *outside* the circuit:

- successive doubles of the base point,
- odd multiples such as $B, 3B, 5B, 7B$, and
- the affine (x, y) coordinates of those table entries.

Inside the circuit, the secret bits only choose among those precomputed constants. The helpers `circuit_1bit_point`, `circuit_2bit_point`, and `circuit_3bit_point` implement those choices as low-degree multilinear polynomials in the secret bits. For example, a 2-bit selector is not a branch; it is one affine expression plus one product term for the bit-pair.

For example, in the first 3-bit window with fixed base $B = G_1$, the multiplier precomputes

$$\{G_1, 3G_1, 5G_1, 7G_1\}.$$

Two witness bits s_0, s_1 select one of those four affine points with `circuit_2bit_point(...)`. A third witness bit t then optionally negates it by keeping the same x-coordinate and multiplying the y-coordinate by $1 - 2t$, so that window contributes one of

$$\pm G_1, \qquad \pm 3G_1, \qquad \pm 5G_1, \qquad \pm 7G_1.$$

This is where the boolean constraints matter. The lookup formulas are multilinear polynomials in s_0, s_1, t. Because separate constraints prove

$$s_0(s_0 - 1) = 0, \qquad s_1(s_1 - 1) = 0, \qquad t(t - 1) = 0,$$

those constraints force the variables to be actual bits, so the lookup evaluates to one genuine table entry. Without those bit constraints, the same polynomial formulas could evaluate to arbitrary affine combinations that do not correspond to any intended window choice.

The construction does not invoke the later affine-addition helpers on arbitrary pairs of unknown points. In the reference construction it calls them only on affine points that construction rules guarantee are not equal and not negatives of each other, so the denominator $x_2 - x_1$ is non-zero on every valid lookup choice.

After selecting chunk points, the construction applies the usual affine addition law. The helper `circuit_ec_add(...)` computes

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

and then the corresponding y_3. The final combine step only needs an x-coordinate, so the last addition uses `circuit_ec_add_x(...)` and skips the final y computation.

The full Purify statement therefore becomes four copies of the same fixed-base gadget:

$$P_{1,x} = \texttt{MulX}(G_1, z_1), \quad P_{2,x} = \texttt{MulX}(G_2, z_2),$$
$$Q_{1,x} = \texttt{MulX}(M_1, z_1), \quad Q_{2,x} = \texttt{MulX}(M_2, z_2).$$

This is the main circuit insight in the repository: once the message is public, Purify reuses two secret scalars across four fixed-base multipliers.

3.4 Step 4: Express Every Nonlinear Operation with Transcript Gates

The symbolic object used during construction is `Transcript`. It is best read as a small constraint-building language: each operation appends a constraint of a specific form. The important operations are:

- `secret(...)`: allocate a new witness variable,
- `boolean(x)`: append the multiplicative constraint

$$x(x-1) = 0,$$

- `mul(lhs, rhs)`: append a new multiplicative constraint

$$lhs \cdot rhs = out,$$

  where `out` is a fresh witness,
- `div(lhs, rhs)`: append a new multiplicative constraint

$$q \cdot rhs = lhs,$$

  where `q` is a fresh witness representing the quotient, and
- `equal(lhs, rhs)`: append the affine equality

$$lhs - rhs = 0.$$

Addition and subtraction are different: they are not separate transcript operations. They happen at the `Expr` level, where one forms affine combinations such as $x + y$, $x - y$, or $3x - 2y + 7$ without immediately emitting a new constraint. Those affine expressions only become constraints when the builder passes them into one of the transcript operations above, such as `mul(...)`, `div(...)`, `boolean(...)`, or `equal(...)`.

The circuit handles division indirectly because the native constraint system has no inverse gate. Instead, `transcript.div(lhs, rhs)` introduces a quotient witness q and proves only the equivalent product relation $q \cdot rhs = lhs$. When the prover knows concrete values, the witness generator fills in $q = lhs/rhs$, but the resulting check still consists only of multiplication and affine equalities. This also means the circuit is not using a generic "complete" division gadget: for a valid witness, every divisor fed into `div(...)` must already be non-zero. In this Purify circuit that covers both affine-addition slopes and the final combine denominator. The affine-add helpers only receive point pairs that construction rules guarantee are not equal or negatives of each other, and the final combine step likewise requires $u \neq v/D$ on the concrete witness. The the transcript language does not prove those non-vanishing facts by itself; it relies on the circuit construction to ensure them, and the implementation checks them when concrete witness values are available.

Underneath the convenience API, the primitive checks reduce to two types:

- multiplicative checks of the form $lhs \cdot rhs = out$, and
- affine checks of the form $lhs - rhs = 0$.

One can of course rewrite an affine check algebraically as a degenerate product such as

$$(lhs - rhs) \cdot 1 = 0,$$

but the backend does *not* encode it that way. It keeps linear relations in a separate sparse row system, because one affine row can constrain many gate wires and commitment wires at once without consuming an extra multiplication gate. The normalization is:

- `secret(...)` allocates a witness but adds no check by itself,
- `mul(...)`, `boolean(...)`, and `div(...)` all land in the multiplicative bucket, and
- `equal(...)` lands in the affine-equality bucket.

