WHITEPAPER

RAIN RNG

Provably-fair randomness for regulated and on-chain gaming — without giving up Web2 speed.

"Randomness for money must be decided by the two parties whose money it is, locked before either can act on it, and verifiable by anyone afterwards — **at the speed of a slot reel.**"

$$r_k = \text{keccak256}(pRev_k \,\|\, hRev_k \,\|\, \text{sessionSeed} \,\|\, \text{channelId} \,\|\, k)$$

VERSION 1.0 · SEPTEMBER 2026
RAIN RISK MARKETS · CANDIDATE STANDARD
PRE-AUDIT · REFERENCE CODE MIT + ATTRIBUTION

RAINRISKMARKETS.COM
POWERED BY RAIN RNG

> ***"Randomness for money must be decided by the two parties whose money it is, locked before either can act on it, and verifiable by anyone afterwards — at the speed of a slot reel."***

Abstract

Every game of chance stands on a random number generator, and today almost every RNG in the industry is a box the player cannot see into. Regulated operators run a certified server-side RNG whose *code* has been tested by a laboratory but whose *outputs* nobody outside the operator can verify. "Provably fair" crypto casinos publish a hashed server seed, but the server still knows the outcome before the player acts. On-chain verifiable random functions (VRFs) and randomness beacons are honest but slow and expensive, and they put a third party in the loop of every round.

RAIN RNG is a two-party commit-reveal randomness ceremony designed to remove all three defects at once. A house and a player each commit — once, at session open — to a long pre-computed hash chain and to one half of a session seed; both commitments are anchored (on-chain when money is involved, in a signed transcript otherwise). Every round consumes the *next* element of *both* chains, so the round's randomness

```
r_k = keccak256(pRev_k || hRev_k || sessionSeed || channelId || k)
```

is **unpredictable and unbiasable as long as either party is honest**, needs **no third party, no oracle fee and no on-chain transaction per round**, and is **publicly recomputable by anyone** from five 32-byte words. One 256-bit `r_k` per round then seeds a standards-based deterministic random bit generator (HMAC-DRBG per NIST SP 800-90A, or a ChaCha20 stream), from which every draw of the round — every reel stop, cascade, bonus pick — is derived by counter. The entire round replays bit-for-bit from `(r_k, gameId)`.

The construction has been live on Arbitrum One since 2026 in two reference deployments (playmarkets.bet and maycasino.xyz), executing complete slot spins with **measured round-trip latencies of 92–104 ms p50** and **540 of 540 independently re-derived live rounds matching**. Those are full spin round trips without pipelining; the v2.1 pipelined design targets an *added* p95 under 30 ms relative to a plain server RNG, and that figure is a target until measured (§11). Version 2.1 — specified in this paper — adds the DRBG layer, chain rotation at 65,536 rounds, and pipelined pre-commitment of round *k+1* during round *k*'s animation, with a **target** of under 30 ms added p95 latency over a plain server RNG. The reference implementation is MIT-licensed with a single attribution condition ("Powered by RAIN RNG"), so that the standard spreads on the same mechanism that makes it trustworthy: visibility.

This paper is written for two readers. An executive or regulator can read §1–2 and §11–12. A laboratory engineer or integrator should read §3–4 and §7–9, and the appendices.

1. The Problem: RNG Trust in Gaming Today

1.1 Three ways the industry produces random numbers

(a) The certified black box. A regulated operator runs a hardware or software RNG that a test laboratory has examined against a standard such as GLI-19 or the equivalent national requirements. The laboratory reviews the algorithm, runs statistical batteries, and inspects seeding and scaling. It then certifies *that build*. From that moment on, the player's assurance rests on three assumptions the player cannot check: that the certified build is the one actually running, that its outputs reach the game logic unaltered, and that the operator has no way to observe or select outcomes. Historically each of these assumptions has failed somewhere — through insider access, replaced binaries, or simply through the operator's own staff knowing the seed. Certification protects the *design*; it does not let anyone verify a *round*.

(b) "Provably fair" server seeds. Crypto casinos popularised a scheme in which the server publishes `hash(serverSeed)` before play, the player supplies a `clientSeed`, and outcomes are `hash(serverSeed || clientSeed || nonce)`. After the server seed is rotated, the player can recompute past results. This is real progress — it makes *post-hoc* misreporting detectable — but it has two structural gaps. First, **the server knows every outcome before the player acts**: it holds `serverSeed` and sees `clientSeed` as soon as it is sent. For any game with decisions inside a round (blackjack, crash cash-outs, step games), the house can adjust its behaviour to the outcome it already knows. Second, **the player only learns the truth when the server chooses to reveal**; a house can rotate, stall or "lose" a seed, and there is no penalty enforced by anything but reputation.

(c) VRFs, beacons and entropy oracles. Chainlink VRF, drand, Pyth Entropy and similar services deliver randomness that is cryptographically verifiable and independent of the operator. They are the right tool for lotteries, NFT mints and infrequent draws. For a slot machine they impose three costs the product cannot absorb: **latency** (an on-chain request/fulfil cycle takes from a second to a minute), **price** (a fee plus gas per request, on every spin) and **a new trusted party** (the oracle's liveness and honesty; a stalled fulfilment is a stalled game). None of them binds the randomness to the *bet*: the oracle does not know or care whether the request was made before or after the player committed money.

1.2 What is actually being asked of an RNG for money

Strip away the mechanisms and the requirement is simple: the number must be fixed **after** both parties are committed and **before** either can react to it, it must be **influenceable by neither** party alone, and any third person must be able to **check** afterwards that this is what happened. Everything else — statistical quality, certification, speed — is necessary but not sufficient.

The industry's three approaches each solve part of this. RAIN RNG's thesis is that the whole requirement is satisfiable with nothing more exotic than hash functions, provided the protocol is built around the *order of events* rather than around the *source of entropy*.

1.3 Why speed is not negotiable

An RNG that is fair but slow does not become a standard; it becomes a footnote. Casino games are judged on feel: a spin that answers in under about 150 ms feels instantaneous, a spin that takes 500 ms feels laggy, a spin that takes 3 seconds is a different product. Operators who have built successful Web2 games will not adopt a fairness layer that costs them the feel of the game. This paper therefore treats **zero perceived added latency** as a design constraint equal in rank to unpredictability, not as an optimisation to be attempted later (§5).

2. Design Goals

RAIN RNG is specified against seven goals. Each is testable, and §11 states for each whether it is measured today or a target for v2.1.

#	Goal	Meaning	Where it is met
G1	**Unpredictable**	No party — house, player, or observer — can compute `r_k` before both reveals for round *k* exist.	§3.3
G2	**Unbiasable by any single party**	Neither party can influence the distribution of `r_k` by choice of its own inputs, given the other's commitments. Formally: `r_k` is uniform if *either* party is honest.	§3.4
G3	**Publicly verifiable per round**	Any third party recomputes any round from public data and a keccak256 tool; on-chain anchors make the commitments undeniable.	§7
G4	**Latency a player cannot perceive**	Design goal: the ceremony must hide inside the reel animation. **Measured today (v2.0, un-pipelined): full spin round trip p50 92–104 ms, p95 ≤ 176 ms.** Target for v2.1 pipelining: added p95 < 30 ms over a plain server RNG — not yet measured.	§5, §11
G5	**Zero per-draw cost**	No oracle fee, no gas, no third-party call per round. Chain interaction only at session open/close and in disputes.	§3, §6
G6	**Certifiable**	The randomness layer maps onto existing laboratory frameworks (GLI-19, SP 800-90A, SP 800-22) and can be certified once, independently of game math.	§9
G7	**Attributable**	Every public deployment identifies itself, so that the standard's adoption is visible and auditable.	§10

Two non-goals are stated explicitly. RAIN RNG does **not** guarantee statistical return-to-player — that is a property of the published payout table and the law of large numbers, not of randomness. And it does **not** make collusion between the two parties impossible: when both the house and the player agree to cheat, there is no one left to cheat in a two-party game (the money layer separately prevents them extracting more than they deposited — §8.1).

3. The RAIN RNG Construction

3.1 Primitives and notation

All hashes are `keccak256` over ABI-encoded static 32-byte words (so that the TypeScript, Python and Solidity implementations are byte-identical). `H(a, b, …)` denotes `keccak256(abi.encode(a, b, …))`. Hex values are 32 bytes unless stated.

