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 1.1 · 14 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 keys a standards-based draw layer — **HMAC_DRBG (SHA-256) per NIST SP 800-90A, instantiated from a protocol-derived 256-bit seed**, or a ChaCha20 block function (RFC 8439) — from which every draw of the round is read by index. 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 un-pipelined full spin round trips through the money protocol. Version 2.1 — implemented and specified in this paper — adds the DRBG draw layer, 65,536-element chains with automatic rotation, and pipelined commitment of round *k+1* during round *k*'s animation. **Pipelining has now been measured**: in a 1,000-spin load test on loopback with a simulated 40 ms round trip to the RAIN node, the RNG work on the operator's `/spin` critical path fell from **p95 41.1 ms (pipelining off) to p95 0.50 ms (pipelining on)**; with an in-process node, from p95 0.72 ms to 0.55 ms. A measurement over a real network is pending. The v2.1 draw layer has passed our own runs of Dieharder, NIST SP 800-22 and TestU01 SmallCrush and a 10^8-outcome scaled-outcome programme (§11); it has **not** yet been evaluated by a laboratory or audited by a third party. 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.

Source: latency — `sdk-v2/examples/slot-spin-endpoint/README.md` (Measured table), `sdk-v2/docs/INTEGRATION-OPERATOR.md` §3; live p50 92–104 ms — `playmarket/fe-ux/docs/engine-v2/REAL-PLAY.md`; 540/540 — `sdk-v2/CHANGELOG.md` (2.0.0, Fairness UX row).

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 still a target.

#	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, live v2.0 money protocol (un-pipelined): full spin round trip p50 92–104 ms, p95 99–176 ms. Measured, v2.1 pipelining (loopback, simulated 40 ms RTT, 1,000 spins): RNG work on the `/spin` critical path p95 0.50 ms, versus 41.1 ms with pipelining off.** Real-network measurement pending.	§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

Source: G4 — `playmarket/fe-ux/docs/engine-v2/REAL-PLAY.md` (91–104 / 99–176 ms across engine games); `sdk-v2/examples/slot-spin-endpoint/README.md` (Measured table).

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)`; off-chain: a hash of the Terms and Open messages)
`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 on-chain money deployments use *n* = 4,096 (4,095 usable rounds per channel; the bankroll layer re-opens near exhaustion). `@rain/rng-session` 2.1 defaults to **n = 65,536** (`DEFAULT_SESSION_CHAIN_LEN`) and rotates chains automatically at 90 % consumption (`DEFAULT_ROTATE_AT = 0.9`; §5.4). Materialising a 65,536-element keccak chain costs about 1.3 s once per session and is done ahead of time.

Source: `sdk-v2/packages/rng-session/src/index.ts` (constants); `sdk-v2/CHANGELOG.md` 2.1.0 (≈1.3 s); `sdk-v2/docs/RNG.md` §2.3 (4,096 on-chain).

3.3 The ceremony

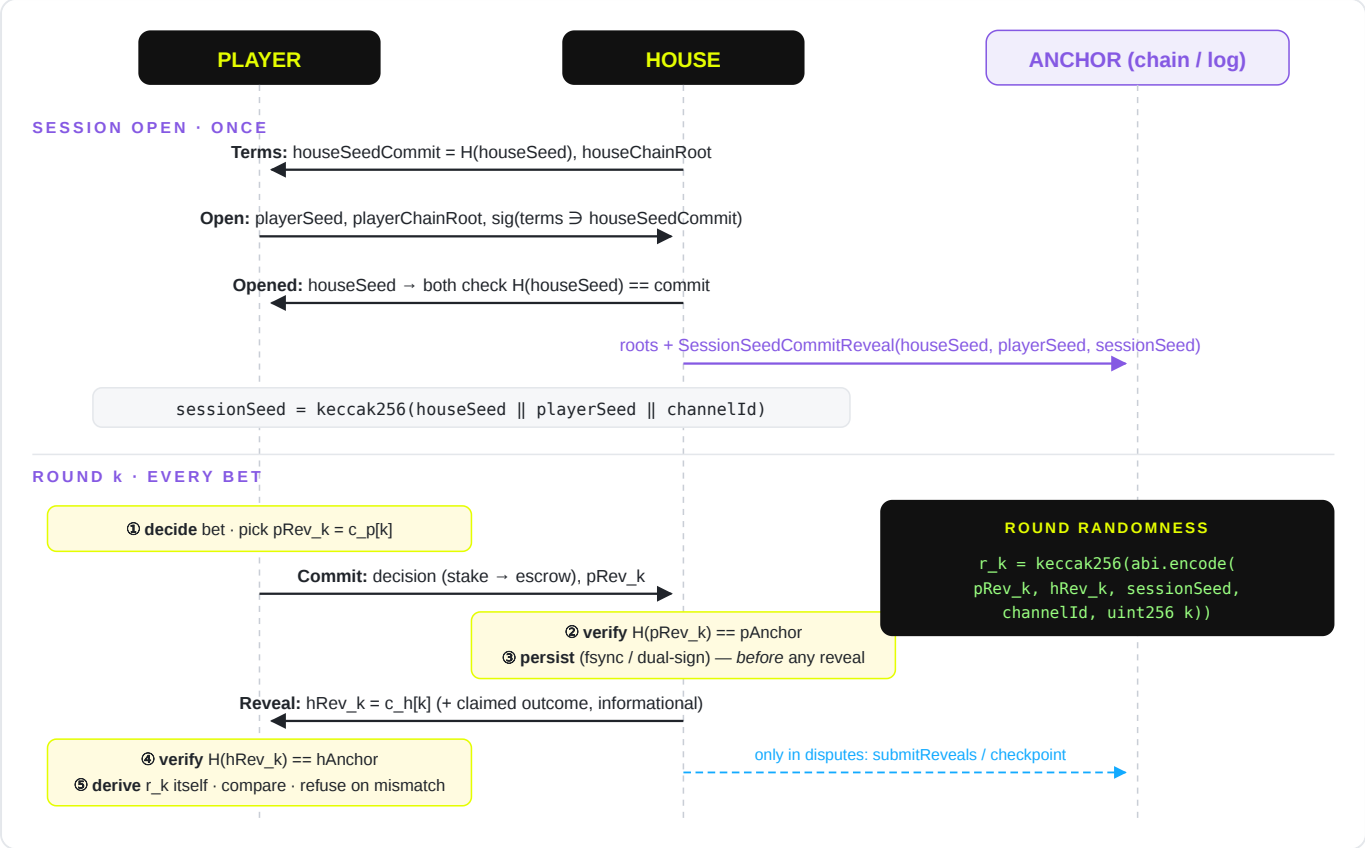

Figure 1 — The RAIN RNG ceremony: one open, then a two-message round.

