On-Chain Bayer–Groth Shuffle Verification for Dealerless Texas Hold'em

Jovonni L. Pharr
Georgia Cyber Warfare Range / BlackMind
info@gacwr.org

November 7, 2026
v0.0.1

Abstract

We present memepoker, a nine-handed Texas Hold'em system on Solana in which there is no dealer, no server and no house: the players' own clients shuffle and deal the cards to each other cryptographically, and Solana programs verify every cryptographic step before the game is allowed to proceed. Cards are ElGamal ciphertexts over Ristretto255 under a joint key that is the sum of per-hand player keys, each registered with a Schnorr proof of knowledge. Every player shuffles the deck in turn and proves the shuffle with a transparent Bayer–Groth argument, so no trusted setup is involved; cards are opened by threshold decryption with Chaum–Pedersen proofs for every share; and a single Fiat–Shamir transcript format, bound to program, cluster, table, hand, seat and phase, is shared byte for byte by the Rust client, the WebAssembly client and the on-chain verifier. The verifier runs natively on Solana's `curve25519` and `sha256` syscalls with a 32-bit-limb scalar arithmetic that cuts scalar multiplication from 5,711 to 1,473 compute units, and it splits the argument into nine independent checks over a 290-byte scratch state so verification can be resumed across transactions. Measured on the compiled verifier in a benchmark harness with the fewest accounts per transaction, one 52-card shuffle round is eight transactions (seven chunk uploads and one verification transaction at 1,016,627 compute units, under the 1.2M per-step budget we set ourselves), a decryption share verifies in about 16,000 compute units, and a complete six-player hand needs 90 cryptographic transactions and 7.9M compute units. The deployed poker program, which passes every account inline, splits verification over three transactions: a round is 14 transactions and about 1.44M compute units, and a six-player hand at most 126 cryptographic transactions and 12.6M compute units. Around the cryptography we build a validity-first enforcement layer: timeouts that any player can resolve by piggybacking on their own move under a virtual clock, SOL-only seat bonds with slashing, whistleblower proofs against key sharing, a second, independently written verifier that backs re-verification fraud proofs, and permanent table freezes on invariant violations. We state the security definitions and give theorems with proof sketches for deck privacy against any coalition that excludes one honest player, soundness of the dealt cards, accountability of every stall, and the fee and chip invariants. We also describe the red-team methodology behind them: more than a thousand named attack tests, per-instruction mutation batteries, coverage-guided and stateful fuzzing, and misbehaving-client runs on a local fork of the chain. Finally, any third party can verify a finished hand: an independent verifier, running natively or in the browser, re-derives the hand from plain chain data and re-checks every key, shuffle and share proof, the privacy of every hole card, the announced cards and the payouts. It trusts the chain's data but not the RPC node that serves it. The program publishes a count and a digest of every step of the hand when it settles, so a node that omits transactions can make verification fail but never pass. To our knowledge this is the first system to verify a transparent shuffle argument (Bayer–Groth) natively inside Solana programs.

	cards hidden on a public chain	nobody can see the deck	deal checked before play	no trusted setup	no challenge window
trusted server or dealer	●	○	○	–	●
randomness beacon / VRF on chain	○	○	●	●	●
commit–reveal of player seeds	○	○	●	●	●
TEE enclave or MPC committee	●	◐	◐	●	●
SNARK-wrapped shuffle (zkShuffle)	●	●	●	○	●
optimistic verification	●	●	○	●	○
memepoker (mental poker, transparent Bayer–Groth, on chain)	●	●	●	●	●

● holds ◐ only if a committee or enclave is trusted ○ does not hold – not applicable

Figure 1: Dealing approaches for a card game on a public chain and the guarantees each gives. Commit–reveal and on-chain randomness make the deck public once it is used; a server, enclave or committee holds or can reconstruct it; a SNARK-wrapped shuffle needs a setup; optimistic checking needs a watcher and a window. Mental poker with a transparent shuffle argument verified on chain gives all five.

1 Introduction

Online poker has always required trust in a dealer. A centralized site holds the deck, sees every card, and is the only party that can say whether a shuffle was fair. Moving the game onto a public blockchain removes the operator as custodian of funds, but it creates a sharper version of the dealing problem: every byte written to a public chain is visible to everyone, so the chain itself cannot hold a secret deck, and any party that does hold the deck becomes a dealer again. The classical answer is *mental poker* [1]: the players deal to each other with cryptography, so that no single party ever knows the order of the deck and every card is opened only with the cooperation of the right players [2].

Mental poker has been understood in theory for decades. What has been missing is a version that a smart-contract platform can actually *enforce*. The chain must check that each shuffle is a permutation of the previous deck and not a deck stacked with aces, that every decryption share is the correct one, and that a player who stops cooperating loses something. Every one of those checks has to fit inside the platform's execution limits. On Solana those limits are tight: a transaction is at most 1,232 bytes, compute is capped at 1.4 million compute units (CU) per transaction, the heap is 32 KiB, and a verifier that does its own field arithmetic in software will exhaust its budget long before it finishes one shuffle proof.

Why the existing approaches fall short here. There are three common ways to deal on chain. The first is to trust a server or an oracle (a randomness beacon, a VRF, a TEE, an MPC network) to hold or generate the deck. That reintroduces a dealer with authority over the cards. A randomness oracle on a public chain cannot even keep the cards hidden, because the randomness is public. The second is to wrap the shuffle in a zk-SNARK, as in the zkShuffle line of work [34]. This gives constant-size proofs and cheap pairing-based verification, but it needs a circuit-specific trusted setup, the circuits need their own audit, and client proving takes seconds to tens of seconds in a browser. The third is to verify only optimistically and to rely on someone watching and challenging within a window. That adds a window to every hand and makes security depend on liveness of a watcher. memepoker takes a fourth path: a *transparent* shuffle argument (Bayer–Groth [3]) over a group for which Solana exposes native syscalls (Ristretto255 [4]), verified in full on chain before any card is dealt. Figure 1 compares what each approach guarantees.

Contributions. This paper documents the system as built in the memepoker repository: the Rust cryptography crates (`pcrypto_*`), the on-chain verifier backend (`pcrypto_solana`) and the poker program (`mp_poker`). Concretely:

- A complete mental-poker protocol for nine-handed Texas Hold'em over Ristretto255 (Sections 2 and 5): a per-hand joint ElGamal key with rogue-key protection, one verifiable shuffle per player, threshold dealing in which a hole card can be opened only by its owner, and public board and showdown reveals. Every message carries a proof that the chain checks.
- A resumable Bayer–Groth verifier designed for constrained runtimes (Section 6). The argument is evaluated as nine independent checks over a 290-byte scratch state: eight multiscalar identities (nine MSMs, since the ciphertext check needs one per component) and one scalar end condition. The checks can be scheduled across transactions in any grouping, and in our benchmark harness an $N = 52$ shuffle verifies in one transaction of 1,016,627 CU.
- A syscall backend and the arithmetic tricks that make the verifier fit: Ristretto validation, group operations and MSM through `sol_curve_*`, a single buffered `sol_sha256` per Fiat–Shamir challenge (7,229 CU instead of 151,576 CU in software), and a 32-bit-limb scalar implementation that is 3.9 times cheaper than the default for multiplication.
- A validity-first enforcement layer (Section 7): piggybacked timeouts under a virtual clock, SOL-only bonds and slashing, whistleblower proofs, second-line re-verification fraud proofs backed by an independently written verifier, invariant proofs, and an optional optimistic claims mode.
- Public verifiability (Section 9): an independent "verify this hand" verifier that rebuilds a finished hand from chain data alone and re-checks it off chain, with a trust model in which the RPC node can withhold data but cannot make any check pass, because the program commits to the hand's complete record at settlement, and a proposition that a verified deal is a fair deal if one participant was honest.
- A security analysis with definitions, theorems and proof sketches (Section 8), measured performance on the compiled programs (Section 10), and the red-team, fuzzing and misbehaving-client methodology used to test it (Section 11).

Novelty. To our knowledge, among the systems we are aware of (the related work in Section 12, as of November 2026; this was not a systematic literature search), memepoker is the first system to verify a transparent verifiable-shuffle argument (Bayer–Groth, with no trusted setup and no SNARK wrapper) natively inside Solana programs. It is also the first dealerless mental-poker protocol on Solana in which every key commitment, every shuffle and every decryption share is verified on chain before play continues. Other Solana card games deal through a VRF or a trusted-execution enclave [37], an MPC network [38], or a Groth16-verified deck commitment held by a single dealer [39]. Existing on-chain mental-poker verifiers target EVM chains or Starknet and wrap the shuffle in a SNARK [34, 35, 36]; ProofPlay [23] pairs a Bayer–Groth proof with a SNARK for each shuffle, and Insta-Pok3r [22] moves shuffling to an offline committee. Solana's own ZK ElGamal Proof program already verifies sigma proofs over twisted ElGamal on curve25519 for confidential transfers [40], so our Chaum–Pedersen verification is not new in kind. What is new is the shuffle argument and the complete card protocol (Section 12).

We are explicit throughout about what is measured, what is estimated and what is proposed but not yet decided. Section 13 is candid about the limits: what contracts cannot prevent (players sharing their own cards out of band), what the analysis assumes (the random-oracle model for Fiat–Shamir), and which numbers were not re-measured in the runs reported here.

2 Notation and Definitions

We define every symbol used in this paper. Table 1 collects them.

The group. $\mathbb{G}$ is the Ristretto255 group [4, 5], the prime-order quotient of Curve25519 [6] built by the Decaf construction. Its order is $\ell = 2^{252} + 27742317777372353535851937790883648493$, a prime, so $\mathbb{G}$ has no small subgroups and no cofactor to clear. Points are written in capitals and use additive notation. G is the standard base point and $\mathcal{O}$ is the identity. Every point travels as its unique 32-byte canonical encoding. A decoder that accepts only canonical encodings rejects all byte strings that are not points, and *every* point taken from player data is decoded this way, on chain with `sol_curve_validate_point`. Scalars live in $\mathbb{Z}_\ell$ and travel as canonical 32-byte little-endian integers below ℓ; non-canonical scalars are rejected, so no proof has two encodings. We write $a \xleftarrow{\$} \mathbb{Z}_\ell$ for uniform sampling.

Hardness assumptions. We use the standard assumptions in $\mathbb{G}$:

Definition 2.1 (DL, DDH)**.** *The* discrete-logarithm *(DL) problem is: given (G, xG) for $x \xleftarrow{\$} \mathbb{Z}_\ell$, find x. The* decisional Diffie–Hellman *(DDH) problem is: distinguish (G, aG, bG, abG) from (G, aG, bG, cG) for $a, b, c \xleftarrow{\$} \mathbb{Z}_\ell$. We assume every probabilistic polynomial-time (PPT) adversary has negligible advantage in both. For Ristretto255 the best known attacks are generic, at about 2^{126} group operations.*

Fiat–Shamir challenges are derived from SHA-256 [7], which we model as a random oracle [8] in the proofs.

ElGamal over a joint key. With per-hand secret keys $sk_i \in \mathbb{Z}_\ell$ and public keys $pk_i = sk_i G$, the joint key of the n participants is $PK = \sum_{i=1}^{n} pk_i$. A card is a point M and its ciphertext is

$$\mathsf{Enc}_{PK}(M; r) = (C_1, C_2) = (rG,\ M + rPK), \qquad r \in \mathbb{Z}_\ell.$$

Ciphertexts add componentwise, and $\mathsf{Enc}_{PK}(\mathcal{O}; \rho)$ is an encryption of the identity. *Re-encryption* adds one: $C + \mathsf{Enc}_{PK}(\mathcal{O}; \rho) = \mathsf{Enc}_{PK}(M; r + \rho)$. A *decryption share* of participant i is $D_i = sk_i C_1$, and $M = C_2 - \sum_i D_i$. A ciphertext with $C_1 = \mathcal{O}$ (randomness zero) is *trivial*: the canonical initial deck is trivial and nothing else may be.

Cards. The 52 cards are fixed public points $M_j = \mathsf{HashToGroup}(\texttt{"memepoker/card/v1"}, j)$, $j = 0, \ldots, 51$. $\mathsf{HashToGroup}(\lambda, i)$ is the Ristretto Elligator map applied to the 64 bytes

$$\mathrm{SHA256}(\mathsf{lp}(\lambda) \| \mathsf{lp}(i) \| \texttt{0x00}) \,\|\, \mathrm{SHA256}(\mathsf{lp}(\lambda) \| \mathsf{lp}(i) \| \texttt{0x01}),$$

where $\mathsf{lp}(x) = \mathrm{u32_le}(|x|) \| x$ is length-prefixing. This is a nothing-up-my-sleeve construction: no one knows a discrete-log relation among the M_j, or between them and G. Verifiers never run the map; they compare against a pinned table that a test re-derives.

Pedersen vector commitments. The shuffle argument commits to vectors of length $n \leq 64$ with the key $(H, G_1, \ldots, G_{64})$, again derived by $\mathsf{HashToGroup}$ (label `"pcrypto/shuffle/v1/commitment-key"`) and pinned:

$$\mathsf{com}(a_1, \ldots, a_n; r) = rH + \sum_{l=1}^{n} a_l G_l.$$

It is perfectly hiding, and computationally binding under DL because no one knows relations among the generators.

Table 1: Symbol reference.

Symbol	Meaning
$\mathbb{G}$, ℓ, G, $\mathcal{O}$	Ristretto255, its prime order, base point, identity
n (players)	participants dealt into the hand, $2 \le n \le 9$
sk_i, pk_i, PK	per-hand secret and public key of seat i; joint key $\sum_i pk_i$
M_j	the public point of card $j \in \{0, \dots, 51\}$
$C = (C_1, C_2)$	an ElGamal ciphertext; a deck is a vector of $N = 52$ of them
π_i, $\rho_{i,k}$	seat i's secret permutation and re-encryption scalars
D_i	seat i's decryption share $sk_i C_1$
$N = m \cdot n$	shuffle matrix shape, m columns of length n (here 4×13)
$H, G_1, \dots, G_{64}$	Pedersen commitment key
$\mathsf{com}(\cdot\,; r)$	Pedersen vector commitment with randomness r
x, y, z	Fiat–Shamir challenges of the shuffle argument
$\mathsf{lp}(\cdot)$	32-bit little-endian length prefixing
tag	transcript domain tag (application, purpose, program, cluster, table, hand, step)

3 Cryptographic Building Blocks

Every proof in memepoker is a public-coin, special-sound, honest-verifier zero-knowledge (HVZK) protocol [9, 10] made non-interactive with the Fiat–Shamir transform [11]. This section gives the transcript discipline they share, then each proof.

3.1 One transcript format, everywhere

Fiat–Shamir implementations fail in practice when the prover and verifier hash different things, or when the challenge does not bind the full statement (the "weak Fiat–Shamir" failure of [12], rediscovered in deployed systems as the Frozen Heart class [13]). memepoker uses exactly one byte format, implemented once in `pcrypto_transcript`. The host (`sha2` crate), the WebAssembly client and the on-chain verifier (`sol_sha256`) differ only in the hash backend.

A transcript is a sequence of SHA-256 *segments*. Segment 0 begins with the domain tag

$$\mathsf{lp}(\texttt{"pcrypto-transcript/v1"}) \,\|\, \mathsf{lp}(\mathit{app}) \,\|\, \mathsf{lp}(\mathit{purpose}) \,\|\, \mathsf{lp}(\mathit{program_id}) \\ \|\, \mathsf{lp}(\mathit{genesis}) \,\|\, \mathsf{lp}(\mathit{instance}) \,\|\, \mathsf{lp}(\mathit{session}) \,\|\, \mathsf{lp}(\mathit{step}),$$

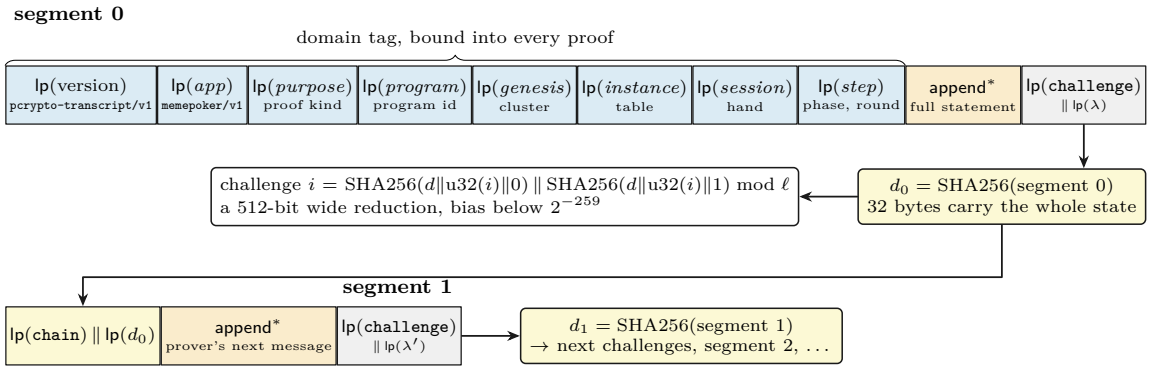
$\mathsf{append}(\lambda, d)$ adds $\mathsf{lp}(\lambda)\|\mathsf{lp}(d)$, and $\mathsf{challenge}(\lambda)$ closes the segment with $\mathsf{lp}(\texttt{"challenge"})\|\mathsf{lp}(\lambda)$ and computes $d = \mathrm{SHA256}(\text{segment})$. The next segment starts with $\mathsf{lp}(\texttt{"chain"})\|\mathsf{lp}(d)$, so the entire state between challenges is the 32-byte d (Figure 2).

The i-th challenge scalar is

$$\mathrm{SHA256}(d \| \mathrm{u32_le}(i) \| \texttt{0x00}) \,\|\, \mathrm{SHA256}(d \| \mathrm{u32_le}(i) \| \texttt{0x01}) \bmod \ell,$$

which is a 512-bit wide reduction with statistical bias below 2^{-259}.

In memepoker the tag binds *app* = `memepoker/v1`, the proof kind as *purpose* (`keycommit`, `shuffle`, `share`, `reveal`, `whistle`), the program id, the genesis hash configured in `ProtocolConfig` at protocol initialization (programs cannot read the cluster's own, so the binding is to that configured value, stored immutably), the table *incarnation* as *instance*, the hand id as *session* and the phase or round as *step*. A table address can be closed and re-created, and its hand ids then restart at 1, so a hand is identified by the triple (table, incarnation, hand id): every table takes the next value of a global counter kept by the protocol program when it is created, `HandStarted` carries it, and *instance* = $\mathsf{H}(\texttt{memepoker/instance/v1} \,\|\, \mathit{table} \,\|\, \mathit{incarnation})$. Each proof then appends its full statement before the first challenge. A proof is therefore valid for exactly one statement, in one hand, at one incarnation of one table, in one program, on one



One byte format for the Rust host (sha2), the WebAssembly client and the program (sol_sha256); only the hash backend differs. Change any tag field and the challenge is unrelated: a proof replayed at another table, hand, phase, program or cluster fails.