Symbol	Meaning
`houseSeed`, `playerSeed`	The two halves of the session seed (32 random bytes each)
`houseSeedCommit = H(houseSeed)`	The house's commitment, published *before* it learns `playerSeed`
`channelId` / `sessionId`	Unique session identifier (on-chain: `H(manager, player, operator, openNonce, chainId)`)
`sessionSeed = H(houseSeed, playerSeed, channelId)`	Fixed at open; binds both halves and the session
`c_h[·]`, `c_p[·]`	The house and player hash chains
`houseChainRoot = c_h[0]`, `playerChainRoot = c_p[0]`	Chain roots, committed at open
`hRev_k = c_h[k]`, `pRev_k = c_p[k]`	The reveals consumed by round *k*
`r_k`	The round's 256-bit randomness

3.2 Hash chains — a pre-committed list of secrets

Each party derives a chain of length *n* from a secret:

```
c[n]   = keccak256(secret)              // raw 32 bytes
c[i-1] = keccak256(abi.encode(c[i]))    // i = n … 1
root   = c[0]
```

The root is committed. The *k*-th reveal is `c[k]`, and the one-step rule `keccak256(abi.encode(c[k])) == c[k-1]` lets a verifier check each reveal against the previous one in O(1) — or against the root in O(k) by hashing *k* times. Because keccak256 is preimage-resistant, a party that has committed to `c[0]` has exactly one valid value it can produce at each step. **It has no choice.** This is the property that makes the house's contribution to every round a fixed, pre-existing fact rather than a fresh pick.

The v1 reference deployments use *n* = 4,096 (4,095 usable rounds per session). Version 2.1 raises the default to **n = 65,536** and rotates chains transparently (§5.4); computing a 65,536-element keccak chain takes tens of milliseconds in a browser and is done once per session.

3.3 The ceremony

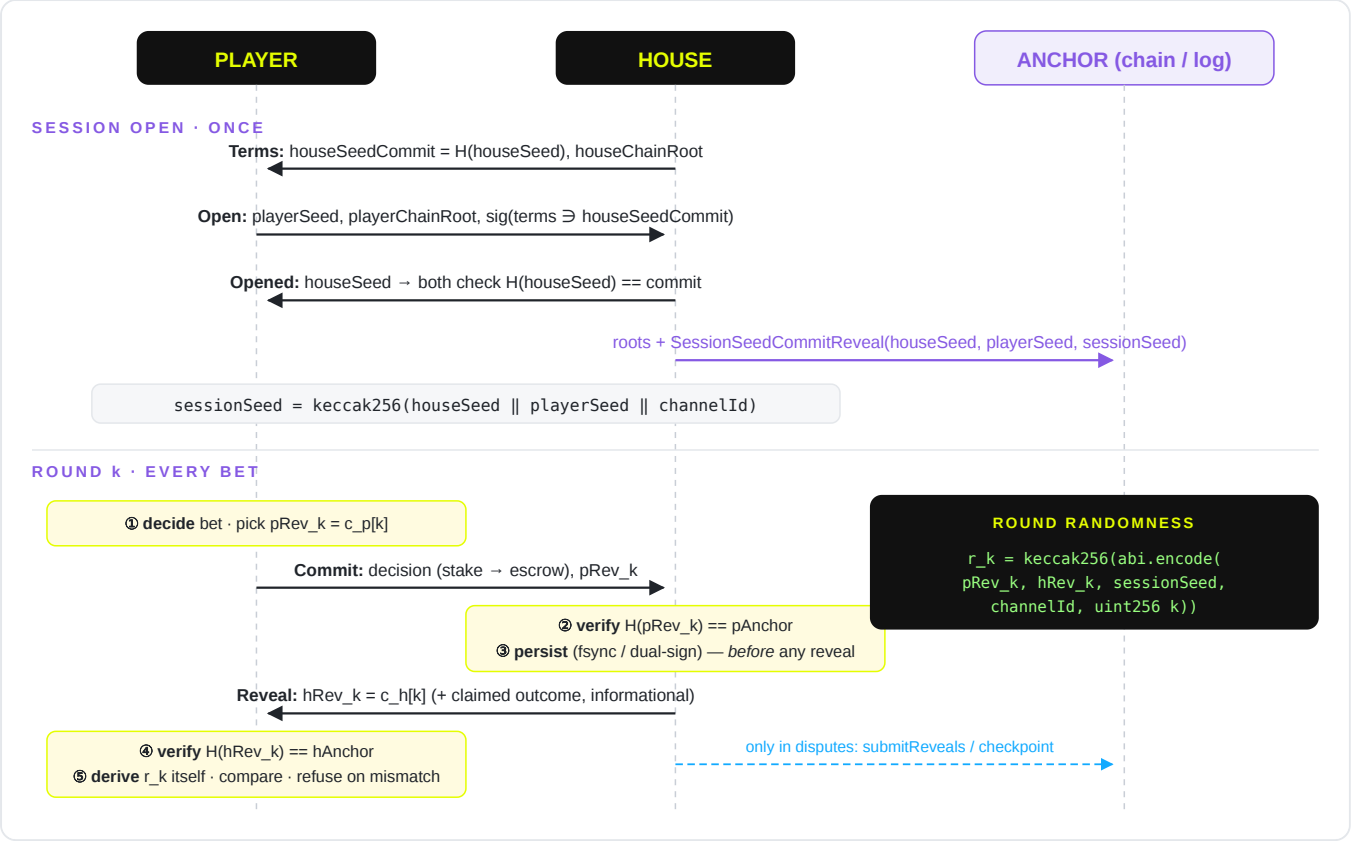

Figure 1 — The RAIN RNG ceremony: one open, then a two-message round.

Session open (once).

1. The house picks `houseSeed` and a chain secret; publishes `houseSeedCommit`, `houseChainRoot` and the session terms.
2. The player derives `playerSeed` and its own chain (from wallet material — §3.6 — or from fresh randomness), and returns `playerSeed`, `playerChainRoot`, and a signature over the terms *including* `houseSeedCommit`. The house therefore cannot change its commitment after seeing the player's seed.
3. The house reveals `houseSeed`. Both sides check `H(houseSeed) == houseSeedCommit` and compute `sessionSeed = H(houseSeed, playerSeed, channelId)`. In the on-chain topology this happens inside `ChannelManagerAA.openChannel`, which stores `sessionSeed` and both chain roots and emits `SessionSeedCommitReveal(channelId, houseSeed, playerSeed, sessionSeed)`.

Round *k* (every bet; two messages).

1. **Commit.** The player fixes its *decision* (the bet, the move) and sends it together with `pRev_k`. In the money topology the decision is a dual-signable state that moves the stake into escrow and pins both parties' current anchors (`pAnchor = pRev_{k-1}`, `hAnchor = hRev_{k-1}`, or the roots when *k* = 1).
2. **Reveal.** The house checks `H(pRev_k) == pAnchor`, **persists the player's commitment durably**, and only then answers with `hRev_k`. It may attach what it believes the outcome is; the client never adopts that value blindly.
3. **Settle.** The player checks `H(hRev_k) == hAnchor`, computes

```
r_k = keccak256(abi.encode(pRev_k, hRev_k, sessionSeed, channelId, uint256 k))
```

itself, derives the game outcome (§4), and compares with the house's claim. A mismatch is refused — and, in the money topology, becomes a provable house fault.

Two ordering rules carry the entire security argument and are enforced in code by the reference implementation (`@rain/rng-session`):

- **Decision-before-reveal (client):** the player's bet and `pRev_k` leave the client before it has seen `hRev_k`.
- **Persist-before-reveal (house):** `hRev_k` leaves the house only after the player's commitment is durably stored, so that a house that later "did not receive" a winning bet can be contradicted by its own log — or, on-chain, by the dual-signed state the player holds.

3.4 Why *r* is random if *either* party is honest

Model keccak256 as a random oracle. `r_k` is the hash of a message that contains `hRev_k` and `pRev_k`.

- **Suppose the house is honest.** Its chain secret was drawn uniformly and its chain root committed before the session; `hRev_k` is a fixed, high-entropy value unknown to the player until the reveal message. However the player chooses `pRev_k` — it cannot, because its own chain is committed; but grant it the freedom — the player must fix `pRev_k` *before* seeing `hRev_k`. The message hashed therefore contains a uniform component the player never saw when it committed. Under the random-oracle assumption `r_k` is uniform and independent of everything the player knew. The player can neither predict it nor bias it.
- **Suppose the player is honest.** Symmetric: `pRev_k` is uniform from the house's point of view until the commit message arrives — and by then the house's own `hRev_k` has already been fixed by its committed chain. The house's only remaining "choice" is whether to answer at all, which is observable and, with money involved, punishable (§8.1).
- **Suppose both are dishonest.** They can agree on any `r_k` they like. There is no victim: a two-party game with both parties colluding is a private agreement, not a fraud.