Session open (once).

1. The house picks `houseSeed` and a chain secret; publishes `houseSeedCommit`, `houseChainRoot` and the session terms (`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 binding over the terms *including* `houseSeedCommit` (`Open.termsHash`; on-chain, the EIP-712 signature). The house therefore cannot change its commitment after seeing the player's seed.

3. The house reveals `houseSeed` (`Opened`). 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` (`Commit{sessionId, k, decision, pRev}`). 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` (`Reveal{sessionId, k, hRev, claimed?}`). 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 (`RainRngHouse` awaits the `persist` hook before any reveal; `@rain/rng-node` appends to a write-ahead log with fsync, or commits to Postgres), 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. A duplicate commit for the same *k* with the same `(decision, pRev)` is answered with the same `hRev`; a different one is refused (HTTP 409). A round is revealed once.

Source: `sdk-v2/packages/rng-session/src/index.ts` (wire types, `HousePersist`); `sdk-v2/packages/rng-node/src/house.ts` (idempotent replay / 409); `sdk-v2/lab/RNG-DESCRIPTION.md` §8.

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. The argument is written out as four claims with stated assumptions in the laboratory package (`lab/RNG-DESCRIPTION.md`, Appendix A), and its empirical counterpart — adversarial player inputs leaving the outcome distribution unchanged — is in §11.2.

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 `r_k` with a game-domain string (§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 — and operator servers in the RNG-as-a-service topology (§6b) — draw 32 bytes from the platform CSPRNG (`randomBytes32()` → `crypto.getRandomValues`, which throws if no CSPRNG is available; there is no fallback to time or `Math.random`).

Source: `sdk-v2/packages/rng-core/src/rng.ts` (`deriveMsg`, `randomBytes32`); `sdk-v2/lab/RNG-DESCRIPTION.md` §2.1.

4. From *r* to a Game: One Seed, a DRBG, Index-Addressed 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 per-(round, game) key → every draw of the round is a pure function of `(r_k, gameId, i)`.

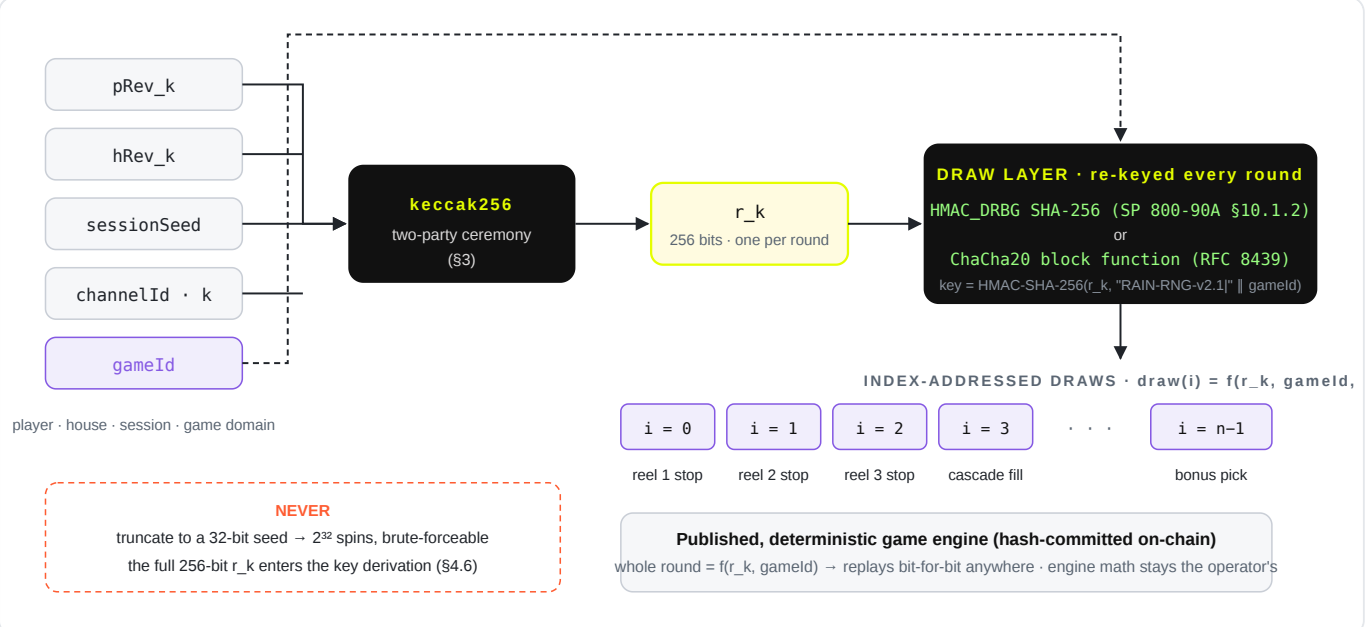

Figure 2 — From the two reveals to every draw of a round.

The reference implementation, `drbg(r, gameId, { mechanism })` in `@rain/rng-core` (mirrored by `rain_rng.drbg` in Python), offers two interchangeable mechanisms, both keyed identically:

- `"hmac-drbg"` **(default) — HMAC_DRBG with SHA-256 as specified in NIST SP 800-90A Rev. 1 §10.1.2, instantiated from a protocol-derived 256-bit seed.** It is the mechanism laboratories already know how to certify, has published CAVP test vectors, and is implementable in a few dozen lines in any language.
- `"chacha20"` — the ChaCha20 block function (RFC 8439 §2.3), keyed identically, block counter = draw index. It is roughly an order of magnitude cheaper per draw and gives O(1) random access to the *i*-th draw, which matters for parallel verification and for bulk statistical collection.