For example, `boolean(x)` emits the multiplicative check $x(x-1) = 0$, and `div(lhs,rhs)` emits the multiplicative check $q \cdot rhs = lhs$ after introducing the quotient witness q.

Within that vocabulary, transcript operations express all of the following:

- bit constraints $b(b-1) = 0$,
- point-selection products such as xy and xyz in the lookup gadgets,
- affine-addition slopes $\lambda = (y_2 - y_1)/(x_2 - x_1)$, and
- the final Purify rational expression

$$\frac{\left(x_1 + \frac{x_2}{D}\right)\left(A + \frac{x_1 x_2}{D}\right) + 2B}{\left(x_1 - \frac{x_2}{D}\right)^2}.$$

In code, `circuit_combine(...)` first forms $v = x_2/D$, then constructs the numerator and denominator with `mul`, and finally enforces the quotient with one more `div`. The final Purify formula is therefore not special-cased at the backend layer; it acts as just another symbolic arithmetic gadget inside the same transcript.

3.5 Step 5: Lower the Transcript into Native BulletProofs Rows

After `circuit_main(...)` finishes, the statement is therefore just a normalized list of emitted constraints in the `Transcript` language:

- one multiplicative bucket containing every instance of $lhs \cdot rhs = out$, including the products introduced by `boolean(...)` and `div(...)`, and
- one affine bucket containing every instance of $lhs - rhs = 0$.

That representation is convenient for building the Purify relation, but it is not yet the native circuit format that the proving backends consume.

The job of `BulletproofTranscript::from_transcript(...)` is to apply one encoding rule to each of those emitted constraints and translate the result into the native circuit object

$$(\texttt{wl}, \texttt{wr}, \texttt{wo}, \texttt{wv}, \texttt{c}).$$

The key point is that the native circuit has *two* layers, not one.

Gate layer. Every recorded multiplicative constraint becomes one numbered gate with wire symbols L_i, R_i, O_i and wire values

$$(a_{L,i}, a_{R,i}, a_{O,i}),$$

enforcing

$$a_{L,i} a_{R,i} = a_{O,i}.$$

Row layer. Every recorded affine equality becomes one sparse linear row over those same gate wires and over the commitment wires.

It is therefore misleading to think "some rows are multiplications." Multiplications become *gates*. The backend uses rows for the affine constraints that tie those gate values together. After lowering, the backend no longer sees elliptic-curve gadgets or transcript expressions. It only sees:

- multiplication gates $(a_{L,i}, a_{R,i}, a_{O,i})$, and
- sparse linear rows tying those gate values together.

An equality such as $lhs - rhs = 0$ is therefore classified as an affine check rather than as a "multiplication by 1" check: the native circuit format has a dedicated linear layer for such relations, and using that layer is both more direct and more efficient.

What the row equation means. These "second kind" checks are just the affine equalities that remain after the lowering has already given all multiplicative constraints their own gates. Concretely, a row appears whenever the backend needs to assert that some wire or public slot equals an affine expression. Typical examples are:

- assignment equalities such as

  $$L_i = 3L_7 - 2O_{11} + 5,$$

  which arise when a gate input is itself an affine expression rather than a bare witness,
- bookkeeping equalities that relate reused witness aliases to already-assigned gate values, and
- the late public-binding equalities $P_{1,x} = \text{pubkey}_1$, $P_{2,x} = \text{pubkey}_2$, and $o = V_0$.

Writing v_k for the commitment-wire values, each such affine row has the conceptual form

$$\sum_i \alpha_i a_{L,i} \; + \; \sum_i \beta_i a_{R,i} \; + \; \sum_i \gamma_i a_{O,i} \; + \; \sum_k \eta_k v_k \; = \; c_j.$$

Here the symbols $a_{L,i}, a_{R,i}, a_{O,i}$ are the *values* assigned to gate i's left, right, and output wires. The coefficients $\alpha_i, \beta_i, \gamma_i, \eta_k$ are not new secret data; they are just the linear coefficients that appear when one lowered transcript equation is written in terms of the BulletProofs wire symbols L_i, R_i, O_i, V_k.

The commitment wires need one more piece of interpretation. They do *not* come from `secret(...)`. A `secret(...)` call allocates an internal witness; the lowering code eventually assigns that witness onto ordinary gate wires and constrains it there. By contrast, a commitment wire V_k is a dedicated scalar slot of the BulletProofs statement itself: it may appear linearly in affine rows, and the proof separately exposes the corresponding public commitment point. In this Purify circuit the lowering introduces exactly one such slot, V_0, through the late output-binding constraint

$$o = V_0.$$

v_0 is therefore not another secret-key coordinate and not another `secret(...)` witness. It is the scalar opening of the committed Purify output. In the prover assignment this value lives in `assignment.commitments[0]`, and the proving wrapper maps it to the public commitment point that accompanies the proof.