Notice what the argument does *not* need: a trusted third party, a clock, a blockchain per round, or any assumption about the *other* party's randomness. That is the sense in which RAIN RNG is "trustless" — each party trusts only itself.

3.5 Domain separation and the role of `sessionSeed`, `channelId`, `k`

Folding `sessionSeed`, `channelId` and `k` into the hash is defence in depth rather than the source of security. `channelId` prevents a reveal pair being reinterpreted in another session; `k` prevents a round's randomness being reused at another index even if a chain element were somehow repeated; `sessionSeed` adds a session-wide two-party value so that even a full compromise of one party's chain secret gives no information about outcomes without the other party's reveals. Per-game derivations extend the message with a game-domain word (§4.2) so that a dice round and a slot round at the same *k* never share randomness.

3.6 Player material without backups

A player using a wallet signs one message, once: `"RAIN session key v1 for <manager> chainId <id>"`. Because ECDSA signatures in wallets are deterministic (RFC 6979), the same wallet always produces the same signature, and from it the client derives `base = keccak256(sig)`, a burner session key `keccak256(base || "session")`, and per-session `chainSecret = keccak256(base || "chain:<openNonce>")` and `playerSeed = keccak256(base || "seed:<openNonce>")`. The player can re-derive its chains on any device without storing anything, and the house never sees `base`. Players without a wallet (the RNG-as-a-service topology, §6b) simply draw 32 random bytes from the platform CSPRNG.

4. From *r* to a Game: One Seed, a DRBG, Counter-Derived Draws

4.1 The design decision

A single round of a modern slot needs hundreds or thousands of random draws: reel positions, cascade refills, wild placements, bonus picks, multiplier choices. The question is how to turn one 256-bit `r_k` into that stream. RAIN RNG's answer in v2.1 is:

One `r_k` per round → one standards-based DRBG instance seeded from it → every draw of the round read from that DRBG by counter.

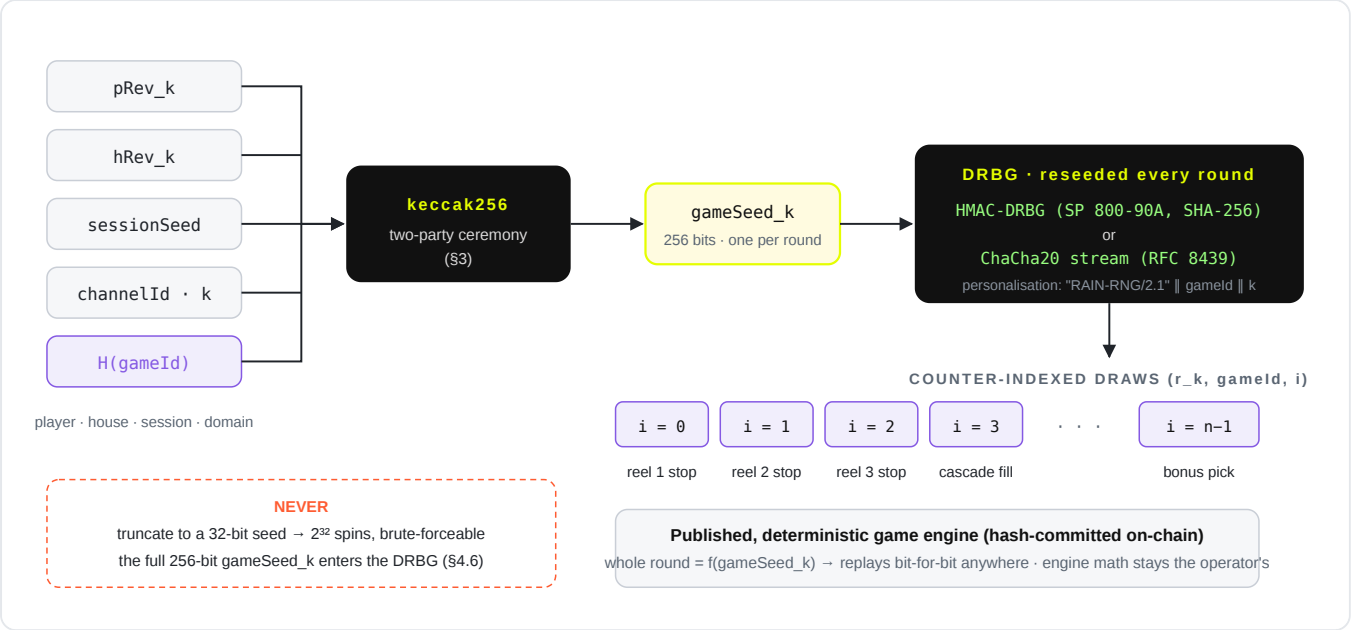

Figure 2 — From the two reveals to every draw of a round.

The reference implementation offers two interchangeable DRBG back-ends, both seeded identically:

- **HMAC-DRBG (SHA-256)**, exactly as specified in NIST SP 800-90A §10.1.2. It is the mechanism laboratories already know how to certify, has published test vectors, and is implementable in a few dozen lines in any language.
- **ChaCha20 stream** (RFC 8439), keyed from `r_k` and the domain string via an HKDF-style expansion, block counter = draw block index. It is faster in constrained environments and gives O(1) random access to the *i*-th draw, which matters for parallel verification.

Both are deterministic functions of `(r_k, gameId, k)`. The exact byte layout of the seed material, personalisation string and draw encoding is normative in the `@rain/rng-core` 2.1 specification and fixed by test vectors shared between the TypeScript, Python and Solidity references (Appendix A); the same inputs must produce the same bytes in every implementation.

4.2 Domain separation per game

The seed material for the DRBG is not `r_k` alone but a game-domain seed:

```
gameSeed_k = keccak256(abi.encode(pRev_k, hRev_k, sessionSeed, channelId, uint256 k, keccak256(bytes(gameId))))
```

This is the formula the live `EngineV2RulesVerifier.seedOf` exposes on Arbitrum. Two consequences: a round of "foundry" and a round of "quarry" at the same cursor are computationally independent, and an operator adding a new game gets a new randomness domain without touching the ceremony. Draws inside a round are then indexed `(gameSeed_k, i)` for *i* = 0, 1, 2, …; a draw is identified by three values `(r_k, gameId, i)` and nothing else.

4.3 Uniform integers, floats and reductions

From the DRBG's byte stream the reference defines `nextU32()`, `nextU64()`, `nextFloat53()` (IEEE double in [0,1) from 53 bits), and `nextInt(N)`. For simple verifier-checked games the reduction is the plain `uint256(r) % N` that the Solidity verifiers use (dice N = 100, roulette N = 37); its bias is below N / 2^256, unmeasurable for any N a game would use, and it must stay a modulo to remain byte-identical to the on-chain code. For engine-driven games with many draws, `nextInt(N)` uses 64-bit draws with modulo, giving bias below N / 2^64 — again far below anything a statistical battery can detect — and the Python and TypeScript mirrors are tested to agree on every draw.

4.4 Full replay

Because every draw is a pure function of `(r_k, gameId, i)` and the game engine is a published, deterministic program, an entire round — base spin, every cascade, every bonus — replays from **five public 32-byte words and a game identifier**. The live Engine V2 deployment commits to `(seed, gameId, capFp, winFp, resultHash)` per round; an independent tool (`tools/engine-v2-audit.mjs`) re-derived **540 of 540** live rounds from the house node's public log in September 2026, checking seed, chain anchors, win amount, result-tree hash and payout on each, with zero mismatches.

4.5 Why not keccak-per-draw?

The v1 engine derives each block of eight 32-bit draws as `keccak256(seed || gameIdHash || round || blockIndex)`. It is secure and it works — the 540/540 audit ran against it. Version 2.1 replaces it with a DRBG for three practical reasons, none of them a weakness in keccak:

1. **Certifiability.** A laboratory can certify "HMAC-DRBG per SP 800-90A seeded with 256 bits of two-party entropy" by reference to a document it already has. A bespoke keccak counter mode has to be analysed from scratch every time.
2. **Portability.** The Python mirror, the Solidity verifier and any operator's Go or Rust service all have vetted HMAC-SHA256 and ChaCha20 libraries. Getting a keccak sponge byte-identical across five languages is possible but is a recurring source of integration bugs.
3. **Cost.** A cascading slot can consume thousands of draws per spin; ChaCha20 produces them at a fraction of the cost of a keccak permutation per 32 bytes, which matters when the same replay runs in the player's browser, in the house node, and in the verifier.