Both are deterministic functions of `(r_k, gameId, i, mechanism)`. The exact byte layout is normative in §4.2 and fixed by test vectors shared between the TypeScript and Python references (Appendix A.7); the same inputs must produce the same bytes in every implementation. The two mechanisms are treated as two implementations of the draw layer and evaluated separately (GLI-19 §4.5.1).

What is and is not claimed. The draw layer is a deterministic pseudo-random function built from the SP 800-90A HMAC_DRBG mechanism. **It is not claimed to be an SP 800-90C random-bit generator**: the seed `r_k` is not an SP 800-90B entropy source but a cryptographic combination of two parties' CSPRNG outputs, fixed by commitments before either party can act on it. This positioning follows the analysis in `sdk-v2/docs/LAB-SUBMISSION-REQUIREMENTS.md` §5 and is the wording used in the laboratory package.

4.2 Domain separation per game — the normative derivation

```
key        = HMAC-SHA-256(key = r_k, msg = utf8("RAIN-RNG-v2.1|" + gameId))            // per-(round, game) key
hmac-drbg: draw(i) = HMAC_DRBG_SHA-256.Instantiate(entropy_input = key, nonce = "", personalization = "")
                        .Generate(32 bytes, additional_input = uint64be(i))           // SP 800-90A §10.1.2; f
resh instance per draw
chacha20:  draw(i) = ChaCha20_block(key, counter = i, nonce = 0^96)[0..32)            // RFC 8439 §2.3
```

HMAC_DRBG parameters: security strength 256; entropy input 256 bits (the derived `key`); nonce empty — permitted when the entropy input already carries ≥ security_strength bits (SP 800-90A §8.6.7 alternative), and stated here as an explicit choice; personalization empty; `additional_input` = the 8-byte big-endian draw index; `max_number_of_bits_per_request` enforced at 2^{19}; reseed never invoked, because each round is a new instantiation. Within a round the index space is 2^{53} (`hmac-drbg`) or 2^{32} (`chacha20`) independent outputs.

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. A draw is identified by `(r_k, gameId, i, mechanism)` and nothing else; `mechanism` is chosen by the integrator at build time and recorded in the fairness panel (`stream.mechanism`). There are no other switches on the certified path.

The legacy Engine V2 formula. The pre-v2.1 slot engines live on Arbitrum use a keccak game seed, `gameSeed_k = keccak256(abi.encode(pRev_k, hRev_k, sessionSeed, channelId, uint256 k, keccak256(bytes(gameId))))`, exposed as `EngineV2RulesVerifier.seedOf` and available as `gameSeed(...)` in `@rain/rng-core`. It remains byte-identical to the deployed verifiers and is what the 540/540 audit re-derived (§4.4). New games use the HMAC derivation above.

Source: `sdk-v2/packages/rng-core/src/drbg.ts` (`RNG_DOMAIN_PREFIX`, `deriveDrawKey`, `drbg`); `sdk-v2/lab/RNG-DESCRIPTION.md` §4–5; `sdk-v2/docs/RNG.md` §2.2, §2.5.

4.3 Uniform integers, floats and reductions

From `draw(i)` the reference defines:

Function	Definition	Bias
`uint32(i)`	first 4 bytes of `draw(i)`, big-endian	none
`uint53(i)` / `float(i)`	first 53 bits; `float = uint53 / 2^53` ∈ [0, 1)	none on the 2^{53}-point grid

Function	Definition	Bias
`intBelow(i, n)`, 1 ≤ n ≤ 2^{32}	rejection sampling: take `uint32(i)`, `uint32(i+1)`, ... until `v < 2^32 − (2^32 mod n)`; return `v mod n`; `intBelowEx` also returns the number of words consumed	**zero** — exactly uniform ("the discard of RNG values is permissible", GLI-19 §3.2.3(a))
`shuffle(i, arr)`	Fisher–Yates/Durstenfeld using `intBelow`	**zero** — each permutation has probability 1/len!
`cursor(start?)`	sequential wrapper (`next`, `uint32`, `float`, `intBelow`, `shuffle`) that advances by the words actually consumed, rejections included, so an engine's replay is deterministic	—

Expected rejection rate of `intBelow` is `(2^32 mod n)/2^32` — $2.3 \cdot 10^{-8}$ for n = 100, ≈ 0.5 in the worst case n = $2^{31}+1$. A 52-card `shuffle` consumes 51 words in expectation.

The certified path is `drbg` + `intBelow` / `shuffle` / `float`. The **legacy** reduction `outcomeMod` = `uint256(r) % N` — dice N = 100, roulette N = 37 — is kept because it must stay a modulo to remain byte-identical to the Solidity verifiers; its relative bias is `< N / 2^256` (< $7.9 \cdot 10^{-77}$ for N = 100), which would take ≈ 10^{155} draws to detect. It is documented with its bound in the scaling inventory and is **outside the v2.1 certification scope**. Every other reduction in the code base — including the vendored game engines' own `%` — is inventoried and classified in `sdk-v2/lab/SCALING-INVENTORY.md`.

Source: `sdk-v2/packages/rng-core/src/drbg.ts`; `sdk-v2/lab/RNG-DESCRIPTION.md` §6 and App. B; `sdk-v2/lab/SCALING-INVENTORY.md`.

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 on 8 September 2026, checking seed, chain anchors, win amount, result-tree hash and payout on each, with zero mismatches.

Source: `sdk-v2/CHANGELOG.md` (2.0.0); `playmarket/fe-admin-login/docs/whitelabel/PROVABLE-FAIRNESS.md`.