Concretely, each row starts from one affine equality that the lowering code produces after it first forms

$$\texttt{combined} = lhs - rhs.$$

After the lowering code replaces witness aliases with wire symbols, that expression has the form

$$\kappa \; + \; \sum_i \alpha_i L_i \; + \; \sum_i \beta_i R_i \; + \; \sum_i \gamma_i O_i \; + \; \sum_k \eta_k V_k.$$

The row then stores those same coefficients in `wl`, `wr`, `wo`, and `wv`,[1] and stores

$$c_j = -\kappa.$$

For example, if one lowered equality becomes

$$7 + 3L_0 - 2R_4 + O_7 + 5V_0 = 0,$$

then row j stores

$$\alpha_0 = 3, \qquad \beta_4 = -2, \qquad \gamma_7 = 1, \qquad \eta_0 = 5, \qquad c_j = -7,$$

and the resulting row checks

$$3a_{L,0} - 2a_{R,4} + a_{O,7} + 5v_0 = -7.$$

Most rows come from one of two sources:

- assignment rows created while lowering each transcript multiplication into concrete wire slots, such as "this left wire equals that affine expression", and
- explicit equality rows, including the late constraints that bind $P_{1,x}$ and $P_{2,x}$ to the public key and bind o to the commitment slot.

Two lowering details are easy to miss but matter:

1. If a transcript witness is just an alias for one BulletProofs wire value, the lowering code maps it directly onto that wire instead of keeping an extra symbolic variable.

2. If a transcript witness appears only in linear equations, the lowering code allocates an extra dummy gate whose left wire carries that witness and whose right and output wires are zero.

After that, the backend pads the number of multiplication gates in the native circuit to the next power of two because both proving backends expect that shape. This gate-count padding is distinct from the later zk slack variable slots in the reduced vector ℓ. For the current Purify parameters the unpadded circuit has about 2030 multiplication gates, and the backend pads the proof-facing shape to 2048 gates.

[1] The packed backend stores commitment-column entries with the opposite sign internally. This matches the imported BulletProofs statement convention and keeps the public commitment point in the natural form $C_k = [v_k]G$ rather than forcing the API to expose $-[v_k]G$ or to negate the opening scalar at the boundary. In this paper's single-commitment case, that lets the protocol publish the public point directly as $R = [r]G$. The displayed equation in the main text is the conceptual affine row; the packed `wv` array represents the same constraint with commitment entries sign-flipped.

3.6 Step 6: Bind the Public Key and the Claimed Output

`verifier_circuit_template(message)` builds the message-specific template. It hashes the message to M_1, M_2, runs `circuit_main(...)` *without* concrete secrets, lowers the symbolic transcript once, and stores three late-bound expressions:

$$P_{1,x}, \qquad P_{2,x}, \qquad o.$$

The template does not immediately tie those expressions to a particular public key. That is why the repository can reuse one message template across many users.

Later, `instantiate_packed(pubkey)` or `add_pubkey_and_out(...)` appends the statement-specific constraints

$$P_{1,x} = \text{pubkey}_1, \qquad P_{2,x} = \text{pubkey}_2, \qquad o = V_0.$$

The first two equalities bind the circuit to the packed public key. The last equality binds the Purify output to commitment slot V_0. In other words, the statement treats the claimed output as the single committed scalar of the BulletProofs-style statement, not as an ordinary public constant row.

Finally, `prove_assignment_data(...)` runs the same symbolic construction with the concrete secret key, checks that the symbolic outputs match native curve arithmetic, and materializes the actual witness columns

$$(a_L, a_R, a_O, V)$$

as a `BulletproofAssignmentData` object.

4 Zero-Knowledge Proofs with BulletProofs

The legacy BulletProofs path proves the native circuit *directly*. After Section 2 builds the circuit, the BulletProofs path performs no further statement reduction.

The key point is the Purify output o. In the protocol, o is meant to remain secret: for nonce proofs, the verifier should learn the public nonce object derived from o, not the scalar itself. Accordingly, Step 6 does *not* bind o to a public field constant. Instead, it binds

$$o = V_0,$$

where V_0 is the single committed-scalar slot of the BulletProofs statement. The BulletProofs backend therefore proves knowledge of a satisfying witness *and* of an opening of that commitment slot, while exposing only the corresponding public commitment point.

Operationally, `prove_experimental_circuit(...)` returns BulletProofs proof bytes together with the public commitment point for V_0. The scalar o itself stays in `assignment.commitments[0]`; the verifier sees only that public point. The API also allows an extra blinding factor for this commitment. If the caller provides a blind, the backend uses a blinded commitment to o. If `blind = std::nullopt`, the wrapper exposes the exact point $[o]$`value_generator` instead.

Legacy PureSign uses that exact-unblinded mode for its nonce proof. There the protocol chooses `value_generator` to be the secp256k1 base generator, so the public commitment point is literally

$$[o]G_{\text{secp}},$$

where o is the raw Purify output carried in the commitment slot. The higher-level nonce object then keeps only the BIP340 x-only encoding of the corresponding even-Y representative. The scalar itself remains hidden; the verifier checks that the x-only projection of the public commitment point matches the advertised nonce.

Inside that wrapper, the imported backend then runs a standard BulletProofs-style polynomial proof for the flattened circuit [4]. The math has three layers.

Witness commitments. First, it commits to the witness vectors. Schematically, it samples fresh blinders α, β, ρ and random masking vectors l_3, r_3, then sends

$$\begin{aligned} A_I &= \mathrm{Com}(a_L, a_R; \alpha), \\ A_O &= \mathrm{Com}(a_O; \beta), \\ S &= \mathrm{Com}(l_3, r_3; \rho). \end{aligned}$$

This is schematic notation: the legacy backend uses a slightly specialized layout for the initial Boolean portion of the witness, but at the level relevant here the roles are simple. A_I commits to the left/right wire vectors, A_O commits to the output-wire vector, and S commits to the random masks. Fiat–Shamir then derives challenges y, z from the transcript.

