Selected answers for "Ideas about Bulletproofs"

August 28, 2025

1 Source, or caveat

These are LLM answers with my review and edit. Unfortunately only answered a few of the more meaty questions.

2 Answer to section 3.4 task

Setting. Let $\mathbb{G}$ be a cyclic group of prime order q with generators $g_1, \ldots, g_n, h$ that are *independent* (no known discrete-log relations). A *vector Pedersen commitment* to $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{Z}_q^n$ with blinding $r \in \mathbb{Z}_q$ is

$$C = \sum_{i=1}^{n} a_i G_i + rH \in \mathbb{G}.$$

The prover wants to convince the verifier that it *knows* $(\mathbf{a}, r)$ with $C = \sum_i a_i G_i + rH$, without revealing them.

Σ-protocol for knowledge of $(\mathbf{a}, r)$.

1. *Commit.* Prover samples $\mathbf{u} = (u_1, \ldots, u_n) \leftarrow \mathbb{Z}_q^n$ and $v \leftarrow \mathbb{Z}_q$, computes

$$T \;=\; \sum_{i=1}^{n} u_i G_i \;+\; v\,H \in \mathbb{G},$$

   and sends T to the verifier.

2. *Challenge.* Verifier samples $c \leftarrow \mathbb{Z}_q$ uniformly at random and sends c.
3. *Response.* Prover returns

$$s_i \;=\; u_i + c\,a_i \pmod q \quad (i = 1, \ldots, n), \qquad s_r \;=\; v + c\,r \pmod q.$$

Verification. Accept iff

$$\sum_{i=1}^{n} s_i G_i \;+\; s_r\,H \;\stackrel{?}{=}\; T \;+\; c\,C \quad \text{in } \mathbb{G}.$$

Completeness. If $s_i = u_i + ca_i$ and $s_r = v + cr$, then

$$\sum_i s_i g_i + s_r h = \sum_i (u_i + ca_i) g_i + (v + cr)h = \Big(\sum_i u_i g_i + vh\Big) + c\Big(\sum_i a_i g_i + rh\Big) = T + cC.$$

Some extra color that ChatGPT helpfully added (which by now should be easy for you to understand):

Special soundness (PoK). Given two accepting transcripts with the same T but challenges $c \neq c'$:

$$(T, c, \{s_i\}, s_r) \quad \text{and} \quad (T, c', \{s_i'\}, s_r'),$$

extract the witness by

$$a_i \;=\; (s_i - s_i')\,(c-c')^{-1} \bmod q, \qquad r \;=\; (s_r - s_r')\,(c-c')^{-1} \bmod q.$$

Hence any prover convincing the verifier can be converted into a knowledge extractor.

(Honest-verifier) Zero-knowledge. Simulator chooses $c, \{s_i\}, s_r \leftarrow \mathbb{Z}_q$ uniformly and sets $T := \sum_i s_i g_i + s_r h - cC$. The simulated $(T, c, \{s_i\}, s_r)$ is distributed identically to a real transcript against an honest verifier.

Communication/complexity. The proof consists of *one* group element T and $n+1$ scalars $(s_1, \ldots, s_n, s_r)$, i.e. $O(n)$ size. Prover and verifier do $O(n)$ multi-scalar multiplications. A non-interactive proof follows by Fiat–Shamir: $c := \mathsf{H}(C \,\|\, T)$.

Remark (two-vector variant, optional). If the commitment is $C = \sum_i a_i g_i + \sum_i b_i h_i$ to two vectors $\mathbf{a}, \mathbf{b}$ under independent bases $\{g_i\}$ and $\{h_i\}$ (no single blinder), the same pattern applies: commit with $T = \sum_i u_i g_i + \sum_i v_i h_i$, respond $s_i = u_i + ca_i$, $t_i = v_i + cb_i$, and verify $\sum_i s_i g_i + \sum_i t_i h_i \stackrel{?}{=} T + cC$.