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 moves new games to the DRBG layer for three practical reasons, none of them a weakness in keccak:

1. **Certifiability.** A laboratory can evaluate "HMAC_DRBG per SP 800-90A instantiated from a protocol-derived 256-bit seed" 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-SHA-256 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. Measured on the reference machine: ≈ 55 µs per HMAC-DRBG draw, ≈ 4.5 µs per ChaCha20 draw, a three-reel spin ≈ 0.2 ms — small against any animation, and the same replay runs in the player's browser, in the house node and in the verifier.

Source: `sdk-v2/docs/RNG.md` §2.5 (µs figures).

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 in production, 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 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 §3.3.2(b) requires that the RNG's state after seeding cannot be determined or reasonably estimated. A 32-bit seed fails that clause on its face.

The rule is therefore: **the full 256-bit `r_k` enters the key derivation; nothing narrower ever seeds a round.** A migration helper `seed32Compat(r)` exists for running legacy 32-bit-seeded engines against RAIN rounds *during development*; it is marked deprecated, prints a warning once per process, is not imported by any production engine, and is listed for the laboratory under "undisclosed switches" so that its absence from production builds can be confirmed.

Source: `sdk-v2/packages/rng-core/src/drbg.ts` (`seed32Compat`); `sdk-v2/lab/RNG-DESCRIPTION.md` §9.

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 keyed 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. This is also how the construction answers GLI-19 §3.3.2(c) ("periodically modify its state through the use of external entropy"): from the house's point of view `pRev_k` is fresh external entropy on every round, not periodically; and each party's secret material is replaced from the CSPRNG at rotation, at least every 65,536 rounds.

5. Web2 UX: Latency — Measured

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 in the money protocol's house node. What remains is `T_network`, 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 EIP-712 signing and state handling) 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. What can still cost a full round trip is the *network hop to a separate RAIN node* in the RNG-as-a-service topology. Pipelining removes it from the critical path.

Source: `playmarket/fe-ux/docs/engine-v2/REAL-PLAY.md` (≤ 14 ms engine, p50 11 ms in-process, 92–104 / 99–176 ms).

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

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. In `@rain/rng-session` 2.1, `RainSpinClient.prefetch()` commits round *k+1* (decision string plus `pRev_{k+1}`) and puts it on the wire the moment round *k* settles, while round *k*'s reels are still spinning; the house persists it and answers with `hRev_{k+1}`, which the client parks. When the game calls `spinSeed(k+1)`, the reveal is already there and the round settles locally with no network wait. Up to `maxInFlight` rounds (default 2, on both house and client) may be in flight; each `pRev` must still chain to the previous accepted one, and rounds settle strictly in order. For auto-spin and turbo modes — where the *decision* is fixed in advance — the network cost of the ceremony disappears into the animation entirely. Where the decision genuinely changes at the last moment (a different stake), the client commits at spin time and pays one round trip for that round — the same as today.

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.

Measured. `examples/slot-spin-endpoint/loadtest.mjs` runs 1,000 sequential spins against an operator `/spin` endpoint backed by RAIN (`spinSeed` → `drbg(r, "example-slot-v1")` → three `intBelow` reel stops), on Node 24, loopback, JSONL write-ahead-log persistence, with the RAIN node either in-process or behind a **simulated 40 ms round trip** and a deliberately harsh 60 ms "animation" gap between spins:

RAIN node	Pipelining	RNG work on the `/spin` handler, p50 / p95	HTTP end-to-end, p50 / p95
in-process	off	0.50 / 0.72 ms	1.22 / 2.19 ms
in-process	**on**	**0.38 / 0.55 ms**	1.05 / 1.52 ms
remote, 40 ms RTT, 60 ms animation	off	40.9 / **41.1 ms**	42.1 / 42.6 ms
remote, 40 ms RTT, 60 ms animation	**on**	**0.35 / 0.50 ms**	1.50 / 1.91 ms

With pipelining the round trip to the RAIN node is off the critical path entirely as long as the animation is longer than the RTT. The v1.0 target of "added RNG latency p95 < 30 ms with pipelining" is met with a wide margin under these conditions. **Conditions matter and are stated**: loopback, simulated RTT, one sequential player, a mock paytable. A measurement over a real network between an operator server and a RAIN node in a different data centre is on the roadmap (§12) and until then the 0.50 ms figure should be read as "the ceremony's own cost", not as a field result.

Source: `sdk-v2/examples/slot-spin-endpoint/README.md` (Measured table); `sdk-v2/examples/slot-spin-endpoint/loadtest.mjs` (parameters); `sdk-v2/packages/rng-session/src/spin.ts` (`prefetch`, `spinSeed`, `maxInFlight`).

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**; the chain is touched at login and at cash-out.

5.4 Chain rotation at 65,536

A session's chain is finite. With 4,095 usable rounds, turbo auto-spin could exhaust a chain in under two hours, forcing a mid-session reopen. `@rain/rng-session` 2.1 sets the default chain length to **65,536** (≈ 18 hours at one spin per second) and **rotates automatically**: when either side's cursor passes the 90 % high-water mark, `needsRotation` flips; the next `spinSeed()` call, once nothing is in flight, runs a four-step rotation ceremony over the same transport — the house issues fresh `Terms` (`rotateTerms`: new `houseSeedCommit`, new chain root), the client answers with a fresh `Open` (`rotate`: new `playerSeed`, new chain root), the house reveals (`rotated`), and both derive a new `sessionSeed`. The cursor restarts at $k = 1$ under a new `sessionId`. No chain transaction is required — unless an `RngAnchorAdapter` is wired into `onRotate`. Every rotation is recorded (`rotations[]`, `history[]`) with a link to the previous session, and old proofs keep verifying against the old session's public data. The chain is fully materialised in memory (32 bytes × 65,536 = 2 MiB per party) and built ahead of time; `rotateTerms()` may be called early so the ≈ 1.3 s of keccak never lands on a spin.

Source: `sdk-v2/packages/rng-session/src/index.ts` (`DEFAULT_SESSION_CHAIN_LEN`, `DEFAULT_ROTATE_AT`, `rotateTerms`/`rotate`/`rotated`, `RotationRecord`); `sdk-v2/docs/INTEGRATION-OPERATOR.md` §7.