Figure 2: The transcript format of `pcrypto_transcript`. Segment 0 opens with the domain tag; each challenge closes a segment and hashes it to the 32-byte state d, from which challenge scalars are drawn by a wide reduction; the next segment chains d.

cluster. Replays across hands, tables, re-created tables, proof kinds, programs, or between a local fork, devnet and mainnet all fail, and a verifier asked for (table, incarnation, hand) never accepts another incarnation's hand.

Intuition. The tag acts as the proof's postal address. Change any line of the address, whether the table, the hand, the cluster or the kind of proof, and the challenge comes out as an unrelated random number, so the old response no longer verifies.

3.2 Schnorr proof of knowledge (key commitment)

To register pk_i, seat i proves knowledge of sk_i [14]. It samples a nonce k, sets $R = kG$, appends pk_i and R to the transcript (label `"pcrypto/schnorr/v1"`, with the holder index, the position of the seat in the deal order, bound), derives c, and outputs (c, z) with $z = k + c \cdot sk_i$: 64 bytes. The verifier recomputes $R' = zG - c\,pk_i$ and accepts iff the recomputed challenge equals c. It also rejects $pk_i = \mathcal{O}$. Special soundness gives an extractor $(sk_i = (z - z')/(c - c')$ from two accepting transcripts with the same R). Under Fiat–Shamir with rewinding this rules out *rogue keys*: a seat cannot choose $pk_i = X - \sum_{j \neq i} pk_j$ for a target X without knowing its discrete log.

Hedged nonces. All sigma-protocol nonces are *hedged*. k is derived from SHA-256 over a label, a challenge drawn from the statement transcript, the secret key and 64 fresh random bytes, then widened and reduced. A broken or repeating RNG therefore cannot reuse a nonce across different statements, which would otherwise reveal sk_i from two proofs. Secrets and nonce material are zeroized after use.

3.3 Chaum–Pedersen proof of share correctness

A decryption share D_i for ciphertext (C_1, C_2) comes with a proof that $\log_G pk_i = \log_{C_1} D_i$ [15]. The prover samples k, forms $A_1 = kG$ and $A_2 = kC_1$, binds $(pk_i, C_1, D_i, A_1, A_2)$ plus the card slot and the current deck commitment into the transcript (label `"pcrypto/dleq/v1"`), and outputs $(c, z = k + c\,sk_i)$, again 64 bytes. The verifier computes

$$A_1' = zG - c\,pk_i, \qquad A_2' = zC_1 - c\,D_i,$$

as two two-point MSMs and checks the challenge. It rejects $\mathcal{O}$ for pk_i, C_1 and D_i: a zero key, a trivial ciphertext and a zero share are never legitimate. The protocol is special sound

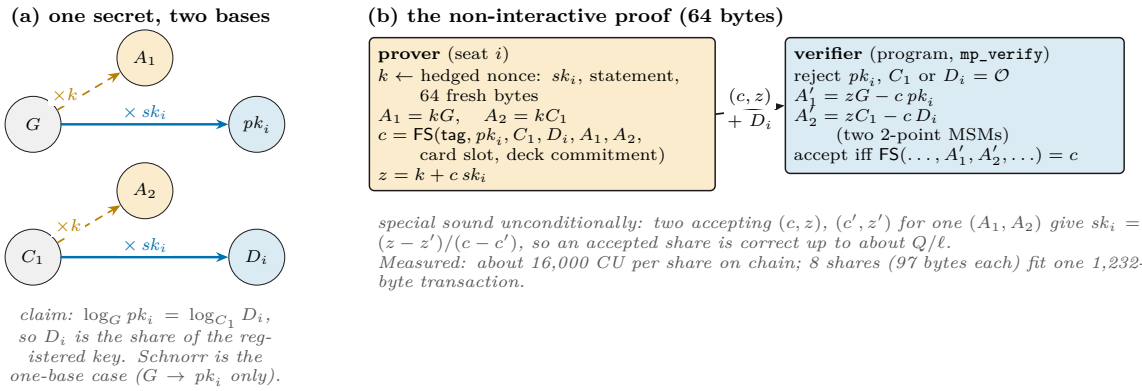


Figure 3: The Chaum–Pedersen (DLEQ) proof of a decryption share. (a) The same secret sk_i maps G to pk_i and C_1 to D_i; the prover commits with one nonce k on both bases. (b) The 64-byte non-interactive proof and its check, with the card slot and the deck commitment bound into the challenge. Per-share cost from Table 8.

unconditionally (two transcripts yield a witness for both logarithms), so an accepted share is correct except with probability about Q/ℓ for an adversary making Q random-oracle queries. Figure 3 shows the proof.

3.4 The Bayer–Groth shuffle argument

Statement. Public: PK, an input deck $C_0, \ldots, C_{N-1}$ and an output deck $C'_0, \ldots, C'_{N-1}$. Witness: a permutation π of $\{0, \ldots, N-1\}$ and scalars $\rho_0, \ldots, \rho_{N-1}$ such that

$$C'_i = C_{\pi(i)} + \mathsf{Enc}_{PK}(\mathcal{O}; \rho_i) \qquad \text{for all } i.$$

We implement the argument of Bayer and Groth [3] without its optional multi-exponentiation reduction. $N = m \cdot n$ is arranged as m columns of length n; for $N = 52$ the shape that minimizes proof size is $m = 4$, $n = 13$.

Protocol. The prover and verifier run the following, with every message appended to the transcript before the next challenge.

1. The prover commits column-wise to $a_i = \pi(i) + 1$, giving $c_A \in \mathbb{G}^m$. Challenge x.
2. The prover commits to $b_i = x^{a_i}$, giving $c_B \in \mathbb{G}^m$. Challenges y, z.
3. *Product argument.* Let $f_i = y\, a_i + b_i - z$. Its commitments are computable by the verifier as $y\, c_A + c_B - \mathsf{com}(z, \ldots, z; 0)$. The prover shows

$$\prod_{i=1}^{N} f_i = t := \prod_{k=1}^{N} (y\, k + x^k - z),$$

   where t is computed by the verifier. For $m \geq 2$ this is a Hadamard-product argument, reduced to a zero argument under the bilinear map $\langle a, b \rangle_y = \sum_l a_l b_l y^l$, followed by a single-value product argument on the column products. For $m = 1$ only the latter is needed.
4. *Multi-exponentiation argument.* The prover shows

$$\sum_{j=0}^{N-1} x^{j+1} C_j = \mathsf{Enc}_{PK}(\mathcal{O}; \rho) + \sum_{i=0}^{N-1} b_i\, C'_i$$

for the b committed in c_B and some ρ it knows.

Lemma 3.1 (Permutation check)**.** *If the product identity in step 3 holds as polynomials in Z, then the multisets $\{y\,a_i + b_i\}$ and $\{y\,k + x^k\}$ are equal. If moreover the openings are fixed before y is chosen, then except with probability at most N^2/ℓ over y the pairs $\{(a_i, b_i)\}$ equal $\{(k, x^k)\}$, so a is a permutation of $1, \ldots, N$ and $b_i = x^{a_i}$.*

Proof sketch. Both sides are monic degree-N polynomials in Z, so equality of products at a random z implies, by Schwartz–Zippel [16, 17] (error N/ℓ), equality of their root multisets. A collision $y\,a_i + b_i = y\,k + x^k$ with $(a_i, b_i) \neq (k, x^k)$ fixes y to one value per pair, at most N^2 values in total. □

With $b_i = x^{\pi(i)+1}$, the right-hand side of step 4 equals $\sum_j x^{j+1} C_j$ plus an encryption of $\mathcal{O}$ exactly when every C'_i re-encrypts $C_{\pi(i)}$, except with probability N/ℓ over x. Together:

Theorem 3.2 (Bayer–Groth [3])**.** *The interactive argument is perfectly complete, perfect special honest-verifier zero-knowledge, and computationally special sound under DL in $\mathbb{G}$ (via the binding of* com*). Its Fiat–Shamir transform with the transcript of Section 3.1 is a non-interactive zero-knowledge argument of knowledge of (π, ρ) in the random-oracle model. For a multi-round protocol that is special sound unconditionally, the transform loses at most a factor $Q + 1$ in knowledge error for Q oracle queries [18]. Bayer–Groth is special sound only computationally: from enough accepting transcripts one extracts either a witness or a nontrivial discrete-log relation among $H, G_1, \ldots, G_{64}$. The bound therefore applies to the relaxed relation "a witness, or such a relation", for which the argument is special sound unconditionally, and an extractor for the relaxed relation yields one for the shuffle relation except with probability $\mathsf{Adv}^{\mathrm{DL}}$. The knowledge error per challenge round is $O(N)/\ell$ (each verification equation is a polynomial of degree at most N in that round's challenge), plus the N^2/ℓ of Lemma 3.1 for y, which leaves $(Q+1)(O(N) + N^2)/\ell + \mathsf{Adv}^{\mathrm{DL}} \approx Q \cdot 2^{-241} + \mathsf{Adv}^{\mathrm{DL}}$.*

Proof format. All points are 32-byte canonical Ristretto encodings and all scalars 32-byte canonical. For $m \geq 2$ a proof is $(11m + 2)$ points and $(5n + 9)$ scalars:

$$c_A[m]\;\; c_B[m]\;\; c_{\hat{f}}\;\; c_{hb}[m-2]\;\; \mathrm{zero}\{c_{a_0}\, c_{b_{m+1}}\, c_d[2m]\, a[n]\, b[n]\, r\, s\, t\}$$
$$\mathrm{svp}\{c_d\, c_{lo}\, c_{hi}\, a[n]\, b[n]\, r\, s\}\;\; \mathrm{mexp}\{c_{a_0}\, c_b[2m-1]\, E[2m-1]\, a[n]\, r\, b\, s\, \tau\}.$$

For $N = 52$ that is 46 points and 74 scalars, 3,840 bytes. The output deck is $52 \times 64 = 3{,}328$ bytes.

Verification as independent identity checks. The verifier in `pcrypto_shuffle::verifier` is arranged for a runtime that cannot hold the whole computation at once. A *begin* step does all structural work: lengths and shape, $PK \neq \mathcal{O}$, no trivial ciphertext in the output deck, canonical scalars, the transcript over both decks and the proof, all eight challenges, and the product target t. It stores them in a scratch of nine scalars (290 bytes). Each remaining check expands every derived commitment linearly into one multiscalar multiplication (MSM) that must equal $\mathcal{O}$. Table 2 lists them. The checks share no point state, so they can run in any order and any grouping across transactions. A *finish* step accepts only when every applicable check has passed. Batching each equation into a single MSM with the derived commitments inlined avoids materializing intermediate points, which on Solana would mean extra syscalls and heap. Figure 4 shows the schedule and the cost of each part.

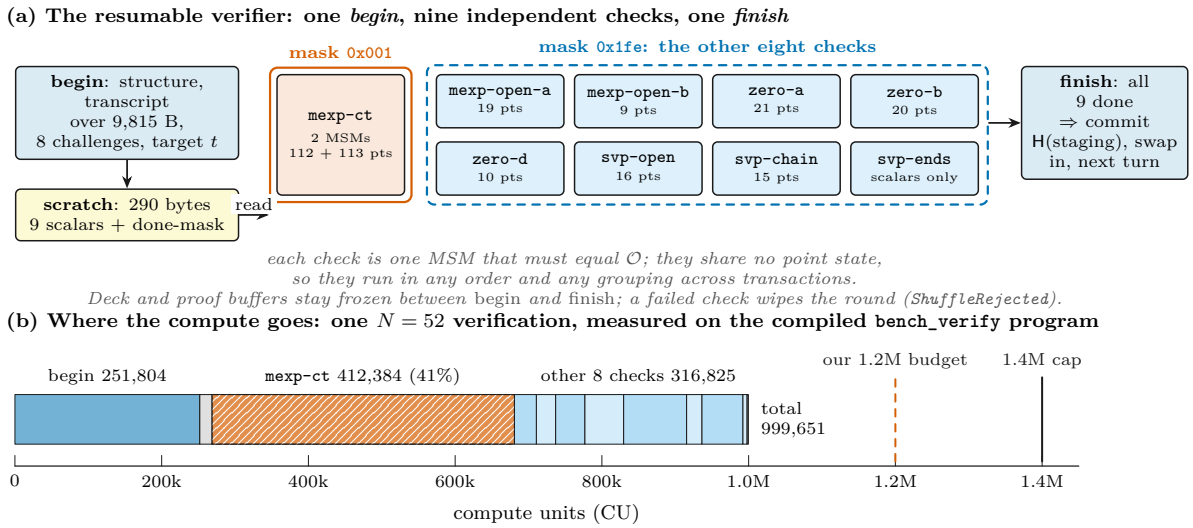


Figure 4: The resumable Bayer–Groth verifier. (a) *Begin* derives every challenge into a 290-byte scratch; the nine checks (MSM sizes at $N = 52$ from Table 2) are independent and run in two masks; *finish* commits the deck. (b) Compute per part of one verification on the compiled `bench_verify` program (marginal CU, data from Table 5, `docs/G_SHUFFLE.md`).

Table 2: The nine verification checks for shape (m, n), and their MSM sizes at $N = 52$ ($m = 4$, $n = 13$).

Check	Equation (each MSM must equal $\mathcal{O}$)	points at $N = 52$
`mexp-ct`	$\sum_j x^{j+1} C_j - \sum_i b_i C'_i - \ldots$, two MSMs ($C_1$, C_2 side)	$112 + 113$
`mexp-open-a`	opening of c_A in the multi-exponentiation argument, $m + n + 2$	19
`mexp-open-b`	opening of c_B side, $2m + 1$	9
`zero-a`	zero argument, A side, $2m + n$	21
`zero-b`	zero argument, B side, $m + 3 + n$	20
`zero-d`	zero argument, D side, $2m + 2$	10
`svp-open`	single-value product, opening, $n + 3$	16
`svp-chain`	single-value product, product chain, $n + 2$	15
`svp-ends`	scalar end conditions only	0

A second, independent verifier. `pcrypto_shuffle::second_line` verifies the same argument in the textbook arrangement. It computes every derived commitment (each c_{F_j}, the zero-argument statement commitments, the input combination) explicitly as a group element and compares separately computed left and right sides. The two verifiers share only the transcript and parsing code. Differential tests run both on every honest and every tampered proof, and the second line backs the re-verification fraud proofs of Section 7. The Schnorr and Chaum–Pedersen proofs have second-line verifiers too, but there they are not independent checks in the same sense. In the compact (c, z) form the only soundness check is the challenge equation, so the second line checks the same equation and differs only in its backend calls: it decodes the points itself and forms each commitment from single-scalar multiplications instead of an MSM. It catches an implementation fault in one arithmetic path, not a flaw in the equation.

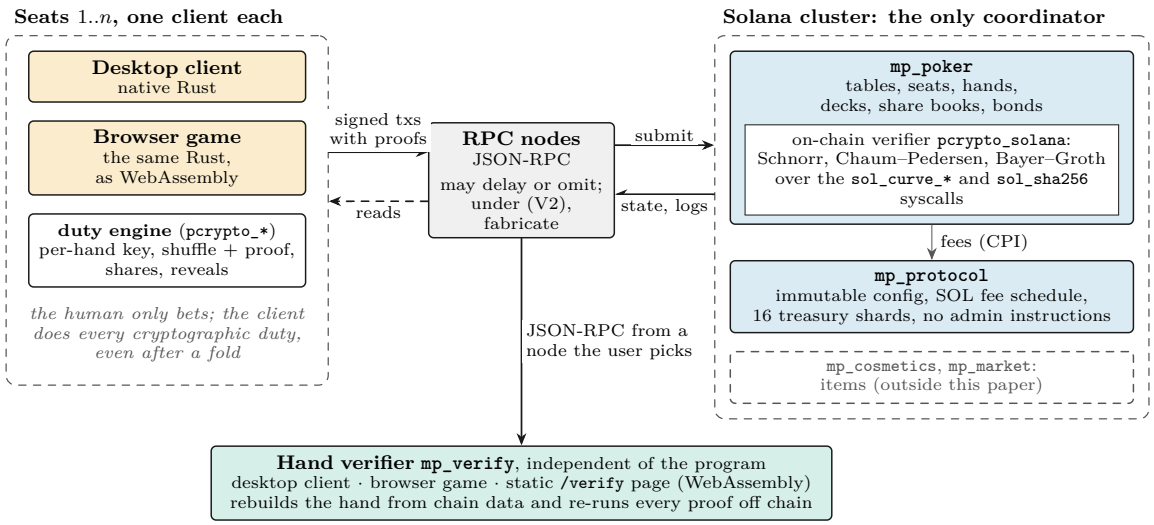


Figure 5: System architecture. Each seat's client performs every cryptographic duty and submits signed transactions with proofs; `mp_poker` verifies them on chain through the curve and hash syscalls. The independent hand verifier rebuilds a finished hand from JSON-RPC data alone and trusts the node only to return authentic data (V1, Section 9.3).

4 System and Threat Model

Parties and programs. A table has up to nine seats. Each seat is a player whose client (a native desktop application or the same Rust code compiled to WebAssembly in a browser) performs every cryptographic duty automatically, including after the player folds; the human only makes betting decisions. Four Solana programs make up the system: `mp_protocol` holds the immutable protocol configuration, the SOL fee schedule and the treasury, and has no admin instructions; `mp_poker` holds tables, seats, hands, decks, share books and bonds; `mp_cosmetics` and `mp_market` handle items and are outside this paper's scope. There is no server, no dealer and no oracle. The chain is the only coordinator, as Figure 5 shows.

The chain as the single sequencer. Every move is a Solana instruction, and the chain orders them. Two conflicting moves for the same step cannot both be accepted: the first advances the hand's `action_seq`, `shuffle_turn` or share bitmap, and the second fails. Equivocation on chain is impossible by construction, which is why the protocol accepts no off-chain signed protocol messages at all. We assume the chain is live and safe: transactions are eventually included, the clock sysvar is monotone up to small drift, and confirmed state is final. We make no assumption about *when* a given transaction lands inside its window.