4.6 Why not a 32-bit seed?

Many existing slot engines expose a "seed the RNG with an integer" hook that takes 32 bits. It is tempting to feed the low 32 bits of `r_k` into such an engine for compatibility. RAIN RNG forbids it, and the reason is worth stating precisely, because the two-party ceremony does *not* rescue it.

- **Entropy collapse.** A round seeded from 32 bits has at most 2^32 ≈ 4.3 billion possible outcomes, regardless of how good the engine is. Every possible spin of that game can be enumerated in advance on a laptop in minutes and stored in a table. The two-party ceremony still guarantees that *which* of the 4.3 billion spins occurs is unbiasable — but the *set* of reachable spins is now tiny and fully known, and any correlation between the seed bits and, say, the bonus trigger is exploitable by whoever studies the table.
- **Collisions.** By the birthday bound, after about 2^16 = 65,536 rounds a session has better-than-even odds of *repeating* a complete spin. Repeated spins are exactly what a statistical battery — and a sharp player — will notice.

- **Partial leaks become total leaks.** If any 32 bits of the seed leak (a log line, a debug panel), the entire round is known. With a 256-bit seed and a DRBG, leaking 32 bits leaks nothing usable.
- **Laboratory requirements.** GLI-19 and its peers require that the RNG's seed space and internal state be large enough that outcomes cannot be predicted by exhaustive search. A 32-bit seed fails that clause on its face.

The rule is therefore: **the full 256-bit `gameSeed_k` enters the DRBG; nothing narrower ever seeds a round.** Legacy engines are adapted by replacing their integer-seeded generator with the RAIN draw interface, not by truncating the seed.

4.7 Per-round reseeding as containment

A useful side effect of "one seed per round" is that the blast radius of any implementation fault is a single round. If a DRBG state were ever exposed — a memory dump, a bug that logged internal state — the exposure ends when the round ends, because round *k+1* is reseeded from `r_{k+1}`, which depends on reveals that did not exist yet. There is no long-lived RNG state on either side to protect, and no "RNG compromise" incident class that extends beyond one spin.

5. Web2 UX: Latency Budget — Measured Today, Target for v2.1

5.1 The latency budget

A spin's wall-clock time as seen by the player is:

```
T_spin = T_input + T_network(commit→reveal) + T_house + T_derive + T_animation
```

`T_animation` is the reel spin the game designer wants — typically 800–2,500 ms. `T_derive` is the local replay of the round, measured at **≤ 14 ms** in-browser for the live Engine V2 games. `T_house` is the house node's verify-persist-reveal path, measured at **11 ms p50** in-process. What remains is `T_network`, which is one round trip plus whatever the transport adds — and this is where the reference deployments' measured **92–104 ms p50 / 99–176 ms p95** per settled spin (browser → house node → browser, including signing and state handling, on a public testnet-grade deployment) comes from.

Compared with a plain server RNG, the ceremony adds exactly one thing: the house must wait for the player's commitment before it can reveal. In a request/response design that wait is *already* the request. The additional work — one keccak to check `pRev`, one durable write, one keccak to check `hRev` — is microseconds to low milliseconds. The **target for v2.1 is that the ceremony adds < 30 ms at p95** over the same game on the same network with a server-side RNG; this figure is a target, not yet a measurement (§11).

5.2 Pipelining: commit *k+1* during *k*'s animation

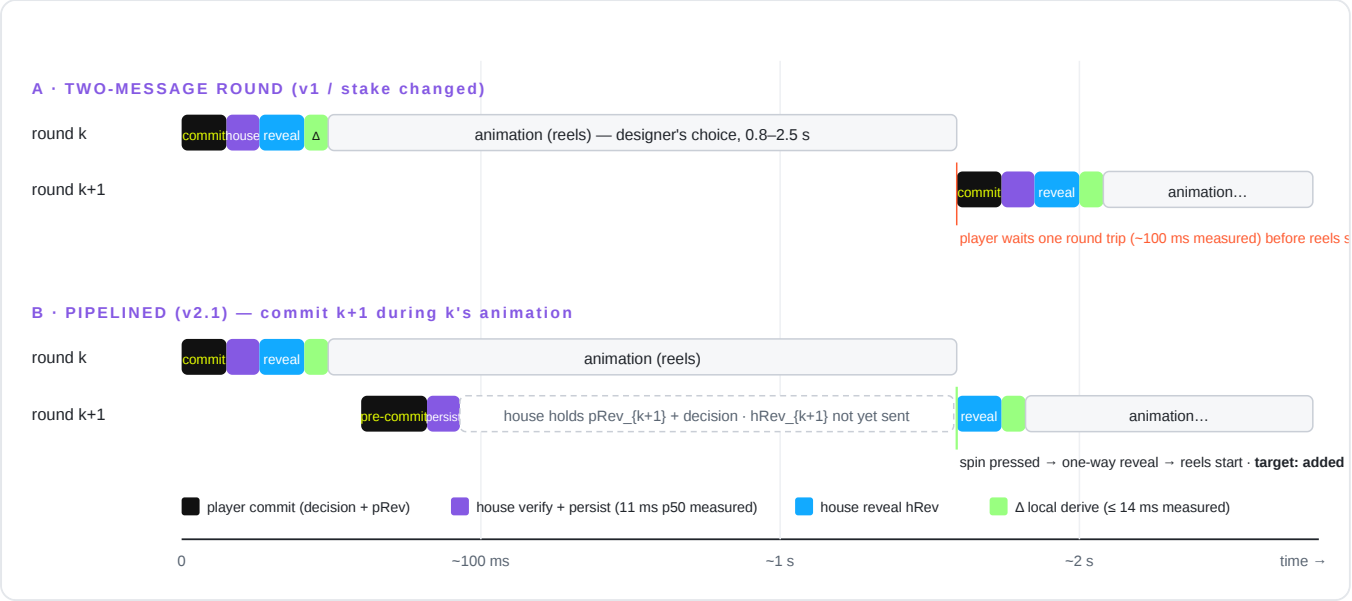

Figure 3 — Pipelined rounds: the ceremony for k+1 hides inside k's animation.

The player's *bet* for round *k+1* cannot be decided before the player decides — but the *transport* can be. Version 2.1 lets the client pre-open the next round the moment round *k* settles: it sends `pRev_{k+1}` bound to a **decision placeholder** (a commitment to "the next bet, whatever its size, at default stake" or simply a pre-signed state for the common auto-spin case) while round *k*'s reels are still spinning. When the player presses spin, the house already holds the player's commitment and answers with `hRev_{k+1}` in one direction of the network, not two. For auto-spin and turbo modes — where the *decision* is fixed in advance — the network cost of the ceremony disappears into the animation entirely.

The security argument is unchanged: the player still commits before seeing `hRev_{k+1}`; the house still persists before revealing. What moves is *when* the commit is sent, not *what* it binds. Where the decision genuinely changes at the last moment (a different stake), the client falls back to the two-message path for that round, which costs one round trip — the same as today.

5.3 What the player sees

Nothing. The fairness panel shows `VERIFIED` when the local recomputation matches the house's claim, and the game plays exactly as a Web2 slot does. There is no wallet pop-up per spin, no "waiting for randomness" spinner, no gas prompt. In the on-chain money topology the live deployments record **zero smart-account transactions during play** across hundreds of spins; the chain is touched at login and at cash-out.

5.4 Chain rotation at 65,536

A session's chain is finite. In v1, 4,095 rounds at turbo speed could exhaust a chain in under two hours of continuous auto-spin, forcing a mid-session reopen. Version 2.1 sets the default chain length to **65,536** and, more importantly, **rotates chains without a visible break**: when the cursor passes a high-water mark (default 90 %), the client and house exchange fresh chain roots *inside* a normal round message, signed like any other state, and the next round after the exchange consumes element 1 of the new chains. In the money topology the new roots are pinned in the dual-signed state and can be checkpointed on-chain; in the transcript topology they are simply part of the signed log. A 65,536-chain is 32 bytes × 65,536 = 2 MiB per party if fully materialised; the reference stores it as a secret plus a cached window and recomputes forward as needed.

6. Deployment Topologies

Figure 4 — Four ways to deploy the same ceremony.

The ceremony is the same in every topology; what changes is where commitments are anchored and whether money rides alongside.

(a) Player ↔ RAIN node — the casino. The player's browser runs `@rain/rng-core`; the operator runs a RAIN house node. The browser holds the player's chain, verifies every reveal and recomputes every outcome. Anchoring is by signed transcript, optionally hashed to a public chain at session end. This is the minimum deployment for a Web2 casino that wants per-round verifiability without any blockchain in the player's path.