6. Deployment Topologies

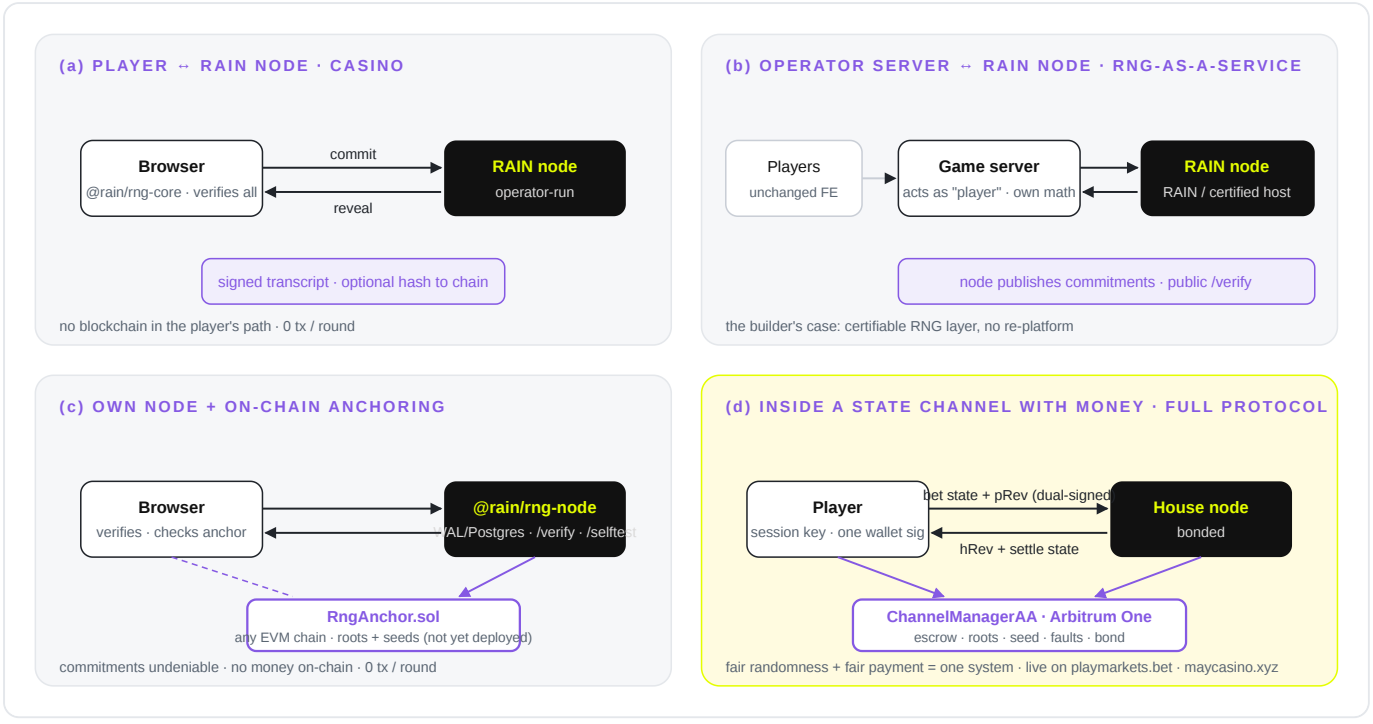

Figure 4 — Four ways to deploy the same ceremony.

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" (party A, `RainSpinClient`): it commits a decision and `pRev` per round to a RAIN node (party B, `RainRngHouse` — run by RAIN, a certified third party, or the operator itself), 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 — five calls (`open`, `spinSeed`, `prefetch`, `drbg`, `verifyRound`) documented in `INTEGRATION-OPERATOR.md`, with in-process, WebSocket and HTTP long-poll transports and a Python mirror for Python servers.

(c) Operator-run node with on-chain anchoring. The operator runs the node itself (`@rain/rng-node`: WAL or Postgres persist-before-reveal, HTTP + WebSocket, `/verify/:sessionId/:k`, `/healthz`, `/selftest`, `/metrics`, a two-worker HA compose file) and publishes its session commitments to an `RngAnchor` contract on any EVM chain. 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. `RngAnchor.sol` is exercised in a local EVM by the test suite and is **not yet deployed** on a public chain. If one legal entity runs both the game and the node, the "two independent parties" argument depends on these published commitments — the laboratory package says so explicitly.

(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	Operator
Anchoring	Signed transcript	Node publishes commitments; `/verify`	`RngAnchor` on any EVM chain (not yet deployed)	`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

Source: `sdk-v2/packages/rng-node/README.md`; `sdk-v2/packages/rng-session/README.md` ("Not deployed"); `sdk-v2/docs/INTEGRATION-OPERATOR.md` §1–2, §8.

7. Verification and Audit

7.1 Player post-hoc verification

Any settled round exposes `channelId` / `sessionId`, `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 (legacy): `roll = r % 100 + 1`. Roulette (legacy): `n = r % 37`. Engine V2 (legacy): `gameSeed` with the sixth word `keccak256(bytes(gameId))`, then run the published engine. v2.1 games: `key = HMAC-SHA-256(r, "RAIN-RNG-v2.1|" + gameId)`, then `draw(i)` per §4.2 and the operator's published math over the cursor.
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?, gameId?, pAnchor?, hAnchor? }).ok`, then `drbg(r, gameId)` to replay the draws. `RainSpinClient.spinSeed()` returns the same fields as a `proof` object with the formula spelled out.

7.2 Public `/verify`

Every RAIN node in topologies (b) and (c) exposes an unauthenticated, CORS-open `GET /verify/:sessionId/:k` that returns the session's public commitments (`houseSeedCommit`, both chain roots, `playerSeed`, `sessionSeed`, `houseSeed` — the latter already public from the `Opened` message), the persisted commit (`decision`, `pRev`, time), the reveal, the recomputed `r` with both chain checks, and the status `revealed` or `committed_not_revealed` — so a persisted-but-unanswered commit is visible to anyone. 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.