Adversary. The adversary statically corrupts any subset of the seats and controls their clients entirely. It may deviate arbitrarily, withhold messages, submit malformed or adversarial transactions, front-run, and observe every on-chain byte and every pending transaction. It may also control the RPC nodes that honest clients use for reads. Clients check that slot and owner are consistent and do not trust returned signatures (Section 11), but a lying RPC can still delay an honest client. Corrupt seats may collude among themselves out of band. We call a seat *honest* if its client follows the protocol and its secrets never leave it. Figure 6 summarizes the model.

Security goals. Informally, for every coalition $\mathcal{A}$ of corrupted seats:

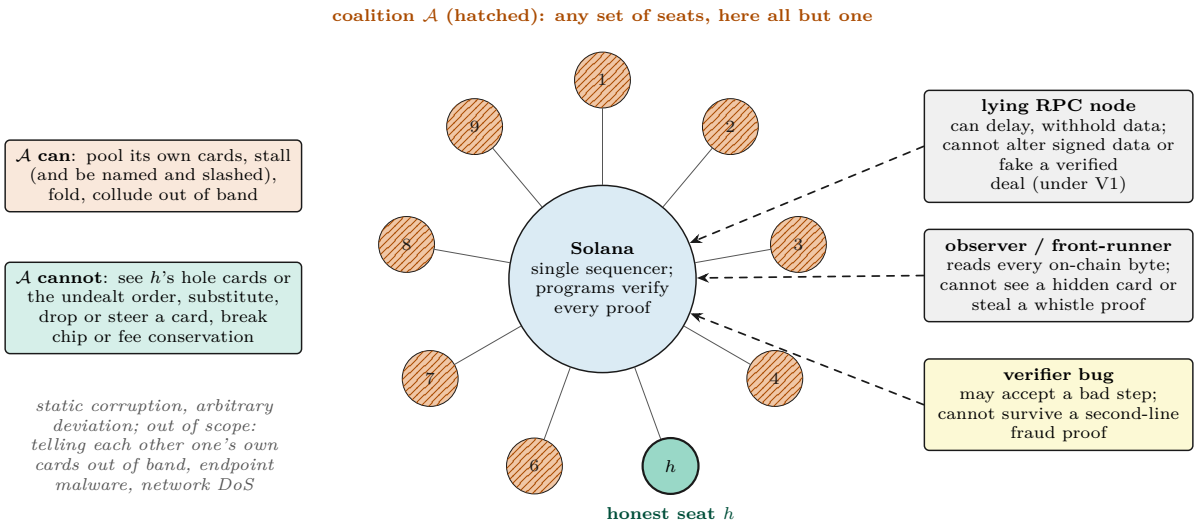


Figure 6: The adversary model. A static coalition $\mathcal{A}$ of seats (hatched), here every seat but the honest one h, deviates arbitrarily; observers, front-runners and lying RPC nodes act from outside; the chain orders every move. Table 6 lists the guarantees per adversary.

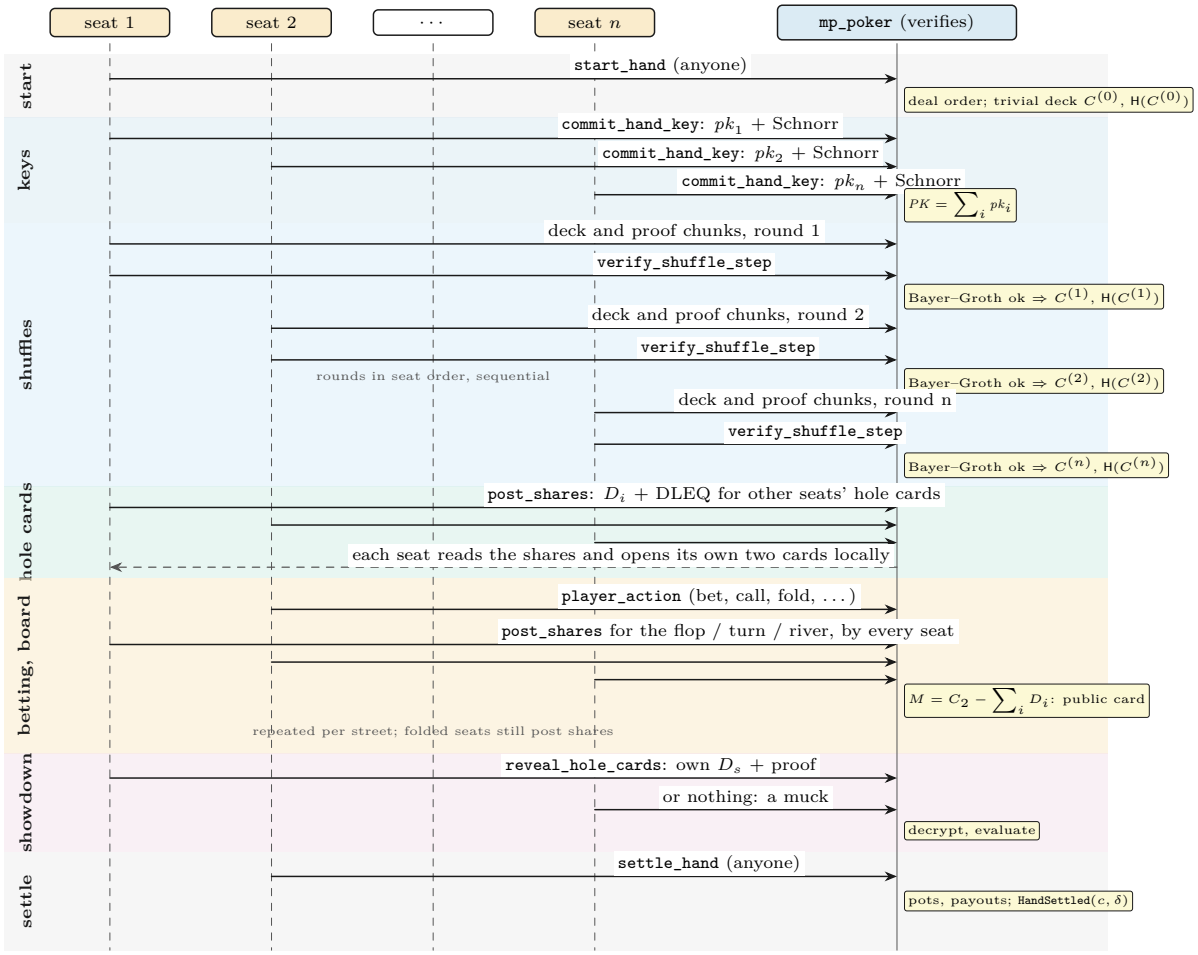
1. *Deck privacy.* If $\mathcal{A}$ excludes at least one honest seat, it learns nothing about the order of the undealt cards.
2. *Hole-card privacy.* $\mathcal{A}$ learns nothing about an honest seat's hole cards unless that seat reveals them at showdown, even when $\mathcal{A}$ is every other seat.
3. *Integrity.* The cards opened in a hand are the cards of one permutation of a standard 52-card deck. No coalition, including all seats but one, can substitute, duplicate, drop or steer a card.
4. *Accountability and liveness.* Every pending obligation has a deadline. After it passes, anyone can move the hand forward, the obligated seat is identified, and a seat that stalls after the deal is named and pays a stall penalty to the seats still contesting the pot (the limits of that guarantee are stated with the last-revealer advantage in the limitations).
5. *Conservation.* Chips are conserved per table; fees are SOL only and equal the published schedule; bonds are SOL only and never mix with chips.

Out of scope. No protocol can stop players from telling each other their own cards over a side channel; Section 7 makes one form of it (sharing keys) self-defeating, and Section 13 states the residue. Endpoint compromise (malware reading a client's memory) and network-level denial of service are also out of scope.

5 The Hand Protocol

A hand among n participants (the seats dealt in) runs the following phases. Each message is a transaction whose proof the program verifies before changing state; each phase has a deadline (Section 7). Figure 7 shows the transactions of one hand.

1. **KeyCommit.** Every participant i samples $sk_i \xleftarrow{\$} \mathbb{Z}_\ell$ from the operating system's CSPRNG (never derived from the wallet key, seed phrase or session key, and zeroized when the hand ends) and posts pk_i with a Schnorr proof under purpose `keycommit`. The



Every arrow is a Solana transaction whose proof the program checks before it changes state. Each obligation has a deadline: `crank_timeout`*, or any move's prologue, resolves a lapse (restart before the deal, abort with staller-forfeits after it).*

Figure 7: The hand protocol as a sequence of Solana transactions, from `start_hand` to settlement. Shuffle rounds run in seat order; each is a set of chunk uploads and verifier steps (8 transactions in the benchmark profile, 14 in the deployed Profile B). Shares for hole cards and board cards come from every seat, folded or not.

program verifies each proof and accumulates $PK = \sum_i pk_i$ on chain. It refuses any key that would make a partial sum $\mathcal{O}$, not only the completing one: the seat submitting it, which provably knows the discrete log of the sum of the earlier keys, is blamed and its slot rolled back.

2. **Initial deck.** The program writes the canonical trivial deck $(\mathcal{O}, M_j)_{j=0}^{51}$ and its commitment $\mathsf{H}(\text{deck})$.

3. **Shuffle rounds.** In seat order starting left of the button, participant i reads the current deck C, samples π_i and $\rho_{i,0..51}$, computes $C'_k = C_{\pi_i(k)} + \mathsf{Enc}_{PK}(\mathcal{O}; \rho_{i,k})$, uploads C' and a Bayer–Groth proof in chunks, and the program runs the resumable verifier (Section 6). Only after *finish* succeeds does the program swap the staged deck into the current deck, store its commitment and advance the shuffle turn. The final order is $\pi_1 \circ \cdots \circ \pi_n$ and is unknown unless every participant colludes (Figure 8).

4. **Deal positions.** Positions are fixed as in a live deal: with participants $p = 0, \ldots, n-1$ in deal order, hole card $r \in \{0, 1\}$ of position p is deck index $r\,n + p$; burns are $2n$, $2n + 4$,

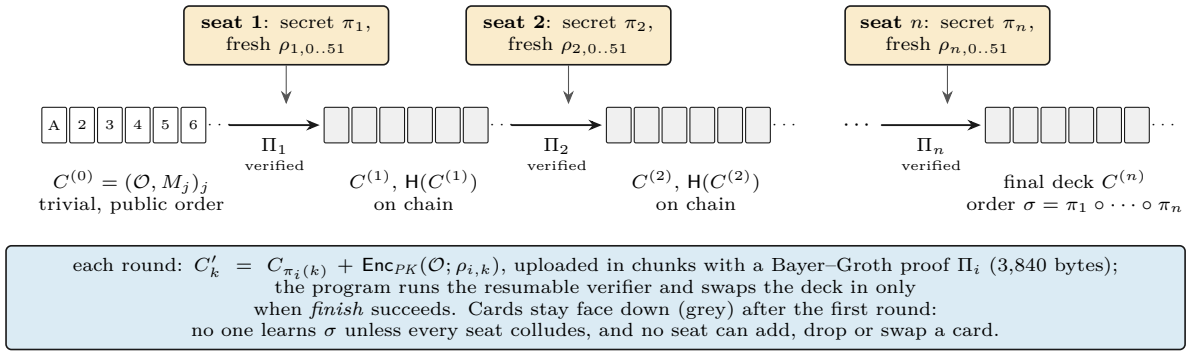


Figure 8: The shuffle chain. Starting from the public trivial deck, each seat permutes and re-encrypts every card with fresh randomness and proves it with a Bayer–Groth argument; the program verifies each round before it becomes the next round's input and stores its commitment.

$2n + 6$; the flop is $2n + 1..2n + 3$, the turn $2n + 5$, the river $2n + 7$. Burns are never decrypted. The highest index used is $25 < 52$; Figure 9 lays the positions out.

5. **Hole cards.** For each hole card of seat s, every participant *other than* s posts $D_i = sk_i C_1$ with a Chaum–Pedersen proof (purpose `share`, binding the card slot and the deck commitment). The program verifies and stores each share. Seat s reads them, computes its own D_s locally, and recovers $M = C_2 - \sum_i D_i$, which it looks up among the 52 card points.

6. **Board.** When a betting round closes, every participant, folded or not, posts shares for the next board cards. The program verifies them, computes $M = C_2 - \sum_i D_i$ itself and records the public card.

7. **Showdown.** In the showdown order, each remaining player who wants to show posts its own D_s for both hole cards with proofs (purpose `reveal`). The program decrypts, evaluates the hand on chain, and records it. Not revealing before the deadline is a muck: the player gives up its claim, and its cards are never revealed. If action closed with a player all-in, hands are tabled before the run-out.

8. **Settle.** The program builds the pots, applies the odd-chip rule and pays out. Keys are per hand, so nothing from this hand helps decrypt a later one.

Intuition. Think of each shuffle as a player taking the deck behind their back, shuffling it, and putting a fresh lock of their own on every card, then proving to the table, without showing the cards, that the same 52 cards came back. A card opens only when every lock-holder removes their lock. For your hole cards everyone else removes theirs in public, and the last lock is yours.

Why hole cards stay private on chain. After the other seats post their shares, anyone can compute $C_2 - \sum_{i \neq s} D_i = M + sk_s C_1$. Since $C_1 = rG$, this is $(C_1, M + r\, pk_s)$: an ElGamal encryption of M under pk_s alone. Its randomness r is the sum of all re-encryption scalars applied to that card, which includes seat s's own fresh $\rho_{s,\cdot}$. Without sk_s the card stays hidden under DDH, even from every other seat combined. This is why the shares can be posted in the clear and verified on chain (Figure 10).

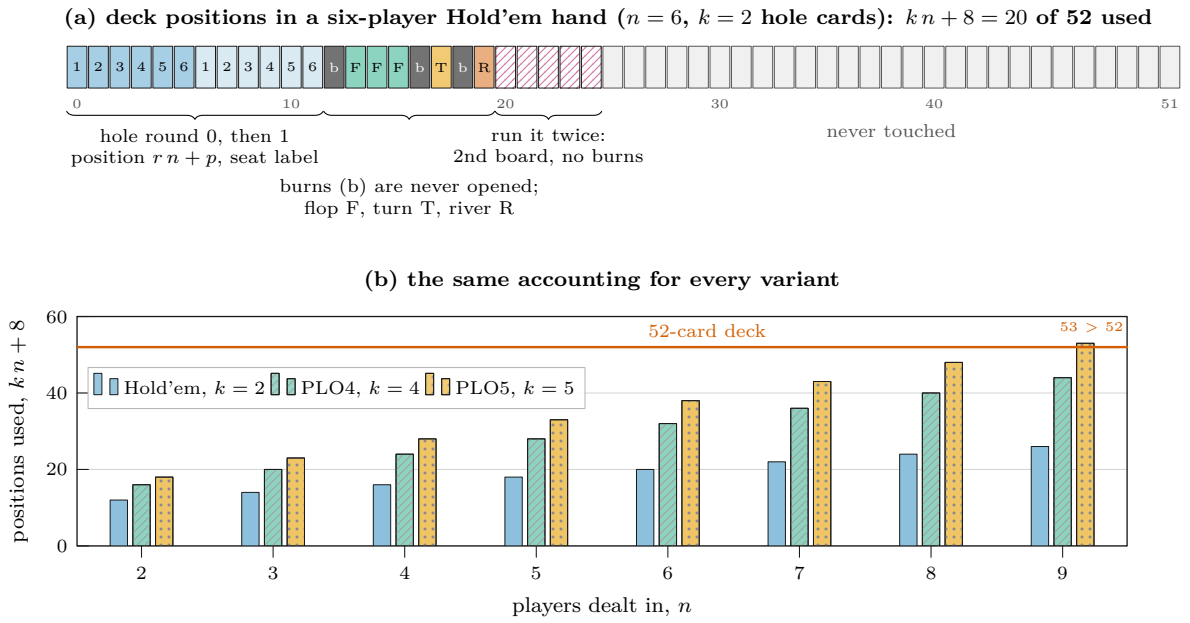


Figure 9: Deal positions. (a) A six-player Hold'em hand uses $2n + 8 = 20$ of the 52 positions; a second board for run it twice takes the next ones with no burns. (b) The same $k\,n + 8$ accounting (`mp_poker_core::deal`) for two-, four- and five-card hands: a five-card deal for nine seats would need 53 positions.

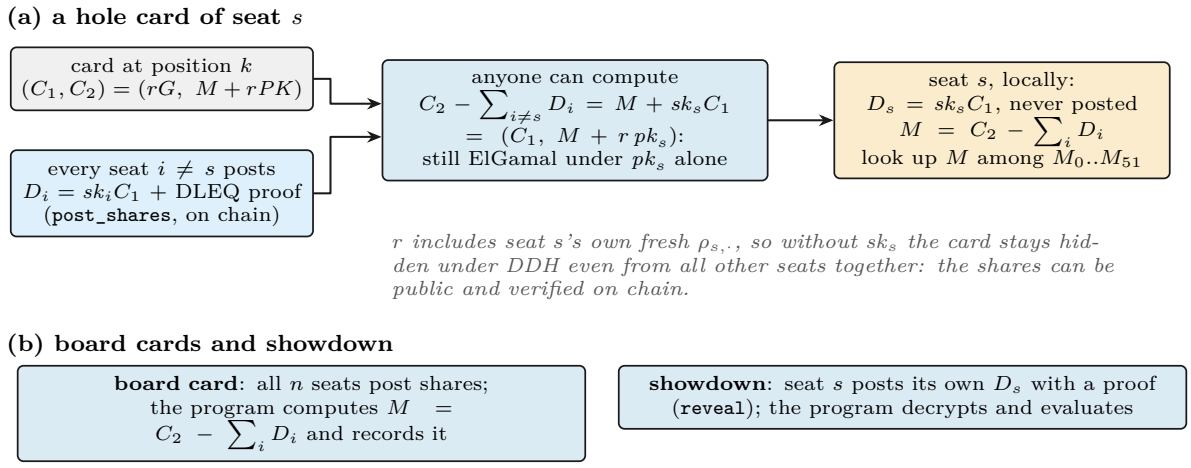


Figure 10: Threshold decryption. (a) For a hole card the other seats post verified shares in public; what remains is an ElGamal ciphertext under the owner's key alone, which only the owner can open. (b) Board cards are opened by the program from all shares, and a shown hand by the owner's own share.

6 Verifying Inside Solana Programs

6.1 The constraints

Table 3 lists the Solana limits that shaped the design. Two facts dominate. A 52-card deck (3,328 bytes) and its proof (3,840 bytes) together are almost six times the size of a transaction. And the work of one verification, about 260 validated point decodes, nine MSMs over 335 points and 325 scalar multiplications, would cost tens of millions of CU if done in software on the SBF

Table 3: The Solana limits that drive the design (agave 2.2 cost tables).