(b) Operator server ↔ RAIN node — RNG-as-a-service. The operator's existing game server plays the role of "player": it commits a decision and `pRev` per round to a RAIN node run by RAIN or a certified third party, receives `hRev`, and derives the round. The operator keeps its game math, its front-end and its player accounts unchanged; what it gains is that its RNG is no longer a box it alone controls — every round is a two-party fact, and the RAIN node publishes commitments and a `/verify` endpoint (§7.2). This is the builder's case in practice: a studio with a working slot who wants a certifiable, attributable RNG layer in a week, not a re-platform.

(c) Operator-run node with on-chain anchoring. The operator runs the node itself (`@rain/rng-node`: WAL persist-before-reveal, high availability via a shared store, `/verify`, `/healthz`, `/metrics`) and publishes its session commitments to an `RngAnchor` contract on any EVM chain (≈ 0.9 M gas to deploy; a few tens of thousands of gas per session commit). Players — or the operator's own compliance function — can prove after the fact that the roots existed before play began. No money moves on-chain.

(d) Inside a state channel with money — the full protocol. The ceremony runs inside RAIN Channels on `ChannelManagerAA`: the same message that commits the bet moves the stake into escrow, the same contract that stores the chain roots enforces the money rules, and a refused or mis-reported round becomes a fault with compensation from a house bond. Fair randomness and fair payment are one system. This is what runs on playmarkets.bet and maycasino.xyz and is specified in the RAIN Risk Markets protocol paper; this whitepaper covers only the randomness layer.

	(a) Casino	(b) RNG-as-a-service	(c) Own node + anchor	(d) State channel
Who is "player" in the ceremony	End-user browser	Operator game server	End-user browser	End-user (session key)
Who runs the house node	Operator	RAIN / certified host	Operator	Operator
Anchoring	Signed transcript	Node publishes commitments; `/verify`	`RngAnchor` on any EVM chain	`ChannelManagerAA` on Arbitrum
Money in protocol	No	No	No	Yes (escrow, faults, bond)
Chain txs per round	0	0	0	0
Integration effort	FE SDK	Server SDK, no FE change	Node + contract deploy	Full protocol

7. Verification and Audit

7.1 Player post-hoc verification

Any settled round exposes `channelId`, `sessionSeed`, `k`, `pRev`, `hRev` (and `gameId` for engine games). With any keccak256 tool:

1. `r = keccak256(abi.encode(pRev, hRev, sessionSeed, channelId, k))` — five 32-byte words, `k` as uint256.
2. Dice: `roll = r % 100 + 1`. Roulette: `n = r % 37`. Engine games: `gameSeed` with the sixth word `keccak256(bytes(gameId))`, then run the published engine.
3. Chain check: `keccak256(abi.encode(hRev_k)) == hRev_{k-1}` (or `== houseChainRoot` at $k = 1$; or hash k times to reach the root).
4. Session seed: `keccak256(abi.encode(houseSeed, playerSeed, channelId)) == sessionSeed`, and `keccak256(abi.encode(houseSeed)) == houseSeedCommit`.

With the SDK: `verifyRound({ pRev, hRev, sessionSeed, channelId, k, N, expected, hAnchor }).ok`.

7.2 Public `/verify`

Every RAIN node in topologies (b) and (c) exposes an unauthenticated `GET /verify?session=…&k=…` that returns the round's public fields, the node's committed roots, and the recomputed outcome — and, once the session is closed, the node's `houseSeed` and chain secret so that the whole session can be re-derived offline. The endpoint is stateless with respect to trust: it *asserts nothing*; it hands the verifier the inputs and lets them compute. Its value is discoverability and a single URL an auditor or regulator can script against.

7.3 On-chain anchors

In topology (d), every anchor is on Arbitrum One:

Anchor	Where	What it proves
`SessionSeedCommitReveal(channelId, houseSeed, playerSeed, sessionSeed)`	event in the `openChannel` tx	both seed halves and the derived session seed existed before any round
`channels(id).playerChainRoot / houseChainRoot`	storage	the chain roots each party committed to

Anchor	Where	What it proves
`channels(id).lastRevealP / cursorP / lastRevealH / cursorH`	storage	reveals forced on-chain via `submitReveals` during disputes
`Checkpointed(channelId, nonce, stateHash)`	event	a dual-signed state (including current anchors) as a dispute floor
`EngineV2RulesVerifier.seedOf(...)`	pure function	the per-game seed formula, callable by anyone
`EngineV2RulesVerifier.engineHash(gameIdHash)`	view	`keccak256(engineBundle || config)` the house committed to for a game

In topology (c), `RngAnchor` provides the same commitments (`Committed`, `PlayerBound`, `SeedRevealed`, `RevealsSubmitted` events; `seedOf`, `outcomeOf`, `gameSeedOf`, `verifyChainElement` pure functions) without any money logic.

A live example. Session open on ChannelManagerAA `0x60743006c2a5Dd5b9907CA37e7381854ff860959`, transaction `0x1c62012f3e449da8a6c4b2b6d9a367b343bea8e17c566cebe279b0b55930ba9e`, block 504,537,592 (2026-09-12 22:31:14 UTC):

```
channelId    0xc6fcfe25563701b9e9ec1167f6ac54ad826e9c07aa0d461b39bb358be4854f0d
houseSeed    0xa52aa98608487152568524baf8212b65a852159cc1de7ede77de43ac3b6daa13
playerSeed   0x8c3d4d13e80e659be1e9617c93ec46d49b992540088c0d6f0da1b12288301c30
sessionSeed  0x0ae271400c5eb64c6e8226f6cddbb0736b483120e4e85db1c7fc5b10588bffbd   (emitted)

keccak256(abi.encode(houseSeed, playerSeed, channelId))
           = 0x0ae271400c5eb64c6e8226f6cddbb0736b483120e4e85db1c7fc5b10588bffbd   ✓ matches
keccak256(abi.encode(houseSeed))
           = 0x5e935d70097cca9bed5d4a890846a31b8fdd5697d249fbfb92db2084326d62de   = houseSeedCommit sign
ed at open
```

7.4 Replaying an entire round from `(r, gameId, i)`

An auditor with the engine bundle (hash-committed on-chain) and the five public words reproduces the round draw by draw. Because draws are counter-indexed, a dispute about "draw 173 of round 4,410" is a dispute about a single, addressable value with a single correct answer, computable by anyone. Statistical audits become trivial: dump `r_k` for every round of a period, replay, and run whatever battery the regulator prefers on the actual draws the players received — not on a lab sample of the algorithm.

8. Security Analysis

8.1 Threat model

Adversary	Attack	Why it fails
House	Choose a favourable `hRev_k`	`hRev_k` is pinned by `keccak(hRev_k) == hRev_{k-1}` back to a root committed before play; there is exactly one valid value.
House	Swap `houseSeed` after seeing `playerSeed`	The player's signature over the terms covers `houseSeedCommit`; the open fails if the reveal does not match.
House	Grind `houseSeed` / chain secret at open to bias the session	Every `r_k` also depends on `pRev_k`, unknown to the house at commit time; no choice of house material shifts the distribution. The only freedom is *not opening*, which is observable and gains nothing.
House	See the outcome, then refuse a losing round	Persist-before-reveal: the player's commitment is durable before `hRev` leaves. In topology (d) the bet state is dual-signed; stalling → `forceClose` → `HouseFault`, stake voided plus compensation `max(2 × grossAtRisk, 5 units)` from the house bond, equal penalty burned. In (a)–(c), the refusal is recorded against the node's own WAL and `/verify`.
House	Misreport the result of a correct `r_k`	The client re-derives from the public engine and refuses; misreports are provable from public data. Hash-committed engines make the bundle itself undeniable.
Player	Choose a favourable `pRev_k`	Symmetric chain pinning.
Player	See `hRev_k`, then refuse to accept a losing round	Decision-before-reveal: the bet was committed first. In (d) the house holds the dual-signed bet with the stake in escrow; after `PLAYER_GRACE` (180 s) it settles the loss on-chain without the player's cooperation.
Player	Abort a round after committing and reuse `k`	`k` is burned: the next commit must use `k+1` and `pRev_k` is now public; a chain element is never consumed twice.
Both collude	Produce a chosen `r_k`	Possible and harmless: no third party's money or outcome depends on it. In (d) the conservation invariant `playerBal + houseBal + fee + grossAtRisk = deposit + allocation` is enforced on every on-chain state, so collusion cannot extract more than was deposited.
Third party	Predict, front-run or influence a round	Rounds are off-chain and need both secrets; there is no mempool exposure and no oracle to bribe. On-chain surfaces are open/close/dispute only.
State leak	Exfiltrate DRBG internal state or a chain window	Contained to one round: each round is reseeded from `r_{k+1}`, which does not exist until both reveals do. A leaked chain window of one party reveals nothing about outcomes without the other party's reveals.
Chain exhaustion	Force a mid-session break or reuse	Rotation at a high-water mark (§5.4); the reference refuses `k > n` and never wraps.
Clock / replay	Replay an old commit or reveal, or across sessions	`channelId` / `sessionId` and `k` are inside every hash and every signed state; in (d) the EIP-712 domain also binds `chainId` and the manager address. No wall clock is part of the randomness.
Griefing	Open sessions and never play; demand reveals to stall	Opening costs the griefer its own commitment work; in (d) `demandReveal` gives the other party 60 s and silence is a fault for the silent side, and `reclaimStale` recovers escrow after 30 days of mutual silence.