Source: `sdk-v2/packages/rng-node/src/house.ts` (`verify`), `src/server.ts` (routes).

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
`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 signed at o
pen
```

Source: `whitepapers/RAIN-RNG-Whitepaper-v1.0.md` §7.3 / App. A.3 (unchanged; re-checked by `npm run test:live` against `SessionSeedCommitReveal` events).

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 index-addressed, 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. The laboratory collection tools (§9.4) do exactly this on synthetic sessions; the same tools run unchanged on a production transcript.

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 binding 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 shows as `committed_not_revealed` on `/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.
House	Answer the same *k* twice with different reveals	Impossible by construction (one chain element per *k*); a second commit for the same *k* with different inputs is refused (409).
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: `RainRngClient.abort()` abandons in-flight rounds whose `pRev` are already public; the next commit must use `k+1`; the house's persisted commit for the aborted *k* remains servable. A chain element is never consumed twice.
Player	Use low-entropy or adversarially chosen secrets to steer outcomes	For any fixed `pRev_k`, `r_k` is uniform because `hRev_k` is unknown to the player and fixed. Demonstrated empirically: 10^8-outcome runs with all-zero, fixed and "grinding" player inputs are statistically indistinguishable from the honest run (§11.2).
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 keyed from `r_{k+1}`, which does not exist until both reveals do. A leaked chain of one party reveals nothing about outcomes without the other party's reveals.
Chain exhaustion	Force a mid-session break or reuse	Rotation at the 90 % 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.

Adversary	Attack	Why it fails
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, or ≈ 0.5 ms on the critical path with pipelining (measured, loopback + simulated 40 ms RTT)**
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	Certified once (BMM, GLI-19, 2022 — §8.5)	Not a gaming-lab construct	Not a gaming-lab construct	**Maps to SP 800-90A HMAC_DRBG + documented seed provenance; §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; HMAC-SHA-256 and ChaCha20 are pseudo-random functions. 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.
- The reference node is **not hardened**: no HSM, secrets in process memory and in the WAL/Postgres in plaintext, no rate limiting beyond a reverse proxy. Key storage and access control are deployment controls (`lab/process/KEY-MANAGEMENT.md`).

- When one operator runs both parties (topology b or c without a third-party node), the two-independent-parties argument rests on *published* commitments (`RngAnchor` or a transparency log). Without them, a selective abort is only recorded, not punished.
- No external security audit of the contracts or the reference implementation, and no penetration test, has been completed at the time of writing (§11.3).

8.4 Liveness versus safety

A house that goes offline cannot steal — the outcome for any committed *k* is fixed by the chain and the persisted commit — but it can stall. `@rain/rng-session` records the stall; the escrow protocol of topology (d) turns it into a compensated fault; topologies (a)–(c) rely on the operator's SLA and on the public visibility of `committed_not_revealed` rounds. This is the same liveness/safety split every off-chain protocol has, and it is stated so that nobody reads "provably fair" as "always available".

8.5 Precedents: how commit-reveal and blockchain RNGs have been certified so far

No laboratory has certified a "provably fair" *ceremony* as such; laboratories certify *an RNG* — the generator plus every scaling step to the final outcome — against GLI-19 §3, GLI-11 §3, UKGC RTS 7 and their national equivalents. Three public precedents frame what RAIN can expect:

- **Chainlink VRF v2 — BMM Testlabs, GLI-19, announced 2 September 2022.** The only public certification of an on-chain randomness primitive. What was certified was the VRF *as an RNG* (elliptic-curve VRF proof with on-chain verification, 256-bit output) under the ordinary GLI-19 RNG requirements — source-code review, statistical output, unpredictability. The release does not claim that the request/fulfil economics, oracle liveness or consuming games' scaling were in scope; no certificate PDF or scope statement is public.
- **BC.Game — iTech Labs, UK RTS, certificates 16 August 2019 and 12 October 2020.** A server-seed/client-seed/nonce "provably fair" casino. iTech certified the **server-side generator and each game's scaling and shuffle code** (Crash, HashDice, Roulette, Plinko, Dice, HiLo, slots, Keno, single- and twelve-deck shuffles), against Marsaglia's Diehard tests, and fingerprinted the certified code. The certificate does not mention the client seed, the commitment hash or player-side verification.
- **Hash Games CW B.V. (Curaçao) — iTech Labs, UK RTS (February 2021 edition), certificates 20 February and 8 April 2024.** Same template as BC.Game: "well-known algorithm… Marsaglia's diehard… certified code fingerprinted."

What this means for RAIN. Nobody has publicly certified a **two-party** commit-reveal in which the *player* is an entropy source and a contract or node enforces the commitment. The closest analogue is the VRF (unpredictable, verifiable, third-party); the closest *process* analogue is iTech's treatment of the crypto casinos (certify generator + scaling, fingerprint the code, ignore the client-side story). RAIN should therefore expect an engineering-level review of a novel construction rather than a template job, and has organised its submission so that the laboratory can define "the RNG under test" in the familiar way: **entropy sources** (`crypto.getRandomValues()` for the house; wallet-derived material or CSPRNG for the player) → **mixing** (`sessionSeed`, hash chains) → **generator** (`r_k`, `drbg()`) → **scaling** (`intBelow`, `shuffle`, `float`) → game rules. Everything from `r_k` onward is deterministic and is fed to the collection tools with fixed seeds; the entropy stage is reviewed by code reading, exactly as OS-seeded software PRNGs are handled today.