3 Answer section 3.5.1

As far as I can tell this has no hallucinations, basically it's right.

Setting. We have independent generators $\mathbf{G} = (G_1, \ldots, G_n)$ and a vector $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{Z}_q^n$. The commitment is

$$P \;=\; \langle \mathbf{a}, \mathbf{G} \rangle \;=\; \sum_{i=1}^{n} a_i G_i.$$

We want a *logarithmic-size proof* that P commits to $\mathbf{a}$, using only folding on one vector. **One-round fold (length m even).** Split into halves: $\mathbf{a} = (\mathbf{a}_L, \mathbf{a}_R)$, $\mathbf{G} = (\mathbf{G}_L, \mathbf{G}_R)$. Define the two cross-commitments:

$$L \;=\; \langle \mathbf{a}_L, \mathbf{G}_R \rangle, \qquad R \;=\; \langle \mathbf{a}_R, \mathbf{G}_L \rangle.$$

Hash the transcript to a challenge $x \in \mathbb{Z}_q^*$ (Fiat–Shamir). Fold vectors and bases:

$$\mathbf{a}' \leftarrow \mathbf{a}_L x + \mathbf{a}_R x^{-1}, \qquad \mathbf{G}' \leftarrow \mathbf{G}_L x^{-1} + \mathbf{G}_R x.$$

Update the commitment:

$$P' \;\leftarrow\; P \;+\; x^2 L \;+\; x^{-2} R.$$

Identity check. By bilinearity,

$$\langle \mathbf{a}', \mathbf{G}' \rangle = \langle \mathbf{a}_L, \mathbf{G}_L \rangle + \langle \mathbf{a}_R, \mathbf{G}_R \rangle + x^2 \langle \mathbf{a}_L, \mathbf{G}_R \rangle + x^{-2} \langle \mathbf{a}_R, \mathbf{G}_L \rangle = P + x^2 L + x^{-2} R = P'.$$

Thus the fold preserves the commitment relation.

Protocol (length $n = 2^k$). Repeat the above $k = \log_2 n$ times. The prover sends (L_i, R_i) each round, the verifier derives x_i, folds $(\mathbf{a}, \mathbf{G}, P)$, and at the end $\mathbf{a}$ and $\mathbf{G}$ are scalars (a, G) (human note: "scalar" in the sense of non-vector! not in the other sense). The prover reveals a, and the verifier checks $P_{\text{final}} \stackrel{?}{=} a\,G$. *Proof size:* $2 \log_2 n$ points + 1 scalar.

Rust-like pseudocode (concise).

```
// Abstract group/field types:
trait Group { /* +, *, scalar mul, etc. */ }
trait Scalar { fn inv(&self) -> Self; }
fn msm(points: &[Group], scalars: &[Scalar]) -> Group { /* MSM */ }

struct Proof { L: Vec<Group>, R: Vec<Group>, a_final: Scalar }

fn fold_prove(mut G: Vec<Group>, mut a: Vec<Scalar>,
              mut P: Group, tr: &mut Transcript) -> Proof {
```

```
    assert!(a.len().is_power_of_two());
    let mut Ls = Vec::new(); let mut Rs = Vec::new();
    while a.len() > 1 {
        let m = a.len()/2;
        let (aL,aR) = (a[..m].to_vec(), a[m..].to_vec());
        let (gL,gR) = (G[..m].to_vec(), G[m..].to_vec());

        let L = msm(&gR, &aL);           // <aL, gR>
        let R = msm(&gL, &aR);           // <aR, gL>
        tr.absorb_point(&L); tr.absorb_point(&R);
        let x  = tr.challenge_scalar();
        let ix = x.inv();

        Ls.push(L); Rs.push(R);

        // Fold vectors/bases component-wise:
        let a_new: Vec<Scalar> = aL.iter().zip(aR.iter())
            .map(|(al,ar)| *al * x + *ar * ix).collect();
        let G_new: Vec<Group>  = gL.iter().zip(gR.iter())
            .map(|(gl,gr)| *gl * ix + *gr * x).collect();

        // Update commitment
        P = Ls.last().unwrap() * (x*x) + P + Rs.last().unwrap() * (ix*ix);

        a = a_new; G = G_new;
    }
    Proof { L: Ls, R: Rs, a_final: a[0] }
}

fn fold_verify(mut G: Vec<Group>, mut P: Group,
               proof: &Proof, tr: &mut Transcript) -> bool {
    for i in 0..proof.L.len() {
        let L = proof.L[i]; let R = proof.R[i];
        tr.absorb_point(&L); tr.absorb_point(&R);
        let x  = tr.challenge_scalar();
        let ix = x.inv();
        // Fold bases and commitment like the prover:
        let m = G.len()/2;
        let (gL,gR) = (G[..m].to_vec(), G[m..].to_vec());
        G = gL.iter().zip(gR.iter()).map(|(gl,gr)| *gl * ix + *gr * x).collect();
        P = L * (x*x) + P + R * (ix*ix);
    }
    // Final check with scalar a_final:
    let g_final = G[0];
    P == g_final * proof.a_final
}
```