Limit	Value
Compute per transaction	1.4M CU (200k per instruction unless raised)
Our per-step budget	1.2M CU, leaving 200k headroom
Transaction size	1,232 bytes, signatures and account keys included
Heap	32 KiB bump allocator (no free); 4 KiB stack frames
Curve syscalls (Edwards, Ristretto)	`sol_curve_{validate_point, group_op, multiscalar_mul}`
Hash syscall	`sol_sha256` over a list of slices

Table 4: Scalar arithmetic on SBF, CU per operation (200-operation loops on the compiled program).

Operation	dalek `Scalar52`	`lean`
multiplication	5,711	**1,473**
canonical decode (+1 add)	3,862	**524**
wide 64-byte reduction (+1 add)	6,643	**1,440**
addition	473	98
subtraction / negation as $0 - a$	364	99
unary negation (dalek, reduces)	2,922	not used

virtual machine. The design therefore rests on four decisions: use only group operations that have native syscalls, make scalar arithmetic cheap, hash each Fiat–Shamir segment with one syscall, and make the verifier resumable so the work can be spread over transactions if necessary.

6.2 A syscall-backed group

The verifiers in `pcrypto_*` are generic over a `Group` trait. `pcrypto_solana::Sol` implements it on the curve syscalls: `decode` is `sol_curve_validate_point` on the Ristretto encoding, `add`, `sub` and `mul` are `sol_curve_group_op`, and `msm` is `sol_curve_multiscalar_mul`. Points never leave their 32-byte encoded form on chain. Pinned constants (the 52 card points, G, and the 65 commitment generators) are copied as bytes with no validation, which saves 73 validations per shuffle verification. On the host the same crate falls back to `curve25519-dalek`, so host tests run exactly the program's code except for the syscall itself.

Scalar arithmetic. The syscalls cover points only. Scalar work (challenge powers, the product target, the coefficient vectors of every MSM) runs on SBF itself, where the default 52-bit-limb scalar type is expensive: the SBF virtual machine has no 128-bit multiply, so each 52-bit limb product is emulated. `pcrypto_elgamal::lean` reimplements the hot paths with 32-bit limbs, a special-form reduction modulo ℓ, and no pack/unpack round trips. The verifier routes every scalar operation through `opcount::{smul, sadd, ssub, sneg}`, which select `lean` when compiled for Solana. Table 4 gives the measured effect.

One hash per challenge. SHA-256 in software on SBF is expensive, and a transcript segment can be several kilobytes (the *begin* step hashes both decks and the proof, 9,815 bytes in all). `SolSha256` implements the transcript's `Hasher` by buffering absorbed bytes in geometrically growing chunks of at most 1 KiB (no reallocation copies on a heap that never frees) and calling `sol_sha256` once over the chunk list at `finalize`. A challenge over a 1,024-byte segment costs 7,229 CU this way against 151,576 CU for the software hash. Over 7,168 bytes it costs 11,143 CU.

6.3 Chunked uploads and the resumable verifier

A shuffle round in `mp_poker` is driven by three instructions over per-table accounts created once by `create_table` at canonical program-derived addresses: a `DeckState` with `current` and `staging` buffers and a written-chunk bitmap, a `ProofBuffer`, and a 290-byte `VerifierScratch`.

- `upload_deck_chunk` and `upload_proof_chunk` accept writes only from the seat whose shuffle turn it is. Chunks are write-once per round and independent of each other, so every chunk of a round can land in the same slot. Every ciphertext component is validated as a canonical point when written.
- `verify_shuffle_step(mask)` advances the scratch. The first call runs *begin*; each call runs the checks named in its bitmask; when all nine are done, *finish* commits H(staging) as the deck commitment, swaps staging into current and advances the turn. Anyone may call it, which helps when the shuffler disconnects right after uploading. A failed check wipes the round (`ShuffleRejected`).
- Between *begin* and *finish* the deck and proof buffers are frozen: the scratch is valid only for the bytes it was derived from, and *finish* trusts the scratch's done-mask. All three accounts must be at their canonical addresses for this table, not merely owned by the program with the right discriminator (Section 11, RT-PG-VS-003).

The canonical schedule for $N = 52$ is two masks, `0x001` (the ciphertext MSM) and `0x1fe` (the other eight checks). When they fit, both run in the same transaction as *begin* and *finish*.

Byte budget. A transaction's fixed overhead is its signatures (65 bytes for one signer), header and blockhash, account keys, a compute-budget instruction, and our instruction header and arguments. With a single signer and the buffer account only, the largest write payload is 1,013 bytes, which is 15 ciphertexts per chunk. With the eight additional accounts `mp_poker` needs, it is 749 bytes inline (Profile B, legacy transactions) or 992 bytes when the accounts are referenced through a per-table Address Lookup Table (Profile A, v0 transactions). Profile A therefore keeps the eight-transaction round. Figure 11 shows the per-table accounts.

Each `post_shares` carries up to k shares of 97 bytes each (card slot, share, proof); $k = 8$ is the largest that fits in one 1,232-byte transaction.

6.4 Compute accounting

Table 5 gives the measured cost of each verifier step on the compiled program; Section 10 describes the harness. The ciphertext MSM is 43% of the total, and *begin*, dominated by transcript hashing, challenge derivation and the product target $t = \prod_k (yk + x^k - z)$, is another quarter. Everything together is 999,651 CU, so the whole verification fits in one transaction under our 1.2M budget. The round's last transaction carries the final 130 bytes of the upload, then *begin*, both masks and *finish*, for 1,016,627 CU.

7 Enforcement: Timeouts, Bonds and Proof Classes

Cryptography makes cheating moves unacceptable. It does not make players cooperate. A player who simply stops can freeze a mental-poker hand, so the enforcement layer has to make stalling identifiable, resolvable by anyone, and costly.

7.1 Validity first

Every claim a player can make through the chain falls into one of two classes. In class **V** (validity at submission), the submitter supplies a proof, or the program computes the result itself, and the

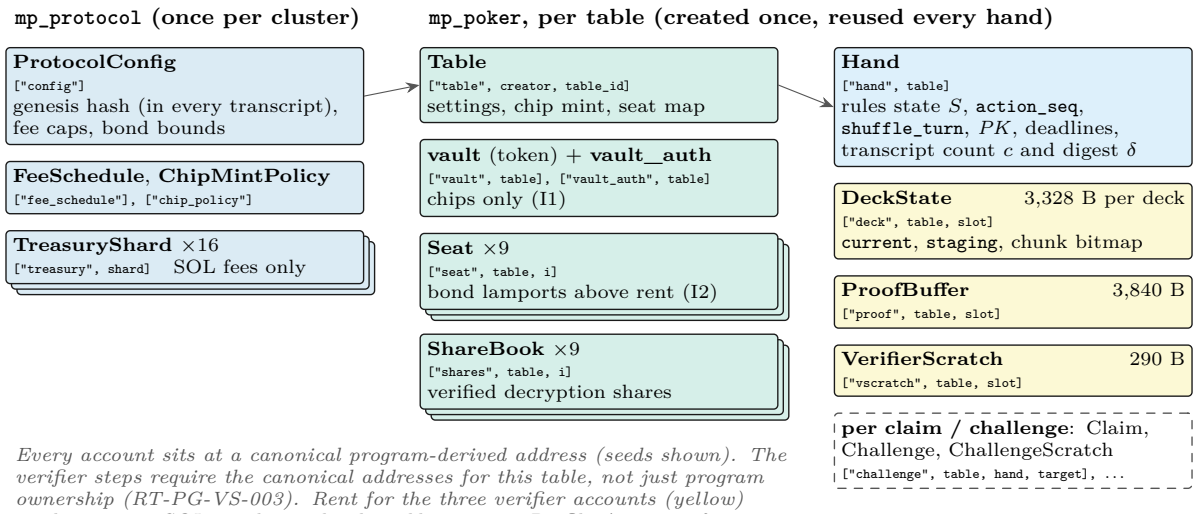


Figure 11: Account layout. Protocol accounts exist once per cluster; each table owns its seats, share books, hand account and the three verifier accounts, all at canonical program-derived addresses (seeds in brackets, from `mp_poker::seeds`). Rent for the verifier accounts is from the *Rent* paragraph of Section 10.

Table 5: Bayer–Groth verification, $N = 52$ ($m = 4$, $n = 13$), each step in its own transaction on the compiled `bench_verify` program. "Marginal" subtracts an empty run (proof parsing and scratch I/O).

Step	CU	marginal CU	heap bytes
begin (structure, transcript, 8 challenges, target t)	251,804		22,790
run with no checks (parse proof + scratch)	16,592		7,038
`mexp-ct`	428,976	412,384	25,536
`mexp-open-a`	46,431	29,839	8,940
`mexp-open-b`	42,865	26,273	8,096
`zero-a`	56,317	39,725	9,132
`zero-b`	69,532	52,940	9,260
`zero-d`	102,435	85,843	8,096
`svp-open`	37,326	20,734	8,652
`svp-chain`	72,462	55,870	8,491
`svp-ends`	22,193	5,601	7,038
finish	2,046		
sum (begin + one run + finish)	**999,651**		

program verifies it before accepting it: a cheat cannot even be recorded. In class **O** (optimistic), the claim is recorded with a SOL claim bond and opens a challenge window during which anyone may refute it. The default is that *everything is V*: key commitments, shuffles, chunk integrity, point validity, decryption shares, board values, showdown reveals, hand evaluation, action legality, pot amounts, session-key scope and timeouts. V has no window in which a false claim stands, needs no watcher, adds no latency, and does not need bonds sized against pot value. That last point matters because pots are in meme tokens that have no on-chain price, while bonds are in SOL.

Class O remains available for one claim, shuffle correctness (`claim_shuffle`, `open_challenge`, `challenge_verify_step`, `finalize_claim`). It is selected per deployment by a fixed proof-kind registry, and it is allowed only for claims that finalize before any chips move, so a false claim can at worst cause a restart.

7.2 Piggybacked timeouts under a virtual clock

Every pending obligation (a bet, a key commitment, a shuffle, a set of shares, a reveal) has an effective deadline $\mathit{eff} = \mathit{start} + \mathit{base} + \mathit{bank} + \mathit{grace}$. A pure function in `mp_poker_core::timing`, shared by the program and the client SDK, resolves lapses:

1. Take the single next obligation in protocol order. If there is none, or $\mathit{now} < \mathit{eff}$, stop.
2. Apply its lapse at $t_{\mathit{lapse}} = \mathit{eff}$: check if legal or else fold for a bet; a missed blind for a forced bet; restart the hand for a key-commit or shuffle stall; abort with staller-forfeits for a share stall after the deal; muck for a reveal.
3. Start the *next* obligation at t_{lapse}, not at *now*, and repeat (up to a bound per instruction).

Every move instruction runs this as a prologue ("piggybacking"), and `crank_timeout` runs it alone for anyone. The caller passes the `action_seq` it expects *after* the lapses its client predicts with the same function. If another transaction landed first, the sequence check fails, or, under the `ApplyLapsesOnly` policy, whatever is still due is applied and the move reports `MoveSuperseded`. Either way there are no double effects.

Proposition 7.1 (Race freedom and path independence)**.** *(i) At any time now, exactly one of "seat A acts" ($\mathit{now} < \mathit{eff}(A)$) and "A lapses" ($\mathit{now} \geq \mathit{eff}(A)$) is possible, and the next seat cannot act over a live obligation of A. (ii) The state reached after resolving all lapses due by time T is the same for every sequence of crank and move transactions landing at or before T that resolve them.*

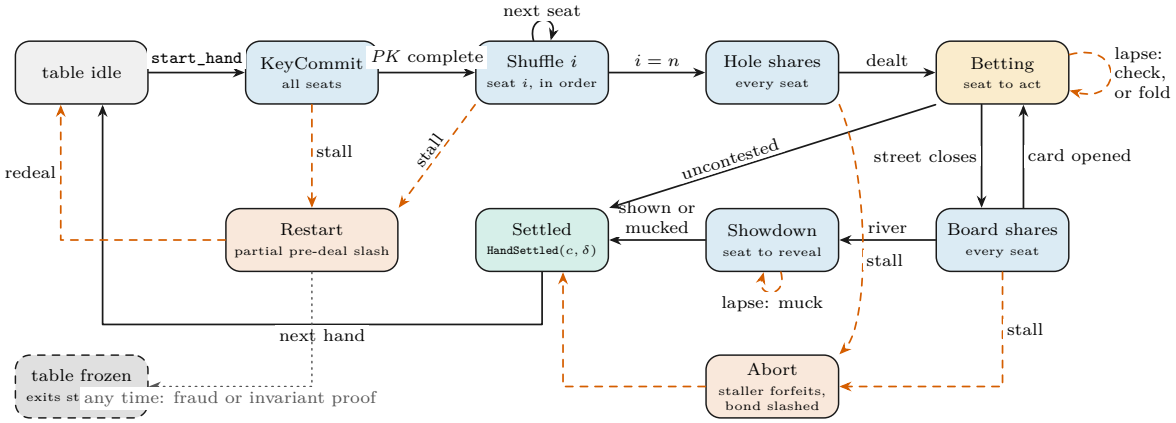
Proof sketch. (i) follows from the prologue: a move by B at $\mathit{now} < \mathit{eff}(A)$ resolves nothing and fails as not B's turn. (ii) By induction on the number of lapses: the k-th lapse is applied at its deadline eff_k, which depends only on the state after $k-1$ lapses and on the table's timing settings, never on submission time. So every resolving sequence computes the same chain $\mathit{eff}_1 \leq \mathit{eff}_2 \leq \cdots$ and the same state. A seat cannot gain time by cranking late, and cannot lose time because someone else cranked late. □

Figure 12 shows the phases, their lapses and the virtual clock.

7.3 Bonds and slashing, in SOL only

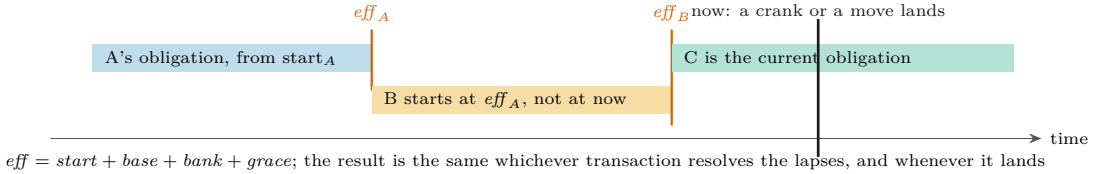
Each seat posts a *seat bond* in lamports when it joins, held in the seat account above rent. The table creator picks the amount within bounds fixed at protocol initialization. The bond backs liveness and the misbehavior that only becomes visible later. It is locked while the seat is dealt in, owes a duty, is subject to an open claim or challenge, or is inside a whistle window. A stall before the deal (key commit, shuffle) restarts the hand with a small partial slash to the other seats; no cards were seen. A stall after the deal (missing shares) aborts the hand with *staller-forfeits*: the staller's chips in the hand go to the others, and its bond is slashed. Slashed lamports pay, in order, the challenger's fee reimbursement, a challenger reward where there is one, and the harmed seats pro rata to their contribution in the hand. Bond lamports never touch the chip vault.

(a) the hand state machine: every non-terminal phase has one obligation and a deadline



solid: protocol progress; dashed: a lapse, resolved by `crank_timeout` *or any move's prologue; dotted: a proof after the fact.*

(b) the virtual clock: each lapse is applied at its deadline, never at the time someone cranks



eff = *start* + *base* + *bank* + *grace*; the result is the same whichever transaction resolves the lapses, and whenever it lands

Figure 12: (a) The hand's phases (solid) and the outcome of a lapse in each (dashed): a stall before the deal restarts the hand with a partial slash, a share stall after it aborts the hand with staller-forfeits, a betting lapse checks or folds, and a reveal lapse mucks. (b) Lapses are applied at their deadlines, so the resulting state does not depend on when, or by whom, they are resolved (Proposition 7.1).

7.4 Proofs after the fact

Three kinds of misbehavior cannot be prevented at submission. Each has a proof that can be submitted afterwards.

- **Key sharing (whistleblower proofs).** If player A hands sk_A to B so that the two can play as a team, B (or anyone it passes the key to) can submit `prove_key_leak`: a fresh Schnorr proof of knowledge of sk_A under purpose `whistle`, binding the submitter's account and seat. It is accepted only after the hand settles and within a window during which A's bond stays locked. It never changes the chip outcome; it only slashes A's bond. If a whistle could abort the hand, a losing colluder could use it to undo a losing pot; because it cannot, it is worth nothing in chips to anyone, and every partner who receives a key can betray the giver for SOL. Binding the submitter defeats front-running, and the distinct purpose stops a key-commit proof from being replayed as a whistle.

- **Verifier bugs (second-line fraud proofs).** Validity-first is only as strong as the verifier. For a window after settlement, the bytes of every accepted step stay committed by hash, and `challenge_accepted_step` lets a bonded challenger run the independent second-line verifier over them, resumably. If the second line rejects, the submitter is slashed, the hand is refunded to everyone, and the table is frozen *permanently* for new hands. No admin can unfreeze it; exits stay open. If it accepts, the challenger's bond goes to the submitter and to the players it delayed.

- **Program bugs (invariant proofs).** `prove_invariant_violation` lets anyone recom-

pute a table invariant (chip conservation, backed bonds). If it fails, the table is frozen permanently with exits open.

7.5 Fees and economic invariants

memepoker takes no rake. Every instruction instead charges a small flat fee in SOL, read from an on-chain schedule (for example 1,000 lamports for a bet, 500 for a deal step, 10^6 to create a table), with a per-action cap fixed at initialization. Fees are never paid in chip tokens. Each fee goes through one code path, which checks the client's `max_fee_lamports` guard, transfers lamports to one of 16 treasury shards, asserts that the treasury grew by exactly the fee, and emits a `FeeCharged` event. The invariants we maintain, and test (Section 11), are:

(I1) per table, vault balance $= \sum$ stacks + chips committed in the current hand;

(I2) $\sum$ bond lamports $= \sum$ posted $- \sum$ refunded $- \sum$ slashed, and every slash is paid out in full;

(F) treasury growth $= \sum$ `FeeCharged`, each equal to the schedule entry for its action; a rejected transaction moves nothing.

8 Security Analysis

We give game-based definitions and theorems with proof sketches. The analysis is in the random-oracle model, and it uses rewinding extraction for the Schnorr proofs of knowledge, which is sound here because key commitments form one sequential phase per hand. A fully formal, tight treatment in the style of a UC functionality for card games [19] is future work (Section 13).

Leakage. For a coalition $\mathcal{A}$, let $L_{\mathcal{A}}$ be the cards that $\mathcal{A}$ is entitled to see in a run: the hole cards of seats in $\mathcal{A}$, the board cards opened so far, and the hole cards revealed at showdown. Ideal dealing leaks exactly $L_{\mathcal{A}}$: from $\mathcal{A}$'s view, every other card position is a uniformly random arrangement of the remaining cards.