Player-provided entropy under GLI-19 §4.5.2(e) and AGCO 4.26. GLI-19 §4.5.2(e) says associated equipment "shall not influence or modify the behaviors of the game's RNG"; Ontario's Standard 4.26 requires the mechanism to be "impervious to… the player or the Operator". Both were written against a player *steering* outcomes. RAIN's framing is that the player's contribution is bounded by construction: the player commits `playerChainRoot` before the house seed is revealed and reveals `pRev_k` before seeing `hRev_k`; for *any* choice of `pRev_k`, `r_k` is uniformly distributed because `hRev_k` is unknown to the player and fixed. The player can **add** unpredictability against the house but cannot **subtract** it or bias the distribution (Claim 2 in `lab/RNG-DESCRIPTION.md` App. A). The statistical report backs this with three adversarial-player data sets — all-zero reveals, a fixed secret reused across sessions, and a 16-candidate grinding strategy — each at 10^8 outcomes for ChaCha20 and 2×10^7 for HMAC_DRBG, indistinguishable from the honest run on every aggregate statistic (§11.2). How a given laboratory classifies this is the first question for the pre-submission call (§12); the fallback framing, if a laboratory insists that "the RNG" have a single operator, is to define the RNG under test as the house node with `pRev_k` treated as *additional input* — the construction is unchanged either way.

Source: `sdk-v2/docs/LAB-SUBMISSION-REQUIREMENTS.md` §4.1–4.4, §1.9, §3.4 (with primary-source URLs in its §10).

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 seed provenance and a recognised generator, 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).
- **Generator:** `drbg(r_k, gameId, {mechanism})` — HMAC_DRBG (SP 800-90A §10.1.2, SHA-256) instantiated from the protocol-derived key, or the ChaCha20 block function — domain-separated per game and per round (§4.2). Two implementations, evaluated separately.
- **Scaling:** `intBelow` (rejection sampling), `shuffle` (Fisher–Yates), `float` (53-bit). Legacy `% N` documented with its bound, out of scope.
- **Reseed policy:** every round, unconditionally; secret material replaced from the CSPRNG at rotation (≤ 65,536 rounds).
- **Runtime health:** known-answer tests before first use, every 24 hours and on demand, with output inhibited on failure (§9.3).

9.2 Mapping to GLI-19 RNG requirements

GLI-19 area (RNG)	RAIN RNG v2.1
§3.2.1 Source-code review	Certified path ≈ 700 lines of TypeScript in `rng-core/src/{sha256,keccak,rng,drbg,bytes}.ts` plus the session and node state machines, **zero third-party dependencies**; Python mirror; every reduction in the repository inventoried (`SCALING-INVENTORY.md`). Undisclosed switches: none on the certified path; `seed32Compat` , `RAIN_CRASH_AT` , `RAIN_KAT_FAULT` disclosed and refused in production.
§3.2.2 Statistical analysis of final outcome output at 99 %	Self-run: Dieharder, NIST STS, TestU01 SmallCrush on raw words; total-distribution χ^2, blocked-χ^2 KS, serial correlation lags 1–8, runs, coupon collector, reel interplay, 52 × 52 shuffle table on scaled outcomes — §11.2. Lab repeats with its own suite on data from the shipped collection tools.
§3.2.3 Distribution / unbiased scaling	`intBelow` rejection sampling, exactly uniform (proof, `RNG-DESCRIPTION.md` App. B.1); `shuffle` uniform over permutations; legacy `% N` bound `< N/2^256` (App. B.2), outside scope.
§3.2.4 Independence	No memory of previous selections in the generator; `k` and `gameId` in every derivation; statistical independence tests in §11.2; formal Claims 1–3 in App. A of the description.
§3.2.5 Range / period	Output space 2^{256} per round; no cycling state; index space 2^{53} / 2^{32} per round; "period" replaced by a collision bound `< 2^-128` .
§3.3.1 Cryptographic strength	keccak-256 (FIPS 202), HMAC-SHA-256 (FIPS 180-4 / RFC 2104), HMAC_DRBG (SP 800-90A), ChaCha20 (RFC 8439), each with known-answer tests in the suite and at runtime.
§3.3.2(a) Direct cryptanalytic attack	Past `r_1..r_k` , all past reveals and `sessionSeed` give no information on `r_{k+1}` (needs two 256-bit pre-images).
§3.3.2(b) Known-input attack / seeding	Seeds never from time or a fixed value, never from one party; CSPRNG failure throws rather than falling back; no truncated seed path in production.
§3.3.2(c) State-compromise extension	Fresh external entropy (`pRev_k`) every round from the house's perspective; compromise of one party's whole chain does not enable bias; secret material rotated at ≤ 65,536 rounds; DRBG instance lives one round.
§4.5.2 Game outcome rules	Outcomes consumed in cursor order; no re-draw for the same *k* (idempotent reveal, 409 on conflict); aborts logged and visible; no timing or channel dependence.
§2.3.2 Control-program self-verification	Node compares the SHA-256 of every runtime artifact against `lab/FINGERPRINTS.json` at startup and every 24 h; strict mode inhibits output on mismatch.

This table is a design mapping prepared by the authors and expanded into standards vocabulary in `lab/RNG-DESCRIPTION.md` ; it is not a laboratory's finding. Formal evaluation is on the roadmap (§12).

Source: `sdk-v2/lab/RNG-DESCRIPTION.md` §1–§11; `sdk-v2/docs/LAB-READINESS.md`.

9.3 SP 800-90A, SP 800-22 and runtime self-tests

Known-answer tests. The HMAC_DRBG implementation is checked against **32 NIST CAVP HMAC_DRBG SHA-256 no-reseed cases** (`packages/rng-core/test/cavp-hmac-drbg-sha256.json`), SHA-256 against FIPS 180-4 vectors, HMAC against RFC 4231, ChaCha20 against RFC 8439 §2.3.2, §2.4.2 and A.1, keccak-256 against the Ethereum vectors. Cross-language vectors (`drbg-vectors.json`: 12 streams × 7 indices, `intBelow`, `shuffle`, ceremony) are generated from the TypeScript reference and asserted by the Python mirror's 26 tests.