8.2 Comparison with alternatives

	Server seed + client seed ("Stake-style")	Chainlink VRF	drand (League of Entropy)	Pyth Entropy	
Trust assumptions	Server honest about seed rotation; server *knows* outcome before player acts	Oracle node honest and live; VRF key secure	Threshold of beacon operators honest; beacon live	Provider honest and live; commit-reveal with provider	**Neither party alone; no third party**
Who knows the outcome before the bet is final	The server	Nobody (if oracle honest) — but request timing is unbound to the bet	Nobody — but beacon rounds are public and periodic	Nobody (if provider honest)	**Nobody** — requires the player's not-yet-sent reveal
Latency per round	~0 (local hash)	Seconds to minutes (on-chain request + fulfil)	~3–30 s beacon period + fetch	Seconds (on-chain reveal)	**One network round trip; target < 30 ms added; ~100 ms measured end-to-end**
Cost per round	0	Oracle fee + gas	0 (public beacon) but gas if consumed on-chain	Fee + gas	**0**
Per-round public verifiability	Only after server seed rotation, and only if it was not swapped	Yes (VRF proof on-chain)	Yes (beacon signature)	Yes (on-chain reveal)	**Yes — five public words + keccak; anchors on-chain or `/verify`**
Selective abort by the house	Trivial (drop the request)	Possible (don't request / don't consume)	Possible (ignore the beacon round)	Possible (don't reveal)	**Recorded against persisted commitment; a fault with penalty in (d)**
Fit for high-frequency games	Yes	No	No	Marginal	**Yes — designed for it**
Lab certifiability of the RNG layer	Bespoke	Not a gaming-lab construct	Not a gaming-lab construct	Not a gaming-lab construct	**Maps to SP 800-90A DRBG + documented entropy input; §9**

VRFs and beacons remain the right choice when there is *no* counterparty — a public lottery draw, a fair ordering, a random airdrop. RAIN RNG addresses the case that dominates gaming volume: a house and a player, thousands of rounds, money on every one.

8.3 Assumptions and residual risks, stated plainly

- Keccak256 is preimage- and collision-resistant and behaves as a random oracle for the derivation. All fairness claims rest on this.
- Each party's own randomness source (Web Crypto, OS CSPRNG, or wallet-signature derivation) is sound *for that party*. A party with a broken CSPRNG harms only itself.
- The client software the player runs actually performs the checks. A player using a malicious client that signs whatever the house says has forfeited the protection the protocol offers — the same is true of every fairness scheme. Open-source reference clients and the `/verify` endpoint exist so that this can be checked by anyone.
- No external security audit of the contracts or the reference implementation has been completed at the time of writing (§11).

9. Standards and Certification Path

9.1 What a laboratory would certify

The proposition to a test laboratory is deliberately narrow: **certify the RAIN RNG layer once**, as a randomness source with a documented entropy input and an approved DRBG, and let each operator's game math be certified separately against that layer — exactly as game math is certified today against a hardware RNG. The layer's boundary is:

- **Input:** two 256-bit values from independent parties, each pre-committed, combined by keccak256 into `r_k` (§3).
- **Conditioning / expansion:** HMAC-DRBG (SP 800-90A) or ChaCha20 stream, seeded with `gameSeed_k`, domain-separated per game and per round (§4).
- **Output:** counter-indexed draws with defined scaling functions (`nextU32` , `nextU64` , `nextFloat53` , `nextInt(N)`).
- **Reseed policy:** every round, unconditionally.

9.2 Mapping to GLI-19 RNG requirements

GLI-19 area (RNG)	RAIN RNG v2.1
Unpredictability / cryptographic strength	256-bit `r_k` from two independent pre-committed inputs; approved DRBG expansion; no party or observer can compute `r_k` before both reveals exist.
Seeding and reseeding	Seed = `gameSeed_k` (256 bits) per round; reseeded every round; seed provenance is auditable (chain roots, anchors). No 32-bit or narrower seed path exists.
Statistical randomness of output	Output is DRBG output; SP 800-22 battery is run on the reference implementation's draws (§9.3) and can be re-run on *actual production draws* via replay (§7.4).
Scaling / mapping to game outcomes	Defined functions; modulo bias bounded (< N / 2^64 for engine draws, < N / 2^256 for verifier reductions); documented and testable.
No influence by game state or by the operator	Draws depend only on `(r_k, gameId, i)`; game state cannot feed back into randomness; the operator's contribution is pinned before play.
Background cycling / prediction resistance	Not applicable — there is no long-lived state to cycle; each round is a fresh instance.
Integrity of the RNG software	Reference implementations are open source with published test vectors; engine bundles are hash-committed on-chain in the reference deployments.
Failure handling	A missing or invalid reveal is a detected fault, never a silently substituted number; the round does not settle.

This table is a design mapping prepared by the authors, not a laboratory's finding. Formal evaluation is on the roadmap (§12).

9.3 SP 800-90A and SP 800-22

The HMAC-DRBG back-end follows SP 800-90A §10.1.2 instantiate / generate / reseed with SHA-256, and is checked against the NIST CAVP known-answer vectors in the reference test suite. Statistical testing uses the SP 800-22 battery (frequency, block frequency, runs, longest run, rank, DFT, non-overlapping and overlapping templates, universal, linear complexity, serial, approximate entropy, cumulative sums, random excursions) on DRBG output from many independent `r_k`. Because the two-party input is itself the output of keccak256 over high-entropy words, the same battery can be run on the sequence of `r_k` values across a session to demonstrate that the ceremony does not introduce structure.

9.4 What stays the operator's

Game math — payout tables, RTP, volatility, hit frequency — is entirely outside the RNG layer and remains the operator's responsibility and the operator's certification. The RAIN layer guarantees the draws; the operator guarantees what the draws pay. This separation is what makes a *shared* RNG certification possible: one layer, many games.

10. Attribution and Governance

10.1 Licence

The reference implementation (`@rain/rng-core`, `@rain/rng-session`, `@rain/rng-node`, the Python mirror and the Solidity anchors) is released under the **MIT licence with one attribution condition**: any product, service, game, bot or AI agent exposed to third parties that uses the RAIN RNG — in whole or part, modified or not — must display the text **"Powered by RAIN RNG"** with a hyperlink to the RAIN fairness page, somewhere its users can see without special effort. Private, internal, research and evaluation use is exempt. The helper `badge()` returns compliant HTML, Markdown and SVG.

10.2 Why attribution is the network effect

A fairness scheme is only as valuable as the number of people who recognise it. The attribution clause is not a marketing device; it is the mechanism by which a player who has learned to check `VERIFIED` on one site knows they can check it on another, and by which a regulator who has evaluated the layer once knows where else it is in use. Every "Powered by RAIN RNG" badge is a public claim that the operator's rounds are recomputable — a claim anyone can test. Attribution turns adoption into verifiability at the ecosystem level, the same way the per-round formula does at the round level.

10.3 Versioning

The ceremony (§3) is version-stable: `r_k`'s formula has not changed since the first on-chain deployment and is what the live verifiers compute. The derivation layer (§4) is versioned by the domain string carried in the DRBG personalisation input (`"RAIN-RNG/2.1"` and successors), so that a verifier always knows which expansion to run, and old rounds remain replayable forever. Wire messages carry an explicit `v` field. Breaking changes to the ceremony itself would be a new major version and a new set of on-chain verifiers; none is planned.