Polynomial compression. Second, it compresses the whole circuit into one polynomial identity. Using y and z, the backend folds all sparse linear rows, all multiplication gates, and the commitment-wire weights into compressed vectors

$$l_1,\ l_3,\ r_0,\ r_1,\ r_3,\ w_V,$$

with $l_2 := a_O$, and defines

$$\begin{aligned} l(X) &= l_1 X + l_2 X^2 + l_3 X^3, \\ r(X) &= r_0 + r_1 X + r_3 X^3. \end{aligned}$$

For a valid witness, all circuit checks are equivalent to the single degree-6 identity

$$t(X) = \langle l(X), r(X) \rangle.$$

The prover computes the coefficients $t_1, \ldots, t_6$ of that polynomial and sends commitments T_1, T_3, T_4, T_5, T_6; the already-committed A_O term absorbs the X^2 coefficient, which is why there is no separate T_2. Fiat–Shamir then derives a challenge x, and the prover forms

$$l = l(x), \qquad r = r(x), \qquad t = \langle l, r \rangle.$$

The prover folds the blinding scalars in the same way, giving the serialized scalars τ_x and μ that accompany the proof.

Inner-product argument. Third, the backend proves the remaining inner-product claim $\langle l, r \rangle = t$. At this point the verifier's public equations have already tied

- $t, \tau_x, T_1, T_3, T_4, T_5, T_6$ and the public commitment points into one commitment equation, and
- A_I, A_O, S into one vector-opening equation for the hidden vectors l and r.

At that point the remaining hidden statement is exactly one inner product. The backend runs the standard BulletProofs inner-product argument: it hashes the transcript and t to derive the extra generator randomizer, recursively folds the vectors in half, emits pairs of points (L_j, R_j), derives folding challenges x_j, and repeats until only tiny final scalars remain. Verification reconstructs the same folded generator coefficients from the x_j values and checks one final multiexponentiation equation.

Concretely, the prover side does the following:

1. build or reuse the message-specific `NativeBulletproofCircuitTemplate`,

2. instantiate it with the public key so the late constraints $P_{1,x} = \text{pubkey}_1$, $P_{2,x} = \text{pubkey}_2$, and $o = V_0$ are present,

3. generate the witness columns with `prove_assignment_data(...)`, and

4. call `prove_experimental_circuit(...)` on that circuit and assignment.

That last step is the concrete BulletProofs prover call in this repository: the wrapper validates the native circuit and witness assignment, binds the exact circuit and statement bytes into one digest, invokes the imported BulletProofs backend, and returns the proof bytes together with the public commitment point for V_0.

The resulting BulletProofs statement is:

> there exist witness columns (a_L, a_R, a_O) and a committed scalar v_0 such that every multiplication gate holds, every affine row holds, the public-key rows hold, and $o = v_0$.

Zero knowledge hides the witness columns and the scalar v_0; the verifier sees only the public commitment point and the proof transcript.

The verifier reruns the public half of the same setup: it reconstructs the same circuit shape, recomputes the same circuit-binding digest, and feeds the circuit, the public commitment point, and the BulletProofs proof into the verification wrapper. In the nonce-proof wrapper, the verifier also checks that the x-only projection of the public commitment point matches the advertised BIP340 nonce.

The BulletProofs path therefore gives the most literal reading of the repository:

> prove that the supplied wire assignment satisfies the Purify circuit.

The protocol adds one convention: it carries the Purify output as one committed scalar slot rather than as a revealed public field element. Its advantage is conceptual simplicity. Its disadvantage is that proof size and prover work scale with the full 2048-gate-style circuit rather than with a more compressed relation.

5 Zero-Knowledge Proofs with BulletProofs++

The BulletProofs++ path starts from *the same native circuit and the same witness assignment* as Section 3. What changes is the public encoding handed to the backend. Instead of proving the full native gate-and-row system directly, the repository first reduces it to one weighted norm relation of the form

$$\sum_{k=0}^{4N-1} \mu_k n_k^2 + \langle \ell, c \rangle = T, \qquad \mu_k = (\rho^2)^{k+1},$$

and then asks the BulletProofs++ backend to prove knowledge of the hidden vectors n, ℓ satisfying that single equation [5, 6, 7, 8]. BulletProofs++ is therefore not a different Purify statement. It gives a public reduction of the same Purify circuit to a proof system built around weighted norm arguments.

This translation step serves a practical purpose. The repository already has one message-specific Purify circuit template, one witness generator, and one direct BulletProofs proof of that statement. Reusing that same circuit as the source of truth means the BulletProofs and BulletProofs++ paths check against exactly the same semantics and exactly the same witness assignment. A bespoke "native BulletProofs++ circuit" might also be possible in principle,

but then the engineering burden would move to proving that this second hand-written object is really equivalent to the original Purify statement. The reduction used here avoids that duplication: first fix one circuit, then change only the proof encoding. The repository does not implement or benchmark such a bespoke alternative, so it does not claim a concrete efficiency number for "not reusing" the circuit. Since the current circuit is already highly specialized to Purify, with public message hashes, fixed-base tables, and x-only outputs, one should think of the likely upside as an unmeasured constant-factor engineering improvement from backend-specific layout choices rather than an already-demonstrated order-of- magnitude gain.