Python-like pseudocode (minimal).

```
# Group/Scalar provide +, *, inv(), msm(points, scalars)
class Proof:
    def __init__(self, L, R, a_final): self.L, self.R, self.a_final = L, R, a_final

def fold_prove(G, a, P, tr):
    assert len(a) & (len(a)-1) == 0  # power of two
    Ls, Rs = [], []
```

```
    while len(a) > 1:
        m = len(a)//2
        aL, aR = a[:m], a[m:]
        gL, gR = G[:m], G[m:]
        L = msm(gR, aL)          # <aL, gR>
        R = msm(gL, aR)          # <aR, gL>
        tr.absorb_point(L); tr.absorb_point(R)
        x  = tr.challenge_scalar(); ix = x.inv()
        Ls.append(L); Rs.append(R)
        a = [al*x + ar*ix for al,ar in zip(aL, aR)]
        G = [gl*ix + gr*x for gl,gr in zip(gL, gR)]
        P = L*(x*x) + P + R*(ix*ix)
    return Proof(Ls, Rs, a[0])

def fold_verify(G, P, proof, tr):
    for L, R in zip(proof.L, proof.R):
        tr.absorb_point(L); tr.absorb_point(R)
        x  = tr.challenge_scalar(); ix = x.inv()
        m = len(G)//2
        gL, gR = G[:m], G[m:]
        G = [gl*ix + gr*x for gl,gr in zip(gL, gR)]
        P = L*(x*x) + P + R*(ix*ix)
    return P == G[0] * proof.a_final
```

Remarks.

- This is exactly the IPA folding logic with the inner-product terms omitted. The algebraic identity $\langle \mathbf{a}', \mathbf{g}' \rangle = P'$ is what makes it work.
- Proof size is $2 \log_2 n$ points + 1 scalar (the revealed a).
- If n is not a power of two, pad $\mathbf{a}$ and $\mathbf{G}$ with zeros to the next power of two. (human note: the as, yes, but the Gs will not be zeros here).

4 Answer Section 3.5.2

Setting. We are in the single-vector folding protocol of Section 3.5.1. Let $\mathbf{G} = (g = G_1, \ldots, G_n)$ be public, $P = \sum_{i=1}^{n} a_i G_i$ be a commitment to a secret vector $\mathbf{a} \in \mathbb{Z}_q^n$, and suppose an adversarial prover outputs an accepting proof consisting of pairs (L_i, R_i), Fiat–Shamir challenges $x_i \in \mathbb{Z}_q^*$, and a final scalar a_{fin} such that the verifier's check passes.

Goal (soundness / PoK). Show that from an accepting prover we can *extract* the full witness $\mathbf{a}$ with overwhelming probability (knowledge soundness).