Definition 8.1 (Card privacy)**.** *The protocol has* card privacy *if for every PPT coalition $\mathcal{A}$ that excludes at least one honest seat there is a PPT simulator $\mathcal{S}$ that, given only $L_{\mathcal{A}}$ and the public parameters, produces a view (all on-chain data plus the coalition's internal state) computationally indistinguishable from $\mathcal{A}$'s real view.*

Theorem 8.2 (Deck and hole-card privacy)**.** *Under DDH in $\mathbb{G}$, in the random-oracle model, the memepoker hand protocol has card privacy for hands run under sequential composition. For hands and tables run concurrently, the same holds if the key-commitment proofs are straight-line extractable, which we then assume (in the algebraic group model [26], or with Fischlin's transform [25]). In particular: (a) a coalition that excludes one honest seat h learns nothing about the undealt cards; (b) it learns nothing about h's hole cards before h reveals them, even when it contains every other seat.*

Proof sketch. The simulator runs the coalition and plays h. *Hybrid 1:* extract the coalition's keys sk_i, $i \in \mathcal{A}$, from their Schnorr proofs by rewinding (sound for sequential composition only: rewinding one hand while others run concurrently can blow up, the issue of [24]; under concurrency we use the straight-line extractor assumed above), and simulate h's Schnorr proof by programming the random oracle. This is indistinguishable by HVZK, except with the negligible probability that a programmed point was already queried. *Hybrid 2:* simulate h's shuffle proof with the Bayer–Groth HVZK simulator (Theorem 3.2) and h's Chaum–Pedersen proofs with theirs, again by programming the oracle. *Hybrid 3:* replace h's output deck by fresh encryptions under PK of the correct multiset of cards, *each at a uniformly random position.* Because the

simulator knows every coalition key, an encryption under PK amounts to an encryption under pk_h, and a distinguisher between Hybrids 2 and 3 yields a distinguisher for the IND-CPA security of ElGamal under pk_h. That reduces to DDH by a standard multi-ciphertext hybrid that loses a factor N. In Hybrid 3, h's permutation is information-theoretically hidden, so the final position of each card is uniform from the coalition's view, up to the later shufflers' permutations, which the coalition knows only for its own members. *Hybrid 4 (opening).* A card in $L_{\mathcal{A}}$ is opened by h's share. The simulator, which knows the card from $L_{\mathcal{A}}$, outputs the equivocated share $D_h = C_2 - \sum_{i \neq h} sk_i C_1 - M$ with a simulated proof, instead of $sk_h C_1$. For the k shares h posts in a hand, a distinguisher between Hybrids 3 and 4 yields one for the k-instance DDH problem on $(G, pk_h, C_1^{(j)}, D_h^{(j)})_{j \leq k}$, which reduces to DDH with a loss of at most k by a standard hybrid; the proofs are simulated in both hybrids. For a card outside $L_{\mathcal{A}}$, such as h's hole card, D_h is never published, and the residual $(C_1, M + r\, pk_h)$ is an ElGamal ciphertext under pk_h alone with randomness that includes h's fresh scalar, so it stays hidden under DDH. The simulated view depends only on $L_{\mathcal{A}}$. □

Definition 8.3 (Deal integrity)**.** *A run has* deal integrity *if there is a permutation σ of $\{0, \ldots, 51\}$, fixed when the last shuffle is accepted, such that every card the program records (board, showdown) and every hole card an honest owner recovers at deck index k equals $M_{\sigma(k)}$.*

Theorem 8.4 (Integrity)**.** *Under DL in $\mathbb{G}$, in the random-oracle model, every run in which the program accepts all shuffles and shares has deal integrity, except with probability at most $(Q+1)(q_s \cdot O(N^2) + q_d)/\ell + q_s \cdot \mathsf{Adv}^{\mathrm{DL}}$ for an adversary making Q oracle queries over q_s shuffle and q_d share proofs, where $\mathsf{Adv}^{\mathrm{DL}}$ is the best advantage against DL in $\mathbb{G}$ of an algorithm running in about the extractor's time.*

Proof sketch. The initial deck is the trivial encryption of $(M_0, \ldots, M_{51})$. By knowledge soundness of each accepted shuffle (Theorem 3.2 with Lemma 3.1), there is an extractor that outputs (π_i, ρ_i) with $C_k^{(i)} = C_{\pi_i(k)}^{(i-1)} + \mathsf{Enc}_{PK}(\mathcal{O}; \rho_{i,k})$; otherwise a DL relation among the commitment generators is found. By induction the final deck encrypts $(M_{\sigma(k)})_k$ with $\sigma = \pi_1 \circ \cdots \circ \pi_n$. The deck commitment and the frozen buffers ensure that the shares are proven against exactly this deck. Each accepted Chaum–Pedersen proof binds $D_i = sk_i C_1$ for the registered pk_i (special soundness, which holds unconditionally), so $C_2 - \sum_i D_i = M_{\sigma(k)}$ for every opened card, whether the program computes it or the owner does. Because the cards are distinct points from a nothing-up-my-sleeve table, the lookup is unambiguous. □

Corollary 8.5 (No steering)**.** *A coalition that includes every seat but one cannot make the honest seat receive a chosen card, or make a chosen card appear on the board, with probability better than $1/$(number of unopened cards) plus a negligible term. Integrity fixes the opened card to $\sigma(k)$, and privacy (Theorem 8.2) makes σ's value at each unopened position uniform from the coalition's view, including at the moment it chooses its own permutations.*

The corollary holds per run of the deal. A coalition that aborts a hand after seeing some cards gets a fresh run, so selective abort is not covered here. It is priced by the stall settlement instead (last-revealer advantage, limitations).

Theorem 8.6 (Accountability)**.** *In every hand, either the hand settles, or by the time the last pending deadline passes, any party can submit a transaction that moves the hand to a restart or an abort. The resulting state names the stalling seat or seats, and in an abort after the deal the non-stalling seats receive at least their own committed chips plus the stalling seats' forfeits.*

Proof sketch. Every non-terminal phase of the hand state machine has exactly one current obligation with a finite effective deadline. `crank_timeout` is permissionless, and by Proposition 7.1 it reaches the unique post-lapse state. The lapse function maps each obligation kind to a named outcome that records the obligated seats. Staller-forfeits redistributes only the stallers' chips,

Table 6: Guarantees by adversary.

Adversary	Can	Cannot
One malicious seat	stall (and pay the stall penalty); fold	see others' cards; alter any card
All seats but one	pool their own cards; stall	see the honest seat's cards or deck order
All seats	everything (no honest party)	break chip or fee conservation
Observer / front-runner	read all public data	see any hidden card; steal a whistle proof
Lying RPC node	delay a client; withhold data	alter signed data; fake a deal or payout
Verifier bug	accept a bad step (if exploited)	survive a second-line fraud proof

so honest stacks never decrease because of a stall. The misbehaving-client runs (Table 12) check this on a live chain. □

Proposition 8.7 (Economic invariants)**.** *Every instruction preserves (I1), (I2) and (F) of Section 7. A whistleblower proof against oneself is never profitable: with seat bond B and reward share $w < 1$, a player that runs two seats and whistles on itself nets $wB - B < 0$. Fees cannot be under- or over-charged: the single fee path checks the treasury's growth after the transfer, and the client's* `max_fee_lamports` *guard bounds what any schedule change can take from a signed transaction.*

These properties are code-level rather than cryptographic. They are enforced by assertions in the program and checked after every step of stateful fuzzing (Section 11).

Concrete security. Ristretto255 provides about 126 bits of generic security. Every challenge is a full 252-bit scalar from a 512-bit reduction. The largest soundness term, $O(N^2)/\ell$ per shuffle for $N = 52$, is below 2^{-240}, so even $Q = 2^{80}$ oracle queries leave the Fiat–Shamir knowledge error far below the group's DL security. Every component has a fixed size, so the adversary has no length to tune.

9 Public Verifiability: Verify This Hand

The theorems of Section 8 hold for every run the program accepts. They ask a player to trust that the program's verifier ran, and ran correctly, and that the client which played the hand reported it faithfully. A player checking a hand afterwards, or a third party who took no part in it, should need neither. memepoker therefore has an independent hand verifier, `mp_verify`, which re-derives a finished hand from plain chain data and re-runs every check off chain. The same Rust code runs in the desktop client, in the browser game and, compiled to WebAssembly, on a static `/verify` page. That page talks JSON-RPC directly to a node the user chooses, so no memepoker server is involved. Anyone who knows a table and a hand number can run it, during play or years later. This section states what the verifier checks, why each check is sound, what it trusts, and what a pass does and does not mean.

9.1 From chain data to a hand record

The verifier takes the `mp_poker` program id, a table, a hand id and, optionally, the hand's time window and the viewer's seat. It reads three kinds of data over JSON-RPC: the protocol configuration account (for the cluster genesis hash bound into every transcript, Section 3.1), the table's hand account (for the final rules state, while the hand is still the table's latest), and the transactions of the hand (Section 9.6). From these it builds a *hand record*

$$\mathcal{R} = (\mathit{genesis},\ (pk_i, \sigma_i^K)_i,\ (C^{(i-1)}, C^{(i)}, \Pi_i)_i,\ (s, k, D, \sigma^D)^*,\ \mathit{board},\ \mathit{shown},\ \chi,\ (p_s)_s,\ \eta,\ S),$$

Table 7: The checks of the hand verifier, the data each reads, and the result its soundness rests on.

Check	Reads	Sound by
completeness	every folded event; (c, δ) at settlement	SHA-256 (random oracle)
keys	all (pk_i, σ_i^K)	Schnorr special soundness
shuffles	$C^{(0)}$; all $(C^{(i)}, \Pi_i)$; commitments	Theorem 3.2, Lemma 3.1
shares	all (s, k, D, σ^D); $\mathsf{H}(C^{(n)})$	Chaum–Pedersen soundness
hole privacy	positions and posters of shares	deal plan (Section 5)
cards	announced cards; $C^{(n)}$; shares	Theorem 8.4
payouts	χ, (p_s); S if $\mathsf{H}(S) = \eta$	rules engine; SHA-256

where σ_i^K is participant i's Schnorr proof, $C^{(i)}$ the deck after round i with $C^{(0)}$ the canonical trivial deck, Π_i the round's Bayer–Groth proof, each share D comes with its poster s, deck position k and Chaum–Pedersen proof σ^D, χ is the number of chips in the hand at its start, p_s the chips paid to seat s, η the hash the program logged over its final rules state, and S that state when it is available.

Only what the program vouched for. A transaction contributes only if its status is success. Instructions count only when they are top-level calls of the given program id, and events (Anchor's `Program data:` log lines) only when they were logged while `mp_poker` itself was the executing program. The decoder tracks the invoke stack, so another program in the same transaction cannot forge an `mp_poker` event. Instruction and event discriminators are derived the way Anchor derives them, never copied, and a test encodes every instruction and event with the IDL-generated types and decodes it with the verifier's own parser, so the two cannot drift.

Rebuilt the way the program built it. The record follows the program's own state machine. A deck or proof chunk counts only together with its upload event, so a superseded move, which writes nothing, contributes nothing. A rejected round (`ShuffleRejected`) is wiped. An accepted round (`ShuffleVerified`, or a class-O `ClaimOpened`) chains its output into the next round's input. A share counts only with its `SharesPosted` or `HoleCardsShown` event. A round the chain accepted but whose uploads the scan did not see in full is kept with its gaps, and the checks report it.

Independent of the program's execution. Every cryptographic check runs the off-chain `pcrypto_*` verifiers over the `curve25519-dalek` backend, not the program's syscall backend. The joint key is recomputed from the registered keys, and the card points and the starting deck are re-derived from HashToGroup rather than read from the program's pinned table. The verifier depends on the program's result only where the result is the claim being checked, the announced cards and the payouts, and on the program's count of the hand's events for completeness (Section 9.2). It does share code with the program: the transcript format, the proof parsers and the primary shuffle-verifier arrangement. A bug in that shared code would not be caught by this verifier. It would be caught by the second line (Section 7).

9.2 What is checked, and why each check is sound

The verifier reports one line per check, each *pass*, *fail* or *info* (could not be checked, with the reason). A hand *passes* when no line fails and at least one passes. A record without the hand's `start_hand` fails, and a hand that has not settled is reported as info, not as a pass. Table 7 summarizes the checks; we take them in order.

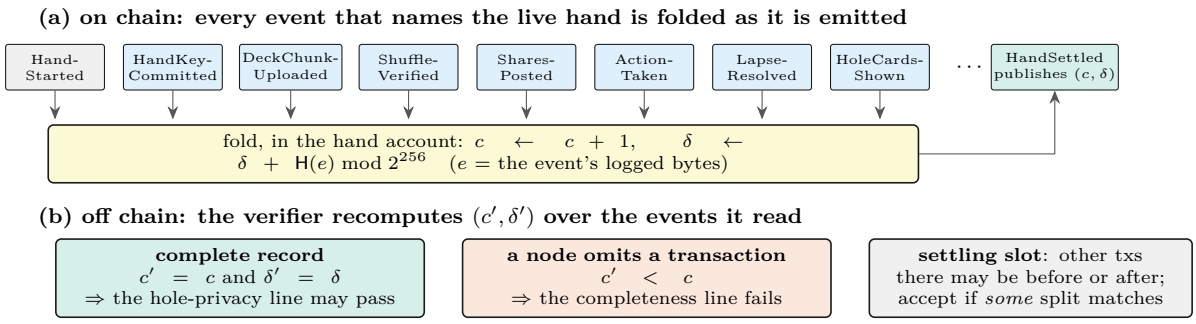


Figure 13: The hand transcript. (a) The program folds every event of the live hand into a count c and a digest δ (event names from `mp_poker::events`) and publishes both at settlement. (b) The verifier recomputes them over the events it read: omission changes the count, and the digest settles which transactions of the settling slot belong to the hand.

Completeness. The program keeps a *transcript* of each hand in two fields of the hand account: a count c and a digest δ. Every event that names the live hand, from `HandStarted` up to the settlement, is folded in as it is emitted, together with its index: the program first logs c, then sets $\delta \leftarrow \delta + \mathsf{H}(c \,\|\, e) \bmod 2^{256}$ and $c \leftarrow c + 1$, where e is the event's logged bytes (discriminator and Borsh fields). The folded events are those of every step that moves the hand: key commitments, deck and proof chunks, verifier steps, accepted and rejected rounds, claims, shares, board openings, actions, reveals, mucks, lapses, superseded moves, and the opening and resolution of a challenge. `HandSettled` publishes (c, δ). The verifier recomputes both from the events in the transactions it read and requires equality. The digest is a sum, not a hash chain, because a node need not list one slot's transactions in block order. The verifier pairs every event it read with its logged index and requires the indices $0, \ldots, c-1$ exactly once each and the published digest. An event emitted after the settlement carries no index and is never counted, so the order of the settling slot's transactions does not matter. (Hands settled before indices existed are matched as before: each other transaction of the settling slot is tried before or after the settlement.) Under (V1) the events read are a sub-multiset of the events executed, so a complete index set already means nothing was left out; the digest binds each index to its event, so a different event at the same index would require a collision of H. Hands settled before the program kept a transcript ($c = 0$) are reported as info: their record cannot be proven complete. This line relies on the program in one respect only: it trusts the program to fold every event of the hand. The LiteSVM suite re-derives (c, δ) from the logs after every instruction it sends and checks that each move on a live hand folds at least one event.

Keys. For each participant i of the deal order the record must hold pk_i and a Schnorr proof that verifies under purpose `keycommit` with the tag (`memepoker/v1`, program, *genesis*, table, hand) and the holder index bound. The joint key is recomputed as $PK = \sum_i pk_i$. By special soundness and Fiat–Shamir with rewinding, as in Section 3, each accepted proof means its poster knew sk_i, so no participant chose its key as a function of the others' and PK is not a rogue key.

Shuffles, chained from the opening deck. Let $C^{(0)} = (\mathcal{O}, M_j)_{j=0}^{51}$ with the M_j re-derived. For $i = 1, \ldots, n$ the verifier requires that round i's recorded input equal $C^{(i-1)}$ byte for byte, that its output be complete, and that Π_i verify for $(PK, C^{(i-1)}, C^{(i)})$ under purpose `shuffle` with round i bound. When the chain logged a deck commitment on acceptance, it also requires

$\mathsf{H}(C^{(i)})$ to equal it, which binds the rebuilt deck to the bytes the on-chain verifier accepted. A dealt hand must have all n rounds. By Theorem 3.2 and Lemma 3.1 each accepted round has an extractable witness (π_i, ρ_i), so $C^{(n)}$ encrypts $(M_{\sigma(k)})_k$ under PK for $\sigma = \pi_1 \circ \cdots \circ \pi_n$, exactly as in the proof of Theorem 8.4. Two points deserve mention. Rounds accepted optimistically under class O, which the chain did not verify at the time, are verified here like any other. And rounds the chain rejected are reported but play no part in the chain of decks.

Decryption shares. Every share is verified by its Chaum–Pedersen proof under purpose `share` or `reveal`, for the poster's registered key, the ciphertext component C_1 at its position in $C^{(n)}$, and the deck commitment $\mathsf{H}(C^{(n)})$ computed by the verifier. Special soundness gives $D = sk_s C_1$ unconditionally, up to the Q/ℓ term. Because the transcript binds the deck commitment, a share proof that verifies against the verifier's $\mathsf{H}(C^{(n)})$ was made for that deck: a share cannot be carried over from a different final deck without a SHA-256 collision or a random-oracle failure.

Hole-card privacy. Using the deal plan of Section 5, the verifier requires that no seat posted a share of its own hole card except in its showdown reveal, that no reveal came from a seat that did not show, that no reveal targeted a board card, and that every other share sits at a position the program accepts: another seat's hole card (once, in the deal), a board position or, at a table that allows run it twice, a second-board position. A seat that leaves the hand releases its shares of every undealt board position in advance, the second-board ones included, whether or not the hand runs twice; a position opens only when every holder's share of it is public, and none of these is a hole card or a burn. As shown in Section 5, until seat s posts its own share of a hole card, the other seats' shares leave that card an ElGamal ciphertext under pk_s alone. So when this line passes on a complete record, every hole card was opened by its owner and nobody else, unless that owner showed it. This check looks for an offending share, so it is sound only on a complete record. It therefore passes only when the completeness line passes. On a record that is incomplete, or that cannot be proven complete, it reports info instead (Section 9.3).