10.4 Reference implementations

Language	Package	Role
TypeScript	`@rain/rng-core`	Pure functions: chains, seeds, `r_k`, `gameSeed`, DRBG, `verifyRound`; zero dependencies; browser / Node / workers
TypeScript	`@rain/rng-session`	The ceremony with persist-before-reveal and decision-before-reveal enforced; transport-agnostic
TypeScript	`@rain/rng-node`	Operator-runnable house node: WAL, HA via shared store, `/verify`, `/healthz`, `/metrics`
Python	`rain-rng`	Byte-identical mirror of core + DRBG for server-side studios and laboratories
Solidity	`RngAnchor.sol`, `ChannelManagerAA.sol`, `EngineV2RulesVerifier.sol`	On-chain commitments and the canonical `seedOf` / `outcomeOf` formulas

Cross-implementation test vectors (Appendix A) are generated from the live front-end engines and re-checked against the deployed verifier on Arbitrum by `npm run test:live`.

11. Live Status — Honest

11.1 What is deployed

Two reference deployments run the ceremony in topology (d) on Arbitrum One (chain id 42161): **playmarkets.bet** and **maycasino.xyz**, on `ChannelManagerAA 0x60743006c2a5Dd5b9907CA37e7381854ff860959` with `EngineV2RulesVerifier 0x5d6048AB261e6151AB44fE6053e7AEb35Cb5C1B1` and `GameMuxVerifier8 0xb72B38123CF3E5F2605f2EED99c53c6D8a331b25`. Both operate in play-money tokens. Contracts are Sourcify-verified.

11.2 Measured vs target

Item	Status	Value
Ceremony `r_k` formula live and verified against on-chain code	**Measured**	Byte-identical: core tests vs `seedOf` on Arbitrum via `eth_call`
Session-seed recompute from a live `SessionSeedCommitReveal`	**Measured**	tx `0x1c62012f…ba9e`, block 504,537,592 — matches (§7.3)
Independent replay of live engine rounds	**Measured**	540 / 540 rounds re-derived OK, 0 bad (8 Sep 2026)
Local wire test of Engine V2	**Measured**	250 spins × 3 games; 2,259 checks each, 0 fails
End-to-end spin latency, browser ↔ house node (topology d)	**Measured**	p50 92–104 ms, p95 99–176 ms across four engine games; max 125 ms in a 491-spin run
House node in-process RTT	**Measured**	p50 11 ms
In-browser round derivation (engine replay)	**Measured**	≤ 14 ms
On-chain transactions per spin	**Measured**	0
Reference test suite	**Measured**	`rng-core`: 14 tests (12 offline pass, 2 live-gated); `rng-session`: 7; vectors: 40 rounds, 24 game seeds, 16 crash ticks, 12 sessions, 5 chains, 6 material derivations
HMAC-DRBG / ChaCha20 derivation layer	**v2.1 — in implementation**	NIST CAVP vectors to be included in the suite
Chain length 65,536 with transparent rotation	**v2.1 — in implementation**	—
Pipelined commit of *k+1* during *k*	**v2.1 — in implementation**	**Target** added p95 < 30 ms vs server RNG; not yet measured
Operator-runnable node (`@rain/rng-node`) with WAL, HA, `/verify`	**v2.1 — in implementation**	Crash test and two-node HA compose in scope
Python mirror	**v2.1 — in implementation**	Byte-identical to TS vectors
External smart-contract audit	**Not done**	Internal review only

Item	Status	Value
Laboratory (GLI-19 class) evaluation	**Not done**	Design mapping in §9.2
SP 800-22 battery on reference output	**Planned with v2.1**	—

11.3 Known limits

1. **Pre-audit.** No external audit of contracts or reference code. Do not run real money at scale before one.
2. **Admin keys.** `setVerifier`, `setOperator`, `setPaused` on the live `ChannelManagerAA` are owner-controlled for pre-audit iteration. They cannot alter an open channel's seed or roots, but must move to a timelock/multisig before real money.
3. **Hybrid engine games are hash-committed, not re-executed on-chain.** Misreports are detectable and punishable via the client's refusal path, not automatically slashed.
4. **Fairness is per round, not statistical.** The system proves no one could bias a round; it does not prove that the RTP a player experienced matches the published figure — that is the payout table and the law of large numbers.
5. **Liveness ≠ safety.** A house that goes offline cannot steal, but in topology (d) exiting requires a `forceClose` and a challenge window (10 min or 24 h by channel size).
6. **The v1 house node is not yet an operator binary.** `@rain/rng-node` (v2.1) is what closes this gap.
7. **Packages not yet on the public npm registry** at the time of writing; the SDK is distributed from the repository.

12. Roadmap

Milestone	Content
v2.1 — Standard candidate	HMAC-DRBG + ChaCha20 layer with CAVP vectors; 65,536-chain rotation; pipelined commit; `@rain/rng-node` with WAL + HA + `/verify`; Python mirror; `INTEGRATION-OPERATOR.md`; measured added-latency figures replacing the target in §5.1; SP 800-22 run on reference output; npm publication.
Lab engagement	Submit the RNG layer (this paper, the spec, vectors, reference code) to a GLI-19-class laboratory for evaluation of the randomness layer independent of any game.
Security audit	External audit of `ChannelManagerAA`, verifiers, `RngAnchor`, and the reference libraries; timelock/multisig for admin keys.
Real-money reference deployment	Topology (d) with a stablecoin settlement token after audit and lab evaluation.
Ecosystem	Third-party operators in topologies (b) and (c); public registry of "Powered by RAIN RNG" deployments; Go/Rust mirrors if demanded.
v3 exploration	Threshold house (multiple operators contribute `hRev`), hardware-backed chain secrets (HSM), and a compact on-chain verifier for DRBG-derived outcomes.

Appendix A — Test Vectors

All vectors are generated from the live front-end engines by `scripts/gen-vectors.mjs` and shipped in `packages/rng-core/test/vectors.json` (generated 2026-09-13). They are re-checked against the deployed verifier on Arbitrum by `npm run test:live`. A selection:

A.1 Keccak256 sanity

```
keccak256("")        = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
keccak256("hello")   = 0x1c8aff950685c2ed4bc3174f3472287b56d9517b9c948127319a09a7a36deac8
```

A.2 Hash chain (length 2)

```
secret = 0xe8c070c07a698b7b44390a779a6bd9807af36ceabe55c2c173df8a1276661189
c[2]   = keccak256(secret)          = 0x718d81c3baa37c03103346aaf26332730589ebac93ac35889528d4279c1f89b6
c[1]   = keccak256(abi.encode(c[2])) = 0x360d7049f585d3e6b6632096f0574664817be4e3e38b687176364fb73c26a6b
b
c[0]   = keccak256(abi.encode(c[1])) = 0xc7c00c62e44af0940af4fa895d5098e9341995caeb062e1882c73b18ce23919
7  (root)
```

A.3 Session seed (live open, Arbitrum One)

```
channelId   = 0xc6fcfe25563701b9e9ec1167f6ac54ad826e9c07aa0d461b39bb358be4854f0d
houseSeed   = 0xa52aa98608487152568524baf8212b65a852159cc1de7ede77de43ac3b6daa13
playerSeed  = 0x8c3d4d13e80e659be1e9617c93ec46d49b992540088c0d6f0da1b12288301c30
sessionSeed = 0x0ae271400c5eb64c6e8226f6cddbb0736b483120e4e85db1c7fc5b10588bffbd
houseSeedCommit = 0x5e935d70097cca9bed5d4a890846a31b8fdd5697d249fbfb92db2084326d62de
```

A.4 Round outcome (canonical `r`)

```
pRev        = 0x668b0ede100e058f38d2bdd057db678fed3a59e525480478db4256f95c353b00
hRev        = 0xd0beea2fada231a45413059e9049e53c90eb9a538afc1b488ecf03a8666f89f7
sessionSeed = 0xafb0441e3ee6ad8edc05d58831ab4df6682a7f63747eb79c8daa1c1017832bdf
channelId   = 0x98dad8ac029c01d821045a7d512b197685cc56d07a52bb6eaf245213194d9a33
k           = 1
r           = 0x72386aa3f653273aa07626135bb112a7d3a87fbdbc4ded7f286f26c8f7489a8d
r % 100 + 1 = 26   (dice)        r % 37 = 23   (roulette)
```

```
pRev        = 0x9e8ae75b6a4611988360a59e623bb8703368637341e14f3abcfaca3e99cb0f25
hRev        = 0x1c2fefa599b548ef793e3a73f6732fc48d8fa6690b64c3016573b676c45873cc
sessionSeed = 0x58f0c9489159443fe4b01b8b9e01d20333da828b09f4a34ff50fc4caeeaa9fb7
channelId   = 0x7c4b9de850eae66a3cbe3a89fa42c8f09dfd37a7a8b1d2a998bf59c9bd2f8c29
k           = 2
r           = 0x198b2201546035462c0a1e6f9fc1f087c84a6469dac5848481c845965b1b095a
r % 100 + 1 = 83   (dice)        r % 37 = 33   (roulette)
```