Key identity per fold (from Section 3.5.1). At one folding step (length m even), with a single vector $\mathbf{a} = (\mathbf{a}_L, \mathbf{a}_R)$ and bases $\mathbf{G} = (\mathbf{G}_L, \mathbf{G}_R)$, the protocol forms

$$L = \langle \mathbf{a}_L, \mathbf{G}_R \rangle, \qquad R = \langle \mathbf{a}_R, \mathbf{G}_L \rangle,$$

derives a challenge $x \in \mathbb{Z}_q^*$, and sets

$$\mathbf{a}' = x\, \mathbf{a}_L + x^{-1}\, \mathbf{a}_R, \qquad \mathbf{G}' = x^{-1}\, \mathbf{G}_L + x\, \mathbf{G}_R, \qquad P' = P + x^2 L + x^{-2} R.$$

The acceptance condition preserves the commitment relation:

$$\langle \mathbf{a}', \mathbf{G}' \rangle = P'.$$

Extractor by *forking* the first challenge (uses the hint). Program the random oracle (Fiat–Shamir) and run the prover twice with the *same* pre-challenge transcript (so L, R agree) but two distinct challenges $x \neq x'$:

$$\mathbf{a}' = x\, \mathbf{a}_L + x^{-1} \mathbf{a}_R, \qquad \widetilde{\mathbf{a}}' = x'\, \mathbf{a}_L + {x'}^{-1} \mathbf{a}_R.$$

This is a 2×2 linear system in the unknown half-vectors $(\mathbf{a}_L, \mathbf{a}_R)$, *coordinatewise.* Its determinant is $\Delta = x \cdot x'^{-1} - x' \cdot x^{-1} = (x^2 - x'^2)/(xx')$, which is nonzero except with probability $O(1/q)$ when $x^2 = x'^2$. Hence, as suggested in the help hint, we can recover the halves explicitly:

$$\mathbf{a}_L = \frac{x\,\mathbf{a}' - x'\,\widetilde{\mathbf{a}}'}{x^2 - x'^2}, \qquad \mathbf{a}_R = x\,\mathbf{a}' - x^2\mathbf{a}_L \;=\; x'\,\widetilde{\mathbf{a}}' - x'^2\mathbf{a}_L.$$

Thus, with two accepting transcripts differing only in x, we uniquely determine $(\mathbf{a}_L, \mathbf{a}_R)$.

Recursing the extraction. Apply the same forking step recursively to the next level on each half:

$$\mathbf{a}_L = (\mathbf{a}_{LL}, \mathbf{a}_{LR}), \qquad \mathbf{a}_R = (\mathbf{a}_{RL}, \mathbf{a}_{RR}),$$

each time rewinding *at that level's first challenge* to obtain two distinct challenges and solving the same 2×2 system to split the current vector into its halves. After $k = \log_2 n$ levels, we recover every coordinate of $\mathbf{a}$.

Why the verifier's data suffice. At each level, the extractor controls the Fiat–Shamir challenges and keeps the pre-challenge transcript fixed (so the sent (L, R) points at that level are identical across the fork). The prover's acceptance at the end ensures each fork produces a consistent folded instance (hence valid $\mathbf{a}'$ or $\widetilde{\mathbf{a}}'$ along that branch), enabling the linear solve above. The base updates $\mathbf{g} \mapsto \mathbf{g}'$ and $P \mapsto P'$ are public and deterministic, so no ambiguity arises.

Conclusion (knowledge soundness). Except with probability $O(k/q)$ over the random challenges (the bad event that some $x^2 = x'^2$), the extractor reconstructs the full witness $\mathbf{a}$. Therefore, the single-vector folding protocol of Section 3.5.1 is a *proof of knowledge* under the Fiat–Shamir (random oracle) heuristic: any adversary that produces an accepting proof can be efficiently rewound to yield $\mathbf{a}$.