Board and shown cards. For every board card the chain announced, the verifier requires a non-reveal share from every participant at that position, computes $C_2 - \sum_i D_i$ from the final deck and matches it against the card table. For every shown hand it does the same with the owner's reveal shares included. Since the shares are correct and the deck is a permutation of the cards, the result is $M_{\sigma(k)}$ and the lookup is unambiguous (Theorem 8.4). A card announced by the chain that is not the card in the deck fails the line.

Payouts. Fees are paid in SOL and there is no rake (Section 7), so every chip in a hand must return to its players. The verifier always checks *conservation*: the payouts sum to the chips the hand started with, $\sum_s p_s = \chi$, and no seat that was not dealt in is paid. When the hand completed and its final rules state S is available, it also checks *correctness*. It requires $\mathsf{H}(S) = \eta$, the hash the settling transaction logged, then re-settles S through the same rules engine the program uses (`mp_poker_core`: pots, side pots, odd chips). It requires the result to equal the paid (p_s), and the board and shown cards inside S to equal the ones the chain announced. For an aborted or restarted hand the stall rules decide the split, and the verifier checks conservation only.

9.3 Trust model: consensus data, not the RPC node

The verifier does not trust the program's verifier, the client that played the hand, or the site that serves the page. Whether it can avoid trusting the RPC node it reads from depends on what the node does, and we separate two cases.

(V1) *Authentic data.* Every transaction the node returns, with its status and logs, was executed by the cluster with that result, and every account it returns holds what the cluster stored. The node may omit, reorder, delay or refuse to return anything.

(V2) *Fabricated data.* The node returns transactions or accounts that the cluster never executed or stored.

Solana's JSON-RPC returns no inclusion proof, and a transaction's status and logs are not signed by anyone. The verifier checks neither transaction signatures nor any proof that a transaction was finalized. (V1) is therefore an assumption about the data source. It holds for the user's own node, and it can be checked by comparing two independent nodes, or by comparing the reported settling signature against a second source. Under (V1) the verifier relies only on consensus data, and the node's only power is to omit.

What omission can do. Most lines assert that positive evidence is present. All n keys must be present with valid proofs. All n shuffle rounds must be present and complete, must chain from $C^{(0)}$, and must hash to the commitments the chain logged. Every announced board card needs every participant's share. The start and the settlement are single events that omission can remove but cannot change. Omitting transactions can make these lines fail, make a hand come out as not found or not settled, or end the scan early, but it cannot make them pass for a hand that does not satisfy them. Three outputs rest instead on something being *absent*: the hole-privacy line passes when no offending share is found, the shuffle line counts the rounds the chain rejected, and an info line reports fraud proven against a seat. Without more, a node could hide an offending share, a rejected round or a fraud proof by omitting the transaction that carries it. The completeness line closes this gap. A node that omits any transaction of the hand changes the count the verifier recomputes, so the completeness line fails with the number of events expected and found. The hole-privacy line passes only on a record proven complete. The rejected rounds and the fraud proofs are reported as exact only then, and as lower bounds otherwise. Fraud proven *after* settlement, inside the second-line windows, is not part of the hand's transcript. The verifier reports what it found of it as info, and makes no claim of completeness for it.

Proposition 9.1 (Omission is safe)**.** *Assume (V1), DL in $\mathbb{G}$, the random-oracle model, and that the program folds every event of a live hand into its transcript. Let $\mathcal{R}'$ be built from any subset of a hand's transactions. (a) If the completeness line passes on $\mathcal{R}'$, then $\mathcal{R}'$ holds every event of the hand up to its settlement, except with negligible probability. (b) If any line passes on $\mathcal{R}'$, then the property it states holds for the hand as the cluster executed it, except with negligible probability. In particular, omitting transactions can make verification fail or report the record as incomplete, but it cannot make it pass for a hand that does not satisfy its checks.*

Proof sketch. Under (V1) every item in $\mathcal{R}'$ is an item of the executed hand. (a) Let E be the set of folded events, the i-th logged with index i, and (c, δ) the pair `HandSettled` published. Under (V1) every indexed event the verifier read is the executed event with that index, and events after the settlement carry no index. The line passes only if the indices read are exactly $0, \ldots, c-1$, so every event of E was read; the digest check then fails unless each pair (i, e) is the executed one, except for a collision of sums of random-oracle outputs, which has negligible probability. (b) The key, shuffle, share, card and payout lines each require their evidence to be present: n keys, n chained rounds, every participant's share per board card, both events for the payouts. Deck chunks are write-once per round and wiped on rejection, so a complete output rebuilt from executed writes is the deck that round uploaded, and the logged commitment ties it to the deck the chain accepted. A share proof that verifies against $\mathsf{H}(C^{(n)})$ ties $C^{(n)}$ to the deck the shares were made for. The soundness arguments of Section 9.2 therefore apply to the true values, with or without omission. The hole-privacy line passes only together with the

completeness line, so by (a) it is evaluated on every share the chain accepted, and its argument applies as well. □

What a fabricating node can do. Under (V2) the verdict speaks only about the data served. A node that alters a hand in place is caught. Changing one byte of a shuffle proof invalidates the proof, and moving one chip in the payouts breaks conservation or the re-settlement; our tests check exactly these two through a lying node (Section 10). But nothing in a hand's proofs names a wallet: the transcript binds program, cluster, table, hand and seat index, not the seat's signing key, and the per-hand keys are fresh. A node prepared to fabricate a whole hand, with its own keys and valid proofs, or to rewrite the start's chip count together with the payouts, could therefore present a self-consistent hand that never happened, and the verifier would pass it. We do not claim otherwise. The defense against (V2) is to choose the data source. The page lets the user name the node, and the report gives the settling transaction's signature so that the hand can be checked against a second source.

9.4 What a pass means

Proposition 9.2 (A verified deal is a fair deal)**.** *Assume (V1), DL and DDH in $\mathbb{G}$, and the random-oracle model. Suppose the verifier passes the key, shuffle, share and card lines for a hand that was dealt, with participants $1, \dots, n$. Then, except with negligible probability:*

(i) there is a permutation σ of $\{0, \dots, 51\}$ such that the final deck encrypts $M_{\sigma(k)}$ at every position k, and every board card and shown card the chain announced at position k is $M_{\sigma(k)}$;

(ii) if at least one participant h ran an honest client, then no coalition of the other participants could bias or predict σ: when each coalition member fixed its permutation, and at every point before a position was opened, the card at that position was uniform, from the coalition's view, over the cards outside its leakage $L_{\mathcal{A}}$. In particular, no coalition could steer a chosen card to a chosen position with probability better than $1/(\text{number of unopened cards})$ plus a negligible term.

Proof sketch. (i) By Proposition 9.1 the passing lines speak about the executed hand. The shuffle line gives an accepted Bayer–Groth proof for every link $C^{(i-1)} \to C^{(i)}$ from the trivial deck $C^{(0)}$, so by knowledge soundness (Theorem 3.2, Lemma 3.1) there are witnesses (π_i, ρ_i), unless a DL relation among the commitment generators is found. By induction $C^{(n)}$ encrypts $(M_{\sigma(k)})_k$ with $\sigma = \pi_1 \circ \cdots \circ \pi_n$. The share line makes every share used correct, and the card line opens each announced card from $C^{(n)}$ with them. This is the argument of Theorem 8.4, run over the verifier's record instead of the program's state. (ii) The shuffle line requires all n rounds, so h's round is in the chain of decks, with π_h uniform and ρ_h fresh because h is honest. The key line ensures the coalition's keys are extractable, so Theorem 8.2(a) applies to the coalition $\{1, \dots, n\} \setminus \{h\}$. Its view is simulable from $L_{\mathcal{A}}$ alone, which gives the uniformity claim, and the steering bound is the corollary to Theorem 8.4. □

Part (i) needs no honest party: it holds even if every seat was one person. Part (ii) does need one, and the verifier cannot check that hypothesis, because no chain data shows whether a client sampled its permutation honestly or kept its key secret. A player who verifies their own hand knows the hypothesis holds, since they were the honest participant. A third party gets (i) unconditionally and (ii) conditionally on at least one participant being honest. A hand among sock puppets of a single person is a fair deal in no useful sense, and nothing can show otherwise.

9.5 How deep the payout check goes

The two depths of the payout check differ in the data they need. Conservation needs only the start and settlement events, so it is available for every hand forever. It catches chips paid to the house or to a seat that was not dealt in, and chips created or destroyed, including a single extra chip. It does not catch a split among the players that conserves chips but is wrong, such as a side pot paid to the wrong player. Full re-settlement catches that as well, but it needs the final rules state S. The program keeps S only in the table's hand account, and the next hand at the table overwrites it. The verifier therefore reaches full depth for the table's latest hand, read from the account, and for any hand whose S the verifying client kept while it watched the hand and passes in. For older hands it reaches conservation only. The report's line states which depth was reached, and in the conservation case it says that the side-pot split was not re-checked.

Full depth rests on two assumptions beyond (V1). First, the binding of η: the program logs $\eta = \mathrm{SHA256}(S)$ over exactly the bytes it settled, so a substituted state requires a SHA-256 collision. Second, re-settlement runs the program's own rules engine, so it shows that the program paid what its rules prescribe for S, not that the rules are correct poker. A bug in `mp_poker_core` that the program shares is not detected here. The red-team suites and fuzzing of Section 11 cover the rules engine instead. Full depth for every hand would mean replaying every betting action from the start event through the rules engine, without relying on the stored S. That needs no new chain data, but it is not implemented.

9.6 The scan and its cost

The scan is a state machine that issues no I/O itself (Figure 14). It says which JSON-RPC calls to make and is fed their answers, so the desktop client, the browser game and the WebAssembly page drive the same code with at most four calls in flight. It proceeds as follows.

1. Two `getAccountInfo` calls: the protocol configuration (genesis hash) and the table's hand account (the final state S, kept only if the account still holds this hand).

2. `getSignaturesForAddress` on the hand account, newest first, 1,000 signatures per page. The hand's transactions reference this account, and every `start_hand` writes it. Failed transactions are not fetched. With a time window (a share link carries the hand's start and settle times), transactions outside the window plus 60 s of slack are not fetched, and paging stops at the first one older than it.

3. `getTransaction` for each listed signature, newest first, until the hand's own `start_hand`, or a transaction of an earlier hand, is reached. Transactions in the same slot as that point are still read, because a node need not list one slot's transactions in block order. The kept transactions are then ordered by slot, by their place in the hand's life (start, keys, each round's uploads before its verification, then play and settlement), and by list order.

The fallback. Seats that use a per-table address lookup table (Profile A, Section 6) load the hand account through it. Some nodes, Surfpool among them, do not index addresses loaded that way, and their listing of the hand account is then missing those transactions. A complete listing always contains the hand's `start_hand`. When the listing ends without it, the scan pages the `mp_poker` program id instead, which is a static key of every one of its transactions. It skips what it already read and keeps transactions that name this table. The table key is visible only inside a transaction, so this filter applies after fetching.

Cost. Let L be the number of signatures listed and F the number of transactions fetched. The scan makes $2 + \lceil L/1000 \rceil + F$ calls. On the primary path with a window, F is the hand's

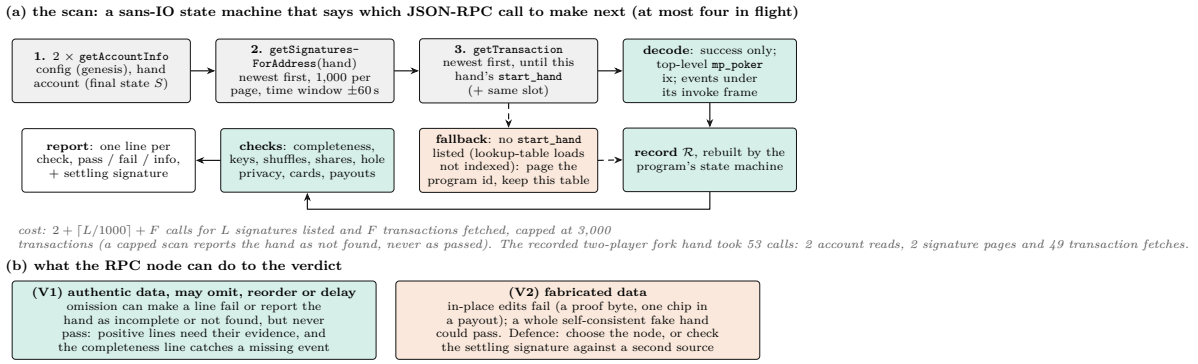


Figure 14: The verifier's scan and trust model. (a) The JSON-RPC calls, the fallback for nodes that do not index lookup-table loads, and the decode, record and check stages, with the measured call count of the recorded fork hand. (b) What a node can do to the verdict under (V1) and (V2).

own transactions plus those in the slack. That is of the order of Table 11's cryptographic transactions plus the betting and settlement, about a hundred at six seats. Without a window, F also includes every later hand's transactions at the table, back to the one being verified. In the fallback, F is every `mp_poker` transaction on the cluster in the window, on every table. The window is what keeps the fallback practical on a busy cluster. Both paths are bounded by a transaction cap (3,000 by default). When the cap is reached, the scan stops and says so. Because the record then lacks the hand's start, the hand is reported as not found: incomplete, never passed. The recorded two-player hand of our fork test took 53 calls: two account reads, two signature pages and 49 transaction fetches. The cryptographic work is the off-chain verification of n shuffles and the hand's shares. On the host it is dominated by about 4.4 ms per shuffle (Section 10), small next to the RPC round trips. We have not timed the WebAssembly build.

10 Implementation and Evaluation

Code. The cryptography is five `no_std` Rust crates with `unsafe` code forbidden: `pcrypto_transcript`, `pcrypto_elgamal` (including `lean`), `pcrypto_dleq`, `pcrypto_shuffle` (prover, primary and second-line verifiers) and `pcrypto_cards` (the deck, deal plan and a public card-game state machine that verifies every message it applies). Together they are 4,847 lines of source and 5,076 lines of tests, with known-answer vectors pinned in JSON so that the host, WebAssembly and SBF builds cannot drift. `pcrypto_solana` and the bond library add 1,060 lines, and `mp_poker` is 8,520 lines of Anchor program (all counts are `wc -l` over the committed `src` files, comments and blank lines included). The same prover code runs natively and in the browser.

Harness. All numbers below were measured locally. `bench_verify` is a test-only program that is never deployed. It runs every verification step through `pcrypto_solana::Sol` over chunk-uploaded accounts. LiteSVM 0.6 executes the real compiled `.so` files with the agave 2.2 cost tables. CU is the transaction's `compute_units_consumed`, including the 150 CU of the compute-budget instruction. The `mp_poker` figures come from full hands with real proofs at 2, 6 and 9 seats.

The hand verifier. `mp_verify` (Section 9) is 2,283 lines of Rust with `unsafe` code forbidden: the sans-IO scan, the transaction decoder, the record builder and the checks. `mp_sdk` drives it over its RPC pool for the desktop client, the browser game and the end-to-end runner, and

Table 8: Primitive costs on the compiled program.

Step	CU	notes
key commit: Schnorr verify + joint-key add	12,404	tx 387 B
deck commitment (SHA-256 of 3,328 B)	2,724	one `sol_sha256`
Chaum–Pedersen share verify, $k = 1$	17,131	tx 490 B
Chaum–Pedersen share verify, $k = 8$	128,534	tx 1,176 B (largest k)
marginal per extra share	15,914	
challenge over 1,024 B, `sol_sha256`	7,229	
the same, software SHA-256	151,576	21×

Table 9: One $N = 52$ shuffle round on chain (single signer, buffer accounts only).

#	transaction	CU	bytes
1–3	write 1,013 B each	838 each	1,232
4	write 960 B (two instructions)	1,676	1,232
5–7	write 1,013 B each	838 each	1,232
8	write 130 B + begin + `0x001` + `0x1fe` + finish	**1,016,627**	861

`mp_verify_web` wraps it for the `/verify` page. Three layers of tests cover it. A synthetic hand built with the real provers (three players, every shuffle, share and reveal) verifies, and each of six tampered variants fails on the expected line: one changed shuffle-proof byte, a wrong announced flop card, payouts that do not add up, a broken DLEQ proof, a seat publishing a share of its own hole card, and a hand that is unfinished or never found. The same synthetic hand is proven complete, and removing any one of its transactions never yields a pass. A share of a seat's own hole card that the chain accepted makes the privacy line fail, and omitting it makes the completeness line fail while the privacy line is withheld. A hand without a transcript passes its positive checks and reports privacy as info. A recorded fork hand replays every JSON-RPC answer through the same scan: it verifies with the full re-settlement and a proven-complete record, and fails on the shuffle line when one answer has a shuffle-proof byte changed. Replayed once for each of the hand's transactions with that one answered as missing, it never passes. Finally, the end-to-end runner plays hands with real bots on a local fork and verifies them through a hook that acts as a lying RPC node. One run changes a shuffle-proof byte, another adds one chip to a payout, and a third omits one share transaction. Each must fail on the right line: shuffle, payout and completeness.

Primitives. Table 8 gives the per-primitive costs. A decryption share verifies for 15,914 CU at the margin, and eight fit in one transaction.

One shuffle round. Table 9 shows the measured schedule of one 52-card round with writes packed to the 1,232-byte limit. It takes eight transactions and 1,023,331 CU in total, and uploads 7,168 bytes. On the host (Apple Silicon), proving takes about 15 ms and verifying about 4.4 ms. The host verification performs 260 validated point decodes, nine MSMs over 335 points (the largest has 113), 325 scalar multiplications, eight challenges and 9,815 hashed bytes. For the 13-card deck of the generic card-game crate ($m = 1$, proof 1,728 bytes), a round is three transactions and 313,473 CU. Figure 15 plots the round against the limits.

Inside the poker program. Table 10 gives the largest CU per instruction measured in complete `mp_poker` hands. These runs use legacy transactions with every account inline (Profile B),

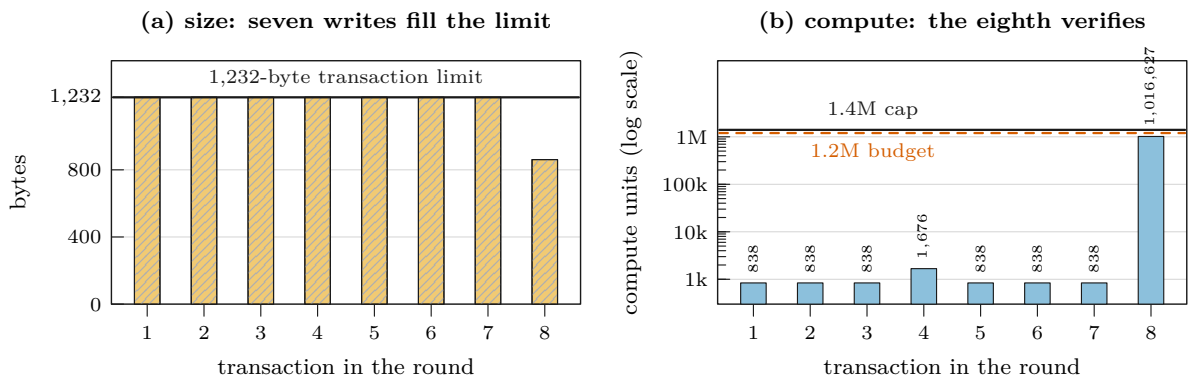


Figure 15: One $N = 52$ shuffle round in the benchmark harness (data: `docs/G_SHUFFLE.md`). (a) Seven write transactions fill the 1,232-byte limit; (b) the eighth carries the verification, at 1,016,627 CU, under our 1.2M budget and the 1.4M cap.