5.1 Public Reduction to One Norm Relation

Let N be the number of multiplication gates in the native circuit, and index the sparse linear rows by j. The helper `build_circuit_norm_arg_public_data(...)` first hashes the circuit-binding digest to derive:

- a non-zero challenge ρ,
- one weight λ_i for each multiplication gate, and
- one weight σ_j for each linear row.

Row aggregation. For row j, write the original affine check as

$$\sum_i \alpha_i^{(j)} a_{L,i} \;+\; \sum_i \beta_i^{(j)} a_{R,i} \;+\; \sum_i \gamma_i^{(j)} a_{O,i} \;+\; \sum_k \eta_k^{(j)} v_k \;=\; c_j.$$

The row weights then form one weighted outer sum over all rows:

$$\sum_j \sigma_j \left(\sum_i \alpha_i^{(j)} a_{L,i} \;+\; \sum_i \beta_i^{(j)} a_{R,i} \;+\; \sum_i \gamma_i^{(j)} a_{O,i} \;+\; \sum_k \eta_k^{(j)} v_k \right) \;=\; \sum_j \sigma_j c_j.$$

Swapping the order of summation collapses that to one aggregate linear relation

$$\sum_i \alpha_i a_{L,i} \;+\; \sum_i \beta_i a_{R,i} \;+\; \sum_i \gamma_i a_{O,i} \;+\; \sum_k \eta_k v_k \;=\; T_{\mathrm{lin}},$$

where

$$\alpha_i = \sum_j \sigma_j \alpha_i^{(j)}, \qquad \beta_i = \sum_j \sigma_j \beta_i^{(j)}, \qquad \gamma_i = \sum_j \sigma_j \gamma_i^{(j)}, \qquad \eta_k = \sum_j \sigma_j \eta_k^{(j)},$$

and

$$T_{\mathrm{lin}} = \sum_j \sigma_j c_j.$$

In the implementation, these collapsed coefficient vectors are `left_coeffs`, `right_coeffs`, `output_coeffs`, and `commitment_coeffs`. The weighted row constants become part of the eventual public target.

Gate reduction. The gate weights then fold in the multiplicative constraints $a_{L,i} a_{R,i} = a_{O,i}$ by adding the weighted equations

$$\lambda_i (a_{L,i} a_{R,i} - a_{O,i}) = 0.$$

Using

$$a_{L,i} a_{R,i} = \frac{(a_{L,i} + a_{R,i})^2 - (a_{L,i} - a_{R,i})^2}{4},$$

each gate contributes one "plus" square branch, one "minus" square branch, and a linear term in $a_{O,i}$. Write

$$p_i = a_{L,i} + a_{R,i}, \qquad m_i = a_{L,i} - a_{R,i}.$$

At that point the quadratic part of gate i has the form

$$d_i^+ p_i^2 + e_i^+ p_i \qquad \text{and} \qquad d_i^- m_i^2 + e_i^- m_i.$$

"Complete the square" here just means applying the elementary identity

$$dz^2 + ez = d\left(z + \frac{e}{2d}\right)^2 - \frac{e^2}{4d},$$

so this identity absorbs the linear term into a shifted square and pushes the leftover correction into the public constant. Applying that identity with $z = p_i$ and $z = m_i$ gives

$$d_i^+(a_{L,i} + a_{R,i})^2 + e_i^+(a_{L,i} + a_{R,i}) = d_i^+(a_{L,i} + a_{R,i} + s_i^+)^2 - d_i^+(s_i^+)^2,$$
$$d_i^-(a_{L,i} - a_{R,i})^2 + e_i^-(a_{L,i} - a_{R,i}) = d_i^-(a_{L,i} - a_{R,i} + s_i^-)^2 - d_i^-(s_i^-)^2.$$

with

$$s_i^+ = \frac{e_i^+}{2d_i^+}, \qquad s_i^- = \frac{e_i^-}{2d_i^-}.$$

The reduction stores these shifts as `plus_shift[i]` and `minus_shift[i]`.

Ordinary-square form. The target norm relation uses a weighted sum of *ordinary* squares with fixed public weights $\mu_k = (\rho^2)^{k+1}$. To fit that shape, the helper `two_square_terms(...)` splits each completed-square coefficient into two ordinary square coordinates. If one branch contributes

$$dz^2,$$

the helper chooses two field elements t_1, t_2 such that

$$t_1^2 + t_2^2 = d.$$

Concretely, writing $\xi^2 = -1$ in $\mathbb{F}_P$, the reduction sets

$$t_1 = \frac{d+1}{2}, \qquad t_2 = \xi\frac{d-1}{2},$$

so indeed

$$t_1^2 + t_2^2 = \frac{(d+1)^2}{4} - \frac{(d-1)^2}{4} = d.$$

When the reduction places these two coordinates in positions k and $k+1$ of the norm vector, it stores them proportional to $t_1 z\rho^{-(k+1)}$ and $t_2 z\rho^{-(k+2)}$, so the explicit ρ^{-1} factors cancel the backend's fixed weights $\mu_k = (\rho^2)^{k+1}$. After the verifier applies those fixed public weights, the two coordinates contribute exactly