A.5 Engine V2 game seed (domain-separated)

```
pRev        = 0xf1c2fbe645543fcdec88b890bba0b0b59391dc287e4fa1396e85cf0e09a50033
hRev        = 0xa5a64e3eca9beafc3ec32842f3be63350fc67d64ab7605172bd311293014b3f6
sessionSeed = 0x162cbbbb4586b29e2b91267f1e9f9e92cb9cf5b7f96e1c325fcdb44c1d9e1ee2
channelId   = 0xf22aa1fc519a4d1de879fd583318033d707c770d5c41a5bfe7cf08894784e1ef
k           = 3170
gameId      = "foundry"   keccak256("foundry") = 0x4eb2f10301a3ed7f2c31091074ca429f73cb8c51539e1a0005e13
2f70b8bb74a
gameSeed    = 0x28421c6bd9a45e5a84b9356069e53649c69c4a47ba4cc8f54e635f3714a55133
```

A.6 Player material from one wallet signature

```
message         = "RAIN session key v1 for 0x60743006c2a5Dd5b9907CA37e7381854ff860959 chainId 42161"
sig             = 0x7d3827165c3916cf2b70269e27af386d39ea5bc61d2729ca0543e04020821798d268…4bed3e
base            = keccak256(sig)               = 0x875a10a61e1f13fb65dfca35c4b560661d666acc070e83b18517c
a3094a0d41a
sessionWalletPk = keccak256(base ‖ "session")  = 0x05c055bd9033f342eba27f5bf22ec1adaed89b9e6b9a08f05a76f
32983158633
openNonce       = 4560430705875
chainSecret     = keccak256(base ‖ "chain:4560430705875") = 0x49afd08131355b891f8a4de4349beba6f70bee20e1
3644e5f8a0cc16d5d814ec
playerSeed      = keccak256(base ‖ "seed:4560430705875")  = 0xb10f6d3d7c6db68aa723b9f608b0f5fda14bfb69ab
9f9da0602d291ceac0bd23
```

A.7 DRBG vectors (v2.1) — HMAC-DRBG SHA-256 CAVP known-answer tests and RAIN `gameSeed → draws[0..15]` vectors ship with `@rain/rng-core` 2.1 and the Python mirror; they are omitted here because the derivation layer is in implementation at the time of writing (§11.2).

Appendix B — API Surface

`@rain/rng-core` (pure functions, zero dependencies)

```
keccak256(bytes|hex) · keccakWord(x) · keccakWords(...w) · gameIdHash(gameId)
houseSeedCommit(houseSeed) · sessionSeed(houseSeed, playerSeed, channelId) · channelId(manager, player,
operator, openNonce, chainId)
verifySessionSeed({channelId, houseSeed, playerSeed, sessionSeed, houseSeedCommit?})
hashChain(secret, len) · chainRoot(secret, len) · chainOk(anchor, reveal) · verifyChainElement(root, k,
elem)
outcome(pRev, hRev, sessionSeed, channelId, k) · outcomeUint(...) · outcomeMod(..., N) · modN(r, N)
gameSeed(pRev, hRev, sessionSeed, channelId, k, gameId) · outcomeWith(pRev, hRev, sessionSeed, channelI
d, ...extra)
verifyRound({pRev, hRev, sessionSeed, channelId, k, expected?, N?, gameId?, pAnchor?, hAnchor?}) → {ok,
r, value, matches, pChainOk, hChainOk}
deriveMsg(manager, chainId) · materialFromSig(sig) · chainSecretFor(base, openNonce) · playerSeedFor(bas
e, openNonce)
randomBytes32() · badge({url?, dark?}) → {text, html, markdown, svg, attribution, url, version}
v2.1: drbg(gameSeed, {backend: "hmac-sha256" | "chacha20"}) → {nextU32, nextU64, nextFloat53, nextInt
(N), bytes(n), index}
```

`@rain/rng-session` (the ceremony, transport-agnostic)

```
new RainRngHouse({persist, chainLen?, houseSeed?, chainSecret?, meta?})
  .getTerms() → Terms · .open(Open) → Opened · .reveal(Commit, compute?) → Reveal · .transcript()
new RainRngClient({chainLen?, playerSeed?, chainSecret?})
  .open(Terms) → Open · .opened(Opened) → sessionSeed · .commit(decision) → Commit · .settle(Reveal, N?)
→ RoundResult · .abort() · .fairness(result)
localOpen(house, client) · localRound(house, client, decision, N?)
RngAnchorAdapter(address, signer): commit · bindPlayer · revealSeed · submitReveals · session(sessionId)
Wire messages: Terms{v:2, houseSeedCommit, houseChainRoot, chainLen, meta?} · Open{v:2, playerSeed, play
erChainRoot, chainLen, termsHash}
               Opened{v:2, sessionId, houseSeed, sessionSeed} · Commit{v:2, sessionId, k, decision, pRe
v} · Reveal{v:2, sessionId, k, hRev, claimed?}
```

`@rain/rng-node` (v2.1, operator-runnable)

```
POST /session/open        → Terms          POST /session/{id}/open  (Open) → Opened
POST /session/{id}/round  (Commit) → Reveal     — WAL append + fsync before hRev is returned
GET  /verify?session=&k=  → public fields, roots, recomputed r / gameSeed; secrets after session close
GET  /healthz · GET /metrics (Prometheus)   HA: two nodes behind a load balancer sharing one durable sto
re
```

On-chain (Arbitrum One)

```
ChannelManagerAA 0x60743006c2a5Dd5b9907CA37e7381854ff860959 — openChannel, submitReveals, demandReveal
(60 s), forceClose, challenge
EngineV2RulesVerifier 0x5d6048AB261e6151AB44fE6053e7AEb35Cb5C1B1 — seedOf(...) pure, engineHash(gameIdHa
sh) view
RngAnchor.sol (deploy yourself, ~0.9 M gas) — commit, bindPlayer, revealSeed, submitReveals, seedOf, out
comeOf, gameSeedOf, verifyChainElement
```

Appendix C — Glossary

Term	Definition
Anchor	The previous reveal (or the chain root at $k = 1$) against which the next reveal is checked: `keccak(rev_k) == anchor`. Also: an on-chain record of a commitment.
Ceremony	The RAIN RNG protocol: session open (commit both seed halves and both chain roots) followed by per-round commit/reveal.
Chain (hash chain)	A sequence `c[0..n]` with `c[i-1] = keccak(c[i])`; `c[0]` is committed, `c[k]` is the *k*-th reveal.
Chain rotation	Exchanging fresh chain roots inside a normal round before the current chains are exhausted.
Commit / Reveal	The two messages of a round: the player's decision + `pRev_k`; the house's `hRev_k`.
Decision-before-reveal	Client rule: the bet and `pRev_k` are sent before `hRev_k` is seen.
DRBG	Deterministic random bit generator (NIST SP 800-90A); here HMAC-DRBG (SHA-256) or a ChaCha20 stream, seeded per round.
Domain separation	Including a context word (game id, version, round) in a hash so that different uses never share randomness.
`gameSeed_k`	`r_k` extended with `keccak256(bytes(gameId))`; the DRBG seed for a round of a specific game.
GLI-19	Gaming Laboratories International standard for interactive gaming systems, including RNG requirements.
HouseFault / PlayerForfeit	On-chain outcomes of a dispute in topology (d): the house refused or misreported (compensation from bond); the player refused a loss (stake forfeited).
Persist-before-reveal	House rule: the player's commitment is durably stored before `hRev_k` is released.
`r_k`	`keccak256(abi.encode(pRev_k, hRev_k, sessionSeed, channelId, k))`; the round's 256-bit randomness.
`sessionSeed`	`keccak256(abi.encode(houseSeed, playerSeed, channelId))`; fixed at open from both parties' halves.
SP 800-22	NIST statistical test suite for random and pseudorandom number generators.
SP 800-90A	NIST recommendation for DRBG mechanisms (Hash_DRBG, HMAC_DRBG, CTR_DRBG).
State channel	An off-chain, dual-signed sequence of states with on-chain escrow and dispute resolution; topology (d).
Topology	One of the four deployment shapes in §6; the ceremony is identical in all of them.
VRF	Verifiable random function: a keyed function whose output comes with a proof of correct evaluation; the basis of oracle randomness services.

Term	Definition
`/verify`	The public endpoint of a RAIN node returning the public inputs of any round for independent recomputation.

RAINRISKMARKETS.COM

PLAYMARKETS.BET · MAYCASINO.XYZ

Powered by RAIN RNG