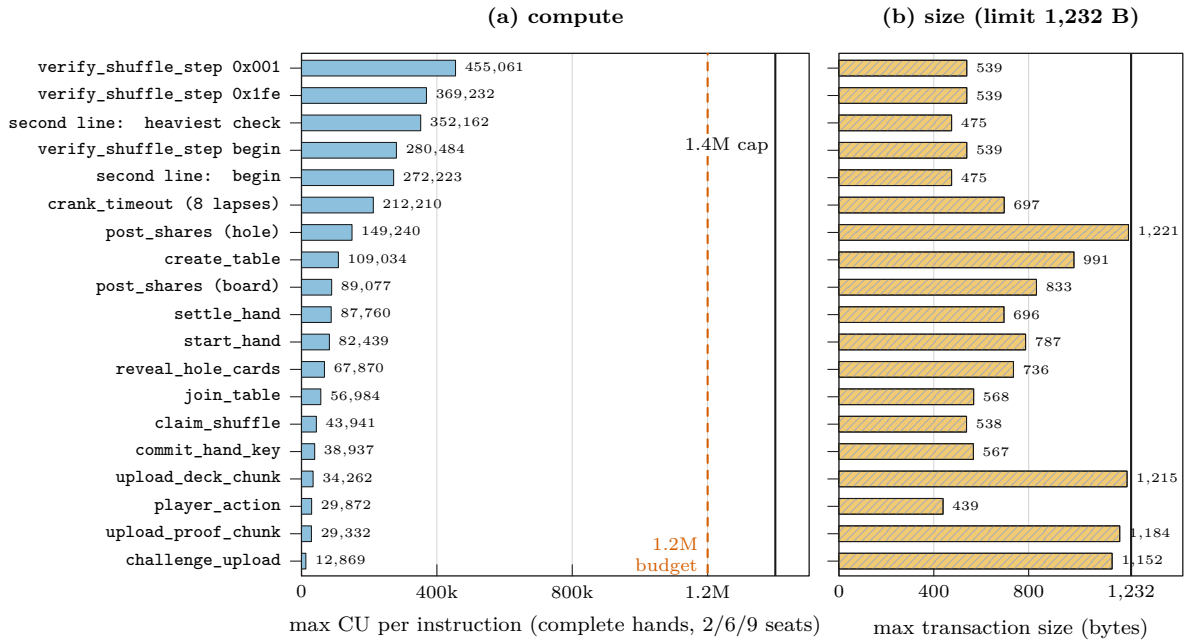


Figure 16: `mp_poker`: largest compute (a) and transaction size (b) per instruction over complete hands with real proofs at 2, 6 and 9 seats, Profile B (data: `programs/target/poker-cu-report.md`, regenerated by `make programs-test`).

so a deck takes five chunks and a proof six, and verification runs as three separate transactions (*begin*, `0x001`, `0x1fe`). That is 14 transactions per round in this profile, against 8 with the lookup table of Profile A. Figure 16 plots the measured maxima against the limits.

The deal, betting and settlement instructions are small next to verification. Outside verification the largest is `crank_timeout` resolving eight lapses, 212,210 CU, then the hole-card `post_shares` at nine seats, 149,240 CU for eight shares. The heaviest second-line challenge step stays at 352,162 CU. A six-seat round costs about 1,451,991 CU over 14 transactions, and a six-player hand at most 126 cryptographic transactions and 12.7M CU (Table 11). Six instructions exceed the per-instruction targets we set in the specification (§ 6.13), though all stay far below the 1.4M cap: `commit_hand_key`, `crank_timeout`, `reveal_hole_cards`, `start_hand`, `upload_deck_chunk`, `upload_proof_chunk`.

Table 10: `mp_poker`: largest CU per instruction over complete hands with real proofs (2, 6 and 9 seats; Profile B).

Instruction	max CU	max tx bytes
`commit_hand_key`	38,937	567
`upload_deck_chunk`	34,262	1,215
`upload_proof_chunk`	29,332	1,184
`verify_shuffle_step` begin	280,484	539
`verify_shuffle_step 0x001`	455,061	539
`verify_shuffle_step 0x1fe`	369,232	539
`post_shares` (hole, 9 seats)	149,240	1,221
`post_shares` (board)	89,077	833
`reveal_hole_cards`	67,870	736
`player_action`	29,872	439
`crank_timeout` (8 lapses)	212,210	697
`start_hand` (9 seats)	82,439	787
`settle_hand` (9 seats)	87,760	696
`challenge_verify_step` (second line, heaviest)	352,162	475

Table 11: Cryptographic cost per hand ($N = 52$, every participant shuffles once). Top: the benchmark harness (Profile A sizes). Bottom: the deployed program.

players	shuffle txs	shuffle CU	key txs	share txs	all crypto txs	all crypto CU
2	16	2,046,662	2	10	28	2,370,100
6	48	6,139,986	6	36	90	7,881,504
9	72	9,209,979	9	54	135	12,681,648
`mp_poker`, Profile B: calls in the measured hands × max CU per instruction (an upper bound)						
2	28	2,901,690	2	10	40	3,772,110
6	84	8,711,946	6	36	126	12,732,918
9	126	13,063,221	9	63	198	20,459,016

Per hand. Every participant shuffles once, posts its hole-card shares for the other seats, one transaction each for the flop, turn and river shares, and its own reveal at showdown. Table 11 gives the totals with the bench profile's transaction sizes, where at six players a hand needs 90 cryptographic transactions and 7.9M CU, and the totals measured in `mp_poker` (Profile B). The shuffle rounds are sequential, but all eight transactions of a round except the last can land in the same slot.

Rent. The per-table verifier accounts are a 290-byte scratch (2,909,280 lamports rent-exempt), the 3,328-byte staged deck (24,053,760) and the 3,840-byte proof buffer (27,617,280). The table creator funds them once, and they are reused for every hand.

What we did not re-measure. The pass criteria we set before building required native proving at most 2 s and WebAssembly proving at most 6 s. Native proving is about 15 ms, more than two orders of magnitude inside the budget. WebAssembly proving runs the same code but was not re-timed for this paper. The end-to-end six-player pre-deal time on a local fork running at mainnet's 400 ms slots (target at most 15 s) is likewise not reported here. A lower bound follows from the schedule: two slots per round and six sequential rounds give about 4.8 s, but it is an estimate, not a measurement.

11 Security Testing Methodology

The proofs in Section 8 are only as good as the code that implements them. We test the implementation adversarially, and every test runs locally.

Named attacks. The catalogue in `docs/security/ATTACKS.md` lists every attack as a test that must *fail to exploit*: the attack is rejected with its named error, or the defense holds and no state is damaged. The off-chain crates carry 1,012 named attacks. 190 of them target the cryptography: 23 on the transcript (length-extension and concatenation ambiguities, label collisions, domain-tag swaps), 33 on ElGamal encodings (non-canonical points and scalars, identity and low-order inputs, malleability), 50 on Schnorr and Chaum–Pedersen (replay across every bound field, zero or identity components, cross-proof-kind replay, nonce reuse), 53 on the shuffle (duplicated, dropped or substituted cards, permuted proof elements, wrong shapes, tampered scratch, mismatched decks) and 31 on the card-game state machine. The on-chain suites add the RT-PK, RT-PG and RT-PR families (accounts, money paths, gameplay, proofs, bonds, timeouts and fees). In those families, every row of the attack catalogue in the specification maps to a test.

Per-instruction batteries. For every entry point of every program, a battery starts from that instruction's own happy-path fixture, changes one thing, and checks three properties. A rejected transaction must leave every referenced account byte-identical. The outcome must equal a pinned result, and every accepted outcome carries a written justification and must have paid its scheduled fee. And the unmodified fixture must still succeed afterwards, as a control. The mutations cover a missing or substituted signer, a foreign owner, a substituted program-derived account (including another table's or another player's), aliasing, swapped mints and token programs, re-initialization and use after close, zero/one/max arguments, skipped or redirected fees, replay, and the upgrade authority acting as an ordinary payer. The inventory in `docs/security/COVERAGE.md` has no blank cell.

Fuzzing. Nine libFuzzer harnesses cover the off-chain decoders and verifiers. Each asserts an invariant, not just the absence of a panic. For example, `shuffle_verify` checks that *any* change to an honest proof or output deck is rejected by *both* verifiers, and `dleq_verify` does the same for any XOR change to proof, share, C_1 or key. The last run used 180 s per target: 70,809 shuffle and 149,620 DLEQ executions, and 9.7 million ElGamal-decode executions, with no crashes, timeouts or out-of-memory errors. A stateful program fuzzer runs LiteSVM against the real `.so` files. It seeds with every instruction in its valid state and mutates bytes, arguments, accounts, signer and writable flags, instruction order and the clock, and after every step it checks seven invariants, including (I1), (F), "a failed transaction changes nothing" and "no program account becomes undecodable". In a 240 s run on `mp_poker` it made 403,753 executions, 88,143 of them accepted mutated transactions, with zero findings. The randomness suite (`docs/security/RANDOMNESS.md`) tests the sampling statistically, and Figures 17 and 18 show its results.

Misbehaving clients on a live chain. A test-only build of the bot client misbehaves once per scenario at a three-seat table on a fresh offline local validator. A build check ensures that none of this code is in the honest bot. Table 12 shows the program's response in the last run; all six scenarios passed with chips conserved and every seat paid out.

What the testing found. Several attacks succeeded against the code as it then stood. Each was fixed, and each is now a regression test. Three are relevant to this paper. *RT-CR-173:* two colluders, each with a valid proof of knowledge, registered keys that summed to the identity. The

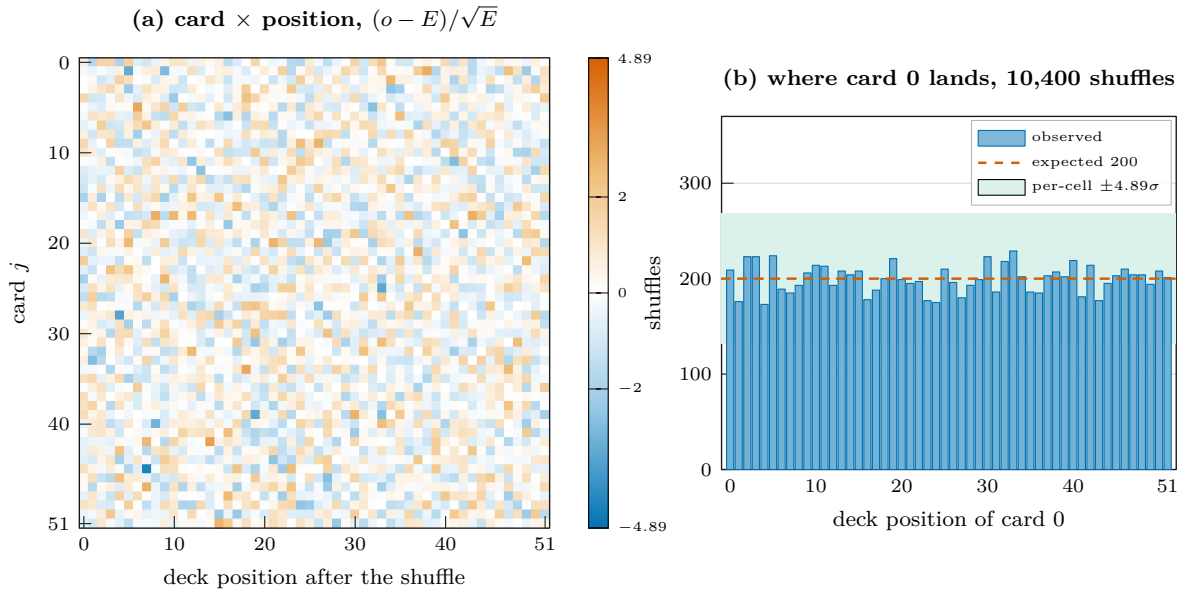


Figure 17: Uniformity of the shuffle's permutation sampling (test RND-U2, 10,400 seeded shuffles of 52 cards). (a) Standardized residuals of the card-by-position counts ($\chi^2 = 2{,}710$, df 2,601, acceptance $[2{,}272, 2{,}958]$); no cell reaches the per-cell 4.89σ bound. (b) Where card 0 lands (KS $D = 0.0063$ against a critical 0.0264; runs $z = -0.29$). Data dumped by the test itself (`RANDSTAT_DUMP_DIR`).

Table 12: Misbehaving-client scenarios on a local fork (3 seats, one misbehaving).

Scenario	Observed program response
stall an action	auto action by piggyback; no slash; next hand completes
act twice	second action refused (`StaleActionSeq`)
bad decryption share	refused; stall settlement to the contenders; bond to the treasury shard
bad reveal	refused; treated as a muck
bad shuffle	`MexpCt` step rejects; round wiped; restart with pre-deal slash
abandon after the deal	contribution and min(stack, $k \cdot$ pot) to the contenders; bond to the treasury shard

identity joint key was refused, but with every slot full and no culprit named, so the table stalled forever. The fix refuses any key that makes a partial sum the identity and blames the seat that submits it, which provably knows the discrete log of the earlier keys' sum. *RT-PG-VS-003:* the resumable verifier bound its scratch, deck and proof accounts by owner, discriminator and table field, but not by address. A program-owned byte copy at a different address could have driven a verifier step. This was not reachable, since only the program creates such accounts and only at their canonical addresses, but the verifier now requires canonical addresses. The fix also pinned two findings as design rules: *finish* trusts the scratch, and steps do not re-derive challenges, so the buffers must be frozen once *begin* has run. *RT-SE-013..016:* a session key checked scope and fee budget against a transaction preview's editable display fields while signing the underlying message bytes. The check now runs on private build-time copies of the message.

12 Related Work

Mental poker. Shamir, Rivest and Adleman posed mental poker with commutative encryption [1]. Barnett and Smart gave the now-standard structure of a joint ElGamal key, re-

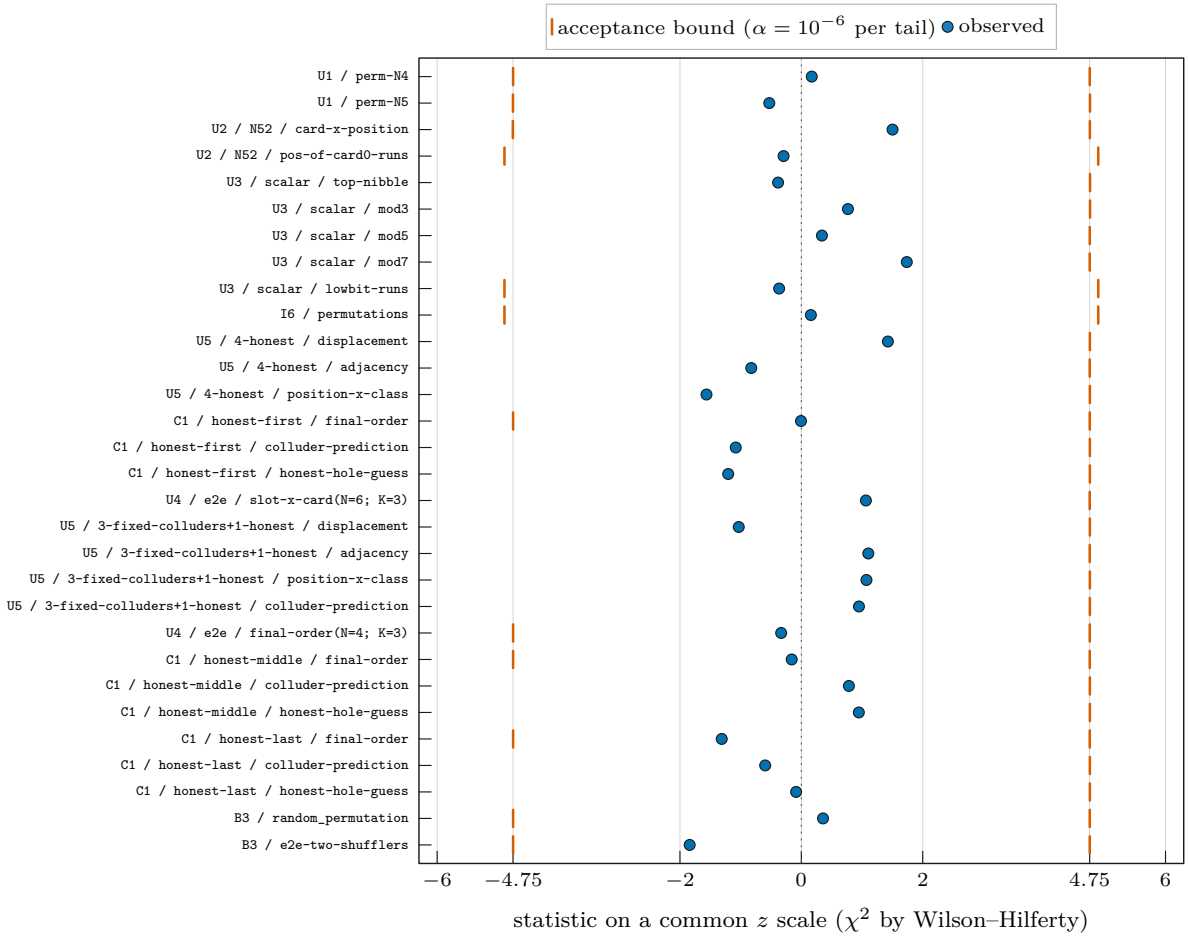


Figure 18: Every χ^2 and z statistic of the randomness suite's last run (`RANDSTAT` lines of `target/randomness.log`) on a common z scale, with its acceptance bounds at $\alpha = 10^{-6}$ per tail. One-sided tests have only an upper bound. The C1 and U5 rows fix three colluders' permutations and keep one honest shuffler: the order stays uniform.

encryption shuffles with proofs, and threshold decryption with proofs [2]. Our protocol follows that structure, replaces its shuffle proof with Bayer–Groth, and adds the on-chain enforcement around it. Royale [19] and Kaleidoscope [20] give universally composable treatments of card games with financial rewards and penalties. Bentov, Kumaresan and Miller [21] reduce poker on Bitcoin-style chains to secure computation with penalties, and observe that algebraic protocols make on-chain verification harder than generic ones. Insta-Pok3r [22] moves shuffling to an offline committee for real-time play.