$$t_1^2 z^2 + t_2^2 z^2 = dz^2.$$

`two_square_terms(...)` is therefore just a field-level decomposition of one coefficient-weighted square into two ordinary squares. That is why every original gate contributes *four* entries to the reduced norm vector: two from the "plus" branch and two from the "minus" branch. Concretely, `reduce_experimental_circuit_to_norm_arg(...)` outputs:

$$n \in \mathbb{F}^{4N}, \qquad \ell \in \mathbb{F}^T.$$

The vector n contains the four square coordinates derived from the shifted $a_{L,i} + a_{R,i}$ and $a_{L,i} - a_{R,i}$ values for each gate. The vector ℓ carries the remaining linear witness data. In the default mode its layout is

$$\ell = \big(a_{O,0}, \ldots, a_{O,N-1}, v_0, \ldots, v_{C-1}, \text{zk slack variable slots}\big),$$

where the final tail consists of zk slack variable slots: coordinates that the reduction introduces by rounding the ℓ-vector length up to a power of two after it places the used coordinates. This is distinct from the earlier gate-count padding of the native circuit. In the basic reduction these zk slack variable slots start at zero; in the masked wrapper the prover reuses them as blinding slots. The extra "+1" in the implementation ensures that this rounded ℓ length always leaves at least some slack beyond the used coordinates. The coefficient vector c has the same shape: the first N entries weight the output-wire part, the next C entries weight the commitment-wire part, and the zk slack variable slots are zero.

The public side then supplies

$$c \in \mathbb{F}^T, \qquad T \in \mathbb{F},$$

and the entire native circuit is equivalent to one public relation

$$\sum_{k=0}^{4N-1} \mu_k n_k^2 + \langle \ell, c \rangle = T,$$

with $\mu_k = (\rho^2)^{k+1}$. This is the object the BulletProofs++ backend proves.

5.2 Transparent Reduced-Witness Proof

The simplest BulletProofs++ entry points are:

- `commit_experimental_circuit_witness`,
- `prove_experimental_circuit_norm_arg`, and
- `verify_experimental_circuit_norm_arg`.

These functions expose the reduction in its simplest form. The witness-commit helper first reduces the native assignment to the hidden vectors (n, ℓ) and computes a witness-only commitment

$$C_{\text{wit}} = \mathsf{Com}(n, \ell).$$

The prover then anchors that commitment to the public target by offsetting it with T:

$$C = C_{\text{wit}} + [T]G.$$

`prove_experimental_circuit_norm_arg(...)` therefore returns exactly two public objects:

- the reduced-witness commitment C_{wit}, and
- the inner BulletProofs++ proof bytes.

Importantly, this public reduction still does *not* publish the Purify output o as a public scalar. The output remains the commitment-wire value v_0, and in the reduction that scalar sits only as one hidden coordinate inside ℓ. The public vectors c and the public target T are derived from the circuit shape and statement-binding bytes; they do not reveal the witness values. The verifier therefore never receives a separate field element for o, only one group commitment to the entire reduced witness together with a proof that the committed witness satisfies the norm relation. That said, this subsection presents only the minimal reduced-witness argument. The next subsection describes the full witness-hiding wrapper that the actual zero-knowledge API uses.

The verifier reruns the same *public* reduction from the circuit and the statement-binding bytes, reconstructs the same anchored commitment C from the transmitted C_{wit}, and checks a norm-argument proof for

$$\exists (n, \ell) \text{ such that } \sum_{k=0}^{4N-1} \mu_k n_k^2 + \langle \ell, c \rangle = T$$

under that commitment. This framing makes the BulletProofs++ design easiest to read: it proves the same Purify statement as BulletProofs, but only after first compressing the native circuit to the reduced witness coordinates (n, ℓ).

5.3 The Masked Zero-Knowledge Wrapper Used Here

The repository's zero-knowledge BulletProofs++ path adds one more layer on top of that transparent reduced proof. Starting from the hidden vectors (n, ℓ), the prover first fills the zk slack variable slots in ℓ with fresh blind scalars and then derives random mask vectors

$$r_n \in \mathbb{F}^{4N}, \qquad r_\ell \in \mathbb{F}^T$$

from the statement digest plus a prover nonce. It next computes

$$t_2 = \sum_{k=0}^{4N-1} \mu_k r_{n,k}^2,$$

$$t_1 = 2 \sum_{k=0}^{4N-1} \mu_k n_k r_{n,k} + \langle r_\ell, c \rangle.$$

These are exactly the coefficients that appear when the verifier later checks the masked witness

$$n' = n + x r_n, \qquad \ell' = \ell + x r_\ell.$$

Since the Purify output is exactly the commitment-wire coordinate $o = v_0$ inside ℓ, this masking step hides o in exactly the same way it hides every other reduced witness coordinate: the verifier never sees ℓ itself, only commitments to the masked combinations. Indeed,

$$\sum_{k=0}^{4N-1} \mu_k (n_k + x r_{n,k})^2 + \langle \ell + x r_\ell, c \rangle = T + x t_1 + x^2 t_2.$$

The outer transcript then mirrors the standard BulletProofs masking pattern:

1. commit to the hidden reduced witness,

$$A_{\text{wit}} = \mathsf{Com}(n, \ell),$$

2. anchor that witness commitment to the public statement,

$$A = A_{\text{wit}} + [T]G,$$

3. commit to the mask vectors together with the linear correction term,

$$S = \mathsf{Com}(r_n, r_\ell) + [t_1]G,$$

4. derive the Fiat-Shamir challenge x from the binding digest, A, S, and t_2,
5. form the masked witness (n', ℓ'), and
6. prove the inner norm argument against

$$C_x = A + [x]S + [x^2 t_2]G.$$

This is not just an abstract description: the implementation explicitly recomputes a direct commitment to (n', ℓ') and checks that it equals the combined point C_x before calling the final inner prover. The verifier mirrors the same steps: recompute the public reduction, derive the same challenge x, reconstruct C_x, and verify the final norm argument against that combined commitment. The outer protocol is BulletProofs-like, while the inner statement remains the BulletProofs++ weighted norm relation.

5.4 Public Commitments and PureSign++

The reduction also supports an "externalized commitments" mode:

- `prove_experimental_circuit_zk_norm_arg_with_public_commitments`,
- `verify_experimental_circuit_zk_norm_arg_with_public_commitments`.

In that variant, the commitment-wire scalars are *not* carried inside the hidden ℓ vector. Instead, `build_circuit_norm_arg_public_data(...)` removes their coefficients from c and stores them separately as public commitment coefficients. The caller then supplies the corresponding public commitment points, and the reduction hashes those points into the statement binding so that no one can swap them after the fact.

The prover side also checks that the supplied public commitment points really match the commitment scalars in the witness assignment. When constructing the anchored A commitment, the implementation subtracts the weighted contribution of those already-public points, so the remaining hidden witness is only

$$\ell = (a_{O,0}, \ldots, a_{O,N-1}, \text{zk slack variable slots}).$$

PureSign++ uses this hook: some commitment objects remain explicit public protocol messages, while the hidden Purify witness is still proved through the same masked norm-argument path.

5.5 Efficiency Note on a Bespoke BulletProofs++ Circuit

The most plausible bespoke win would be to share selector algebra across the two branches that use the same secret scalar: one could compute each signed-window selector basis once for z_1 and fan it out to both the G_1 and M_1 tables, and likewise once for z_2 across the G_2 and M_2 tables. That would remove some duplicated lookup products and some generic lowering overhead, but it would not change the basic Purify arithmetic.

This matters for expectations about size. In the current benchmark surface the BulletProofs++ proof payload is about 909 bytes versus about 1124 bytes for legacy BulletProofs, a roughly 20% reduction. The BulletProofs++ backend's proof size is logarithmic in the reduced dimensions, so small structural cleanups do not necessarily shrink the proof at all. The current reused path pads the native circuit to 2048 gates, and that gate-count padding alone makes the dominant reduced dimension $4 \cdot 2048 = 8192$, landing exactly on the 909-byte size tier. A bespoke path that merely avoided this native-circuit padding would likely move to about $4 \cdot 2030 = 8120$, which is enough to drop one tier to about 844 bytes. This estimate is about the norm-vector

dimension $4N$; it does not require removing the separate zk slack variable slots later used in ℓ for the masked wrapper.

The *next* proof-size drop would require pushing the dominant reduced dimension below 4096, which would mean something much more aggressive than just removing bookkeeping. The realistic upside is therefore probably a modest additional byte reduction and some prover speedup, not a second dramatic proof-size collapse.

6 Why Keep Both Proof Paths

The repository keeps both BulletProofs and BulletProofs++ because they serve different engineering purposes:

- BulletProofs **is the direct baseline.** It proves the native circuit without any extra reduction layer and therefore acts as the simplest end-to-end reference for "this Purify statement is satisfiable".
- BulletProofs++ **is the structured path.** It exploits the fact that the Purify circuit can be compressed into one weighted norm relation, which is a much better fit for the BulletProofs++ backend and for the repository's newer PureSign++ surfaces.
- **The benchmark harness benchmarks both backends against the same statement.** The benchmark harness reports both legacy BulletProofs rows (for example `experimental_circuit.legacy_bp.prove`) and the corresponding BulletProofs++ rows, which makes the tradeoff explicit in one place.

7 Caveats

Two caveats are worth stating explicitly:

1. This repository marks the BulletProofs++ interfaces as experimental. They are useful for research, benchmarking, and higher-level protocol work, but callers should still treat them as an evolving proving path.
2. The paper describes one mathematical statement and one masking wrapper, but the concrete BulletProofs++ prover and verifier still depend on a separate experimental implementation of BulletProofs++. The remaining caveat is therefore about implementation maturity and integration risk, not about a different public statement.

8 Summary

Purify is the statement. The native sparse circuit is the common intermediate form. BulletProofs proves that circuit directly. BulletProofs++ first folds the same circuit into a weighted norm argument and then proves that reduced relation, wrapped in a BulletProofs-style masking transcript. That split is the easiest way to understand the design of `purify-cpp`: one circuit, two proving encodings.