Verifiable shuffles. Neff [27] and Furukawa–Sako [28] gave the first efficient shuffle proofs. Groth [29] and Terelius–Wikström [30] gave the commitment-based arguments behind modern mix-nets, and Bayer and Groth [3] reached sublinear communication. Curdleproofs [31] modernizes Bayer–Groth with inner-product arguments for Ethereum's single-secret leader election; it is a library, not a contract verifier. Off-chain Rust implementations of Barnett–Smart with Bayer–Groth shuffles exist. Geometry's arkworks library [32] and the recent `ziffle` crate [33] are examples; neither verifies on a chain.

On-chain card games. zkShuffle [34] proves Barnett–Smart shuffles and decryptions with Circom/Groth16 circuits over Baby Jubjub and verifies them in EVM contracts; zkHoldem

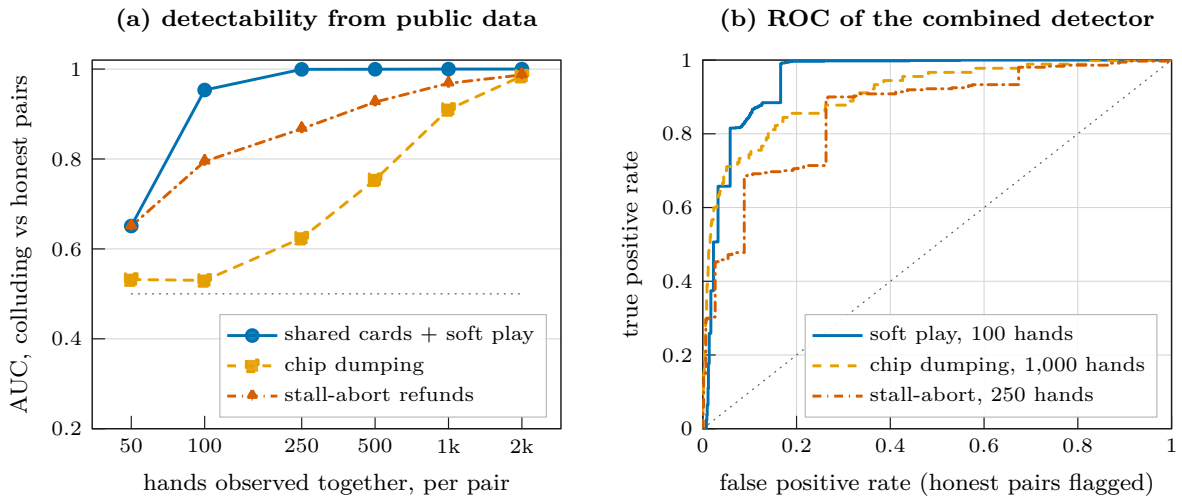


Figure 19: Collusion detection from public data only (self-play analysis, `tools/analysis`, `docs/analysis/ECONOMICS_AND_EXPLOITS.md`): 36 players with three colluding pairs, 180k hands per scenario. (a) AUC of the combined detector against hands a pair played together; (b) its ROC curves at 100, 1,000 and 250 hands.

runs on it. Zypher's Secret Engine [35] uses a PlonK shuffle with Solidity verifiers. A Starknet mental-poker implementation verifies Noir proofs with Garaga-generated Cairo verifiers [36]. All of these use SNARKs, so they need either a trusted setup (Groth16) or a universal one, and their proving is heavy in a browser. We evaluated a Baby Jubjub/Groth16 design for Solana (verified with the `alt_bn128` syscalls and a Poseidon deck commitment) as our fallback. The transparent design passed our benchmark gate, so the fallback was never needed. On Solana, existing poker projects deal with a VRF inside a TEE rollup [37], with an MPC network that keeps the cards encrypted [38], or with a single dealer whose deck commitment is verified by Groth16 at about 85k CU [39]. Each of these trusts a dealer, a committee or a setup in a way that memepoker does not.

Sigma proofs on Solana. The ZK ElGamal Proof program [40] natively verifies equality, validity and range proofs over twisted ElGamal on curve25519 for Token-2022 confidential transfers. It is the reason Solana exposes the curve25519 syscalls we build on. We use the syscalls directly from our own program rather than that program's fixed proof set, because no shuffle argument is among its instructions.

13 Discussion and Limitations

What contracts cannot prevent. Players who read their own cards to each other over voice or chat collude invisibly; no proof can exist for a fact the chain never sees. Whistleblower proofs make key sharing self-defeating, but they cannot touch players who simply talk. Centralized sites fight this with server-side analytics, which a system without servers cannot run. Our mitigations are client-side and statistical only. Figure 19 shows how fast a detector over public chain data separates colluding pairs in our self-play analysis, and Figure 20 what the stall-abort refund was worth to a colluding pair under the rules analysed there.

The analysis is a sketch. The theorems of Section 8 are proved at the level of hybrids and reductions, in the random-oracle model, with rewinding extraction for the Schnorr proofs. A

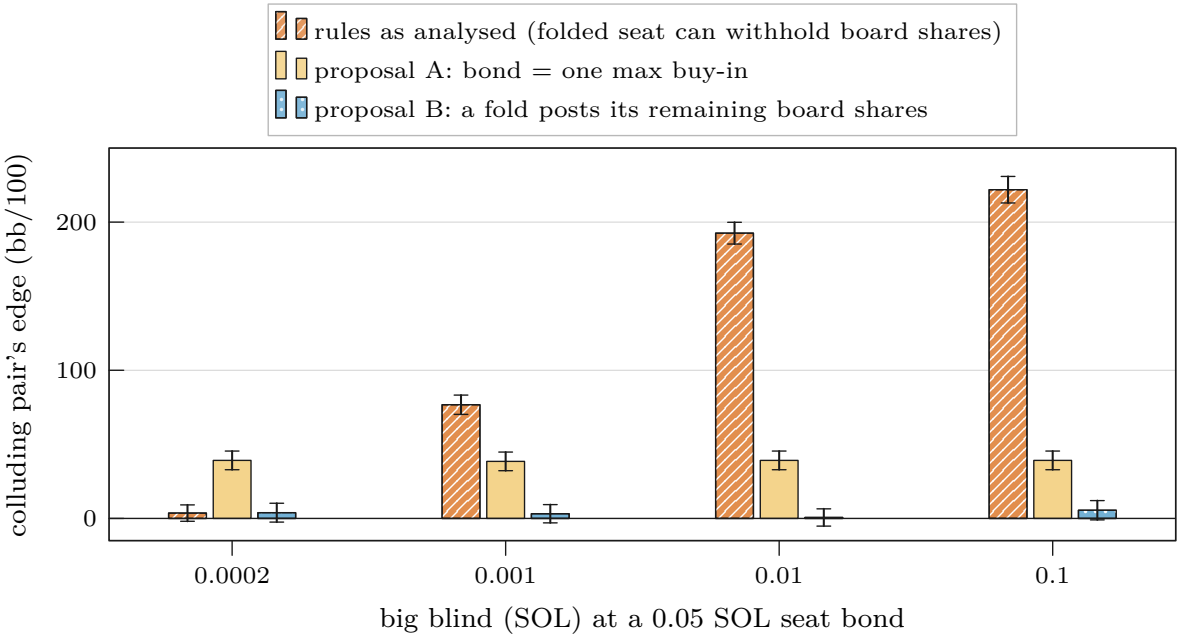


Figure 20: The stall-abort refund in the self-play analysis: a colluding pair's edge over honest players (mean ± 95% interval, bb/100) when a folded teammate withholds board shares while the pair is behind, by stake at a fixed 0.05 SOL seat bond. Sizing the bond to a buy-in (A) caps it; letting a fold post that seat's remaining board shares (B) cuts it but does not remove it: a timeout fold, an in-hand loser and a short or all-in staller still profit. Under the final rules (fold release, a no-peek posting order and an equity settlement with the stall penalty, Section 13) no coalition variant gains outside the 95% interval at any stake.

machine-checked or fully composable proof is future work, in the style of the formal treatments of [19, 20]. So is a tight bound for the multi-ciphertext DDH hybrid.

Verifier bugs. Validity-first enforcement trusts the primary verifier. The second line, differential fuzzing and the red-team suites reduce that risk without removing it. If the second verifier were the buggy one, an honest submitter could be slashed. We accept that residual risk and state it here. The verifier is custom code and has not yet had an external audit.

The hand verifier trusts its data source. The verifier of Section 9 trusts that the node returns transactions the cluster really executed. Solana's RPC gives no inclusion proofs, so a node willing to fabricate a whole self-consistent hand could make it pass. Its completeness line trusts the program to fold every event of a live hand into the transcript; the program's tests check this after every instruction, but a hand-scoped event left out of the fold would be invisible to it. Figure 13 illustrates the fold. Fraud proven after settlement lies outside the transcript, so the verifier reports it only as found. Hands settled before the program kept a transcript cannot be proven complete, and their privacy line is reported as info. The payout check re-settles only hands whose final state is still available.

Key theft equals key leaking. A whistleblower proof cannot tell a voluntary leak from malware that stole sk_A. Keys are memory-only by default and live for one hand, but a compromised machine can cost its owner the seat bond.

Latency. Shuffles are sequential, about 2 slots per participant at best, and `mp_poker`'s current Profile B measurement needs 14 transactions per round rather than 8. Pipelining the next hand's key commitment and shuffles during the current hand is designed for (the account seeds reserve a slot byte) but not implemented. End-to-end pre-deal times on a mainnet-paced local fork and WebAssembly proving times were not re-measured for this paper.

Last-revealer advantage and selective abort. As in every threshold-decryption protocol, the last party to post shares for a card sees the outcome first and may abort. Our self-play analysis found that the original abort rule made this a profitable option. That rule refunded every seat still in the hand and split the stalling seat's chips by contribution, and the bond was the only cost. A colluding pair whose folded teammate withheld its board shares whenever the pair was behind gained $+153.5 \pm 13.0$ bb/100 at 0.001 SOL blinds and $+443.8 \pm 17.9$ at 0.1 SOL (Figure 20). Letting the clock run out instead of folding, withholding the second board of a run-it-twice hand, and stalling the hole shares after seeing a partner's cards worked as well. The program now requires a seat that leaves the hand, by folding or by timeout, to post its shares for every board position not yet open. Hands stay face down until it has. Every stall after the deal forfeits the staller's contribution and $\min(\text{stack}, k \cdot \text{pot})$, shared equally by the k seats still contesting, and its bond goes to the treasury rather than to any seat. When the staller's stack covers that penalty, a coalition ends below its worst showdown result whatever the cards. With this rule a colluding trio loses 120.4 ± 21.0 bb/100. A short or all-in staller pays only what it has, so a pair still gained $+22.8 \pm 13.0$ to $+69.0 \pm 13.8$ bb/100 in our upper-bound model. Two further rules close this. First, a seat with no decision left (folded, or all in when a card becomes due) must post its shares for every undealt board position before any in-hand share of that card is accepted, so only a seat with chips behind, which can pay the penalty, can ever see a card before choosing to stall. Second, after a stall in play the contenders reveal and a resumable crank pays every pot by exact equity over the undealt cards (one transaction on the river, three on the turn for two Hold'em contenders; before the flop, 1,024 run-outs sampled from a seed fixed by the hand's transcript, unbiased with a standard error of at most 1.6% of a pot). The staller is excluded and keeps its penalty. With an all-knowing coalition, every variant we simulated (timeout fold, in-hand loser, short or all-in staller, run it twice) has a team uplift whose 95% interval contains or lies below zero at 0.0002 to 0.1 SOL blinds, for pairs at six seats and trios at nine.

Future work. Batching the Chaum–Pedersen checks of one `post_shares` into a single MSM; applying Bayer–Groth's multi-exponentiation reduction to cut the ciphertext MSM; verifiable encryption of hole-card shares to their owner as an optional extra layer; and a formal verification of the verifier's equation arrangement.

14 Conclusion

memepoker shows that a complete, dealerless mental-poker protocol can be enforced by a Solana program within the platform's limits, with no trusted setup and no committee. Each player's key commitment, each 52-card Bayer–Groth shuffle and each decryption share is verified on chain before the game moves. In our benchmark harness a full shuffle verification fits in one transaction of about one million compute units, and a round in eight transactions; the deployed program, which passes every account inline, uses three verification transactions and 14 per round. The pieces that made this possible are mundane but decisive: the native Ristretto syscalls, one hash syscall per challenge, a scalar arithmetic tuned to the SBF virtual machine, and a verifier recast as independent identity checks over a 290-byte state. Around the cryptography sits an enforcement layer that makes stalling attributable and expensive, and makes key sharing self-defeating. And after the hand, anyone can re-check it from chain data alone, without trusting

the program's execution or the client, and trusting the RPC node only to return real chain data; the verifier does share the transcript, the proof parsers and the primary Bayer–Groth arrangement with the program, which the second line covers. Behind all of this sit more than a thousand named attacks, mutation batteries and fuzzers that keep the implementation honest about the claims made here.

References

[1] A. Shamir, R. L. Rivest, and L. M. Adleman. Mental poker. Technical Report MIT/LCS/TM-125, MIT, 1979. In *The Mathematical Gardner*, 1981.

[2] A. Barnett and N. P. Smart. Mental poker revisited. In *Cryptography and Coding*, LNCS 2898, pages 370–383, 2003.

[3] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, LNCS 7237, pages 263–280, 2012.

[4] H. de Valence, J. Grigg, M. Hamburg, I. Lovecruft, G. Tankersley, and F. Valsorda. The ristretto255 and decaf448 groups. RFC 9496, IRTF CFRG, 2023.

[5] M. Hamburg. Decaf: Eliminating cofactors through point compression. In *CRYPTO*, LNCS 9215, pages 705–723, 2015.

[6] D. J. Bernstein. Curve25519: New Diffie–Hellman speed records. In *PKC*, LNCS 3958, pages 207–228, 2006.

[7] National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS PUB 180-4, 2015.

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, pages 62–73, 1993.

[9] R. Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, CWI and University of Amsterdam, 1996.

[10] I. Damgård. On Σ-protocols. Lecture notes, Aarhus University, 2002.

[11] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, LNCS 263, pages 186–194, 1986.

[12] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the Fiat–Shamir heuristic and applications to Helios. In *ASIACRYPT*, LNCS 7658, pages 626–643, 2012.

[13] J. Miller. Coordinated disclosure of vulnerabilities affecting Girault, Bulletproofs, and PlonK (the Frozen Heart vulnerabilities). Trail of Bits blog, April 2022.

[14] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[15] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, LNCS 740, pages 89–105, 1992.

[16] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

[17] R. Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM*, LNCS 72, pages 216–226, 1979.

[18] T. Attema, S. Fehr, and M. Klooß. Fiat–Shamir transformation of multi-round interactive proofs. In *TCC*, LNCS 13747, pages 113–142, 2022.

[19] B. David, R. Dowsley, and M. Larangeira. ROYALE: A framework for universally composable card games with financial rewards and penalties enforcement. In *Financial Cryptography*, LNCS 11598, pages 282–300, 2019.

[20] B. David, R. Dowsley, and M. Larangeira. Kaleidoscope: An efficient poker protocol with payment distribution and penalty enforcement. In *Financial Cryptography*, LNCS 10957, pages 500–519, 2018.

[21] I. Bentov, R. Kumaresan, and A. Miller. Instantaneous decentralized poker. In *ASIACRYPT*, LNCS 10625, pages 410–440, 2017.

[22] S. Garg, A. Kate, P. Mukherjee, R. Sinha, and S. Sridhar. Insta-Pok3r: Real-time poker on blockchain. Cryptology ePrint Archive, Paper 2024/1061, 2024.

[23] ProofPlay. ProofPlay: verifiably fair and private shuffling for on-chain Texas Hold'em, with a Bayer–Groth proof and a zkSNARK per shuffle. ETHGlobal New York 2025 showcase, https://ethglobal.com/showcase/proofplay-kcxkk, 2025.

[24] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *Journal of Cryptology*, 15(2):75–96, 2002.

[25] M. Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *CRYPTO*, LNCS 3621, pages 152–168, 2005.

[26] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *CRYPTO*, LNCS 10992, pages 33–62, 2018.

[27] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *ACM CCS*, pages 116–125, 2001.

[28] J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In *CRYPTO*, LNCS 2139, pages 368–387, 2001.

[29] J. Groth. A verifiable secret shuffle of homomorphic encryptions. *Journal of Cryptology*, 23(4):546–579, 2010.

[30] B. Terelius and D. Wikström. Proofs of restricted shuffles. In *AFRICACRYPT*, LNCS 6055, pages 100–113, 2010.

[31] Ethereum Foundation cryptography research. Curdleproofs: A zero-knowledge shuffle argument inspired by Bayer–Groth, for Whisk. https://github.com/asn-d6/curdleproofs, 2023.

[32] Geometry Research. mental-poker: A Rust library for Barnett–Smart mental poker with a Bayer–Groth shuffle argument. https://github.com/geometryxyz/mental-poker, 2022.

[33] C. Ricketts. ziffle: Mental poker library implementing Bayer–Groth 2012 shuffle proofs. Rust crate, version 0.1.0, https://docs.rs/ziffle, October 2026.

[34] Manta Network. zkShuffle: Mental poker on SNARK for Ethereum. https://docs.manta.network/docs/zkShuffle/Overview, 2023.

[35] Zypher Network. zk-Shuffle SDK and UZKGE (Universal Zero-Knowledge Game Engine). https://github.com/zypher-game/uzkge, 2024.

[36] D. Pinones. mental-poker: Noir shuffle proofs verified on Starknet with Garaga. https://github.com/dpinones/mental-poker, 2025.

[37] Abhist17. Teexas: Trustless no-limit hold'em on Solana in a MagicBlock private ephemeral rollup. https://github.com/Abhist17/teexas, 2026.

[38] Arcium. Encrypted computation for Solana: MPC card-game examples. https://github.com/arcium-hq/examples, 2025.

[39] Panmoni. PushFlip: A Solana card game with Groth16 and Poseidon Merkle deck commitments. https://github.com/Panmoni/pushflip, 2026.

[40] Anza. The ZK ElGamal Proof program. https://docs.anza.xyz/runtime/zk-elgamal-proof, 2024.