A Origin of the Combine Formula

The Purify combine map comes from averaging the sum and difference formulas on the base curve. The second curve E_2 is a D-twist of E_1, so if

$$v = X([z_2]M_2),$$

then

$$\tilde{v} = \frac{v}{D}$$

is the corresponding untwisted x-coordinate on the base curve. Let

$$\tilde{Q} = (\tilde{v}, \tilde{y}_2)$$

denote the corresponding untwisted point on E_1, where $\tilde{y}_2$ is the untwisted y-coordinate. Writing

$$P = (u, y_1), \qquad Q = \tilde{Q} = (\tilde{v}, \tilde{y}_2)$$

as points on the curve $y^2 = x^3 + Ax + B$, the affine addition law gives

$$x_{\text{add}}(P, Q) = X(P + Q) = \frac{(y_1 - \tilde{y}_2)^2}{(u - \tilde{v})^2} - u - \tilde{v},$$

$$x_{\text{sub}}(P, Q) = X(P - Q) = \frac{(y_1 + \tilde{y}_2)^2}{(u - \tilde{v})^2} - u - \tilde{v}.$$

Here $P + Q$ and $P - Q$ mean elliptic-curve point addition in the group law, producing new curve points; $X(P + Q)$ then means "take the resulting point and project it to its x-coordinate in $\mathbb{F}_P$." It is not multiplication. For the addition case, the usual affine slope is

$$\lambda_{\text{add}} = \frac{\tilde{y}_2 - y_1}{\tilde{v} - u},$$

and the x-coordinate formula is $X(P + Q) = \lambda_{\text{add}}^2 - u - \tilde{v}$. Since the formula squares the slope, this is identical to writing

$$\lambda_{\text{add}}^2 = \frac{(y_1 - \tilde{y}_2)^2}{(u - \tilde{v})^2}.$$

The "add" in x_{add} therefore refers to adding the points, not to adding the y-values.

For the subtraction case, $P - Q = P + (-Q)$, so replacing $Q = (\tilde{v}, \tilde{y}_2)$ by $-Q = (\tilde{v}, -\tilde{y}_2)$ turns the slope numerator into $\pm(y_1 + \tilde{y}_2)$, whose square gives the displayed $(y_1 + \tilde{y}_2)^2$ term.

Averaging these two expressions cancels the mixed term in $y_1\tilde{y}_2$:

$$\frac{x_{\text{add}}(P, Q) + x_{\text{sub}}(P, Q)}{2} = \frac{y_1^2 + \tilde{y}_2^2}{(u - \tilde{v})^2} - u - \tilde{v}.$$

This averaging step is useful because neither $x(P + Q)$ nor $x(P - Q)$ alone is naturally an x-only formula: each contains a cross-term involving $y_1 y_2$. Taking the average eliminates that cross-term, so the result depends only on the curve equation and the two x-coordinates. That is exactly what makes Purify circuit-friendly.

Using the curve equation

$$y_1^2 = u^3 + Au + B, \qquad \tilde{y}_2^2 = \tilde{v}^3 + A\tilde{v} + B,$$

this simplifies to

$$\frac{x_{\text{add}}(P, Q) + x_{\text{sub}}(P, Q)}{2} = \frac{(u + \tilde{v})(A + u\tilde{v}) + 2B}{(u - \tilde{v})^2}.$$

Purify uses exactly this average with $\tilde{v} = v/D$.

References

References

[1] Jonas Nick and Pieter Wuille. *Purify: A Provably-Secure PRF with Low Multiplicative Complexity*. GitHub repository and reference implementation. https://github.com/jonasnick/purify

[2] Pieter Wuille, Jonas Nick, and Tim Ruffing. *BIP 340: Schnorr Signatures for secp256k1*. Bitcoin Improvement Proposal 340, Final, 2020. https://bips.dev/340

[3] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. *MuSig-DN: Schnorr Multi-Signatures with Verifiably Deterministic Nonces*. Cryptology ePrint Archive, Paper 2020/1057. ACM CCS 2020. https://eprint.iacr.org/2020/1057

[4] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. *Bulletproofs: Short Proofs for Confidential Transactions and More*. IEEE Symposium on Security and Privacy, 2018. Cryptology ePrint Archive, Paper 2017/1066. https://eprint.iacr.org/2017/1066

[5] Heewon Chung, Kyoohyung Han, Chanyang Ju, Myungsun Kim, and Jae Hong Seo. *Bulletproofs+: Shorter Proofs for a Privacy-Enhanced Distributed Ledger*. IEEE Access, 2022. https://doi.org/10.1109/ACCESS.2022.3167806

[6] Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. *Bulletproofs++: Next Generation Confidential Transactions via Reciprocal Set Membership Arguments*. Cryptology ePrint Archive, Paper 2022/510. https://eprint.iacr.org/2022/510

[7] Blockstream Research. *secp256k1-zkp*. GitHub repository. https://github.com/BlockstreamResearch/secp256k1-zkp

[8] Jonas Nick et al. *secp256k1-zkp*, `bulletproof-musig-dn-benches` branch. GitHub repository used as the upstream implementation reference for the experimental BulletProofs++ backend in this project. https://github.com/jonasnick/secp256k1-zkp