Runtime health tests (SP 800-90A §11.3; GLI-19 §2.3.2). `@rain/rng-node` runs 11 known-answer tests — SHA-256, HMAC-SHA-256, four CAVP HMAC_DRBG cases, two ChaCha20 vectors, keccak-256, and the RAIN stream vectors for both mechanisms — **before first use, every 24 hours and on** `GET /selftest`. On failure the node refuses to start or, if configured, enters an error state in which every RNG endpoint returns `503 selftest_failed` and `/healthz` reports `kat: "fail"`; bits generated during testing are never output. The same cycle compares runtime artifacts against the fingerprint manifest.

Statistical batteries. NIST SP 800-22 (STS 2.1.2, all 15 tests, 1,000 sequences × 10^6 bits per mechanism), Dieharder 3.31.1 (full `-a` battery on an unbounded stream that cannot rewind) and TestU01 1.2.3 SmallCrush have been run on the reference implementation's output, fed through the real ceremony; results in §11.2. Because the two-party input is itself the output of keccak256 over high-entropy words, the same batteries can be run on the sequence of `r_k` values across a session to demonstrate that the ceremony introduces no structure — and, via replay, on the actual draws a production period delivered.

Source: `sdk-v2/packages/rng-node/src/selftest.ts`; `sdk-v2/lab/RNG-DESCRIPTION.md` §10; `sdk-v2/packages/rng-py/README.md`; `sdk-v2/lab/REPORT-STATISTICAL.md` §2.

9.4 Evidence package

The laboratory submission package lives in `sdk-v2/lab/` and is indexed against the headings of GLI's *Composite Submission Requirements* v2.0 §2.2 — the document that defines what a GLI RNG submission must contain, and which the other laboratories' public requirements closely track.

GLI CSR v2.0 §2.2 item	Where in the package	State
RNG source code — final, complete, compilable, commented, with edit history and compiled binaries	`packages/{rng-core,rng-session,rng-node}/src`, `packages/rng-py/rain_rng`, `dist/`; git history; readable (non-minified) vendored game engines; per-file SHA-256 in `FINGERPRINTS.json` (201 files)	Have
RNG final-outcome collection tool — same RNG as live, user-specified draw count, parsable output	`tools/collect-outcomes.mjs` — CSV; dice, roulette, coin, five-reel slot, 52-card shuffle (+ legacy `% N`), three adversarial-player modes, checkpointed; imports the built packages, re-implements nothing	Have
Raw-output collection tool — ≈ 96 Mbit un-scaled binary	`tools/collect-raw.mjs` — big-endian uint32 words from `drbg(r_k, gameId)` through the real ceremony; deterministic by seed; parallel and unbounded-stream modes for Dieharder `-g 200` / TestU01	Have
RNG description and documentation	`RNG-DESCRIPTION.md` — GLI-19 §3 order and vocabulary, §3.3.2 (a)(b)(c) answered explicitly, App. A independence/unpredictability claims, App. B scaling proofs	Have
Technical source-code description — "from instantiation to final outcome"	`RNG-DESCRIPTION.md` §2–§6 with file:line references; `SCALING-INVENTORY.md` (every `%` in the repository, classified certified / legacy / not-RNG)	Have
Statistical data analysis	`REPORT-STATISTICAL.md` — tables generated from the logs by `tools/report-tables.sh`, nothing typed by hand; raw logs, seeds and SHA-256 of every data file under `results/`; invalid (rewound) Dieharder runs retained and quarantined as non-evidence	Have (self-run; Crush/BigCrush and part of the HMAC battery not run — §11.2)
Software verification — lab recompiles, signatures must match	`BUILD.md` (pinned toolchain: Node v24.14.0, npm 11.9.0, TypeScript 5.9.3), `FINGERPRINTS.json`, `npm run lab:verify-build` — two consecutive clean builds produced identical digests for all 201 files (`ca7e4b96…`); `solc` not installed here, so contract bytecode is not yet fingerprinted	Have

GLI CSR v2.0 §2.2 item	Where in the package	State
Runtime self-verification (GLI-19 §2.3.2; SP 800-90A §11.3)	`packages/rng-node/src/selftest.ts` — KATs before first use, daily, on demand; output inhibited on failure; artifact fingerprint check; tests	Have
Game description / pay tables	Out of RNG scope; game configs vendored; rules in the operator repositories	Partial (operator's)
Platform items (§2.9 / §2.11): change control, SDLC, key management, incident management, hosting/HA/DR	`process/CHANGE-CONTROL.md`, `process/SDLC.md`, `process/KEY-MANAGEMENT.md`, `process/INCIDENT-RESPONSE.md`, `process/HOSTING-HA-DR.md`	Have / Partial (CI records, second reviewer and real-Docker HA execution to establish)
Penetration test; external audit (GLI-19 App. B.9; §6 of the requirements analysis)	—	**Missing** — third parties required

Checklist score. Against the 20-item artifact checklist in `docs/LAB-SUBMISSION-REQUIREMENTS.md` §7, the repository moved from **HAVE 4 · PARTIAL 8 · MISSING 8** (2.1.0 as first committed, 13 September 2026, morning) to **HAVE 13 · PARTIAL 5 · MISSING 2** after the `lab/` package (same day). The two remaining MISSING items — penetration test and external audit — cannot be produced in-house.

Source: `sdk-v2/lab/README.md`; `sdk-v2/lab/BUILD.md` §2–3; `sdk-v2/lab/FINGERPRINTS.json`; `sdk-v2/docs/LAB-SUBMISSION-REQUIREMENTS.md` §7.

9.5 What stays the operator's

Game math — payout tables, RTP, volatility, hit frequency, how many draws a feature consumes — 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. The example paytable shipped with the operator integration is explicitly a **mock**. 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 draw layer (§4) is versioned by the domain prefix in the key derivation (`"RAIN-RNG-v2.1|"` and successors), so that a verifier always knows which derivation to run, and old rounds remain replayable forever. Wire messages carry an explicit `v` field (currently `2`). Breaking changes to the ceremony itself would be a new major version and a new set of on-chain verifiers; none is planned. `seed32Compat` will be removed in 3.0.

10.4 Reference implementations

Language	Package	Version	Role
TypeScript	`@rain/rng-core`	2.1.0	Pure functions: chains, seeds, `r_k`, `gameSeed`, `drbg()`, `verifyRound`, `badge()`; own keccak-f[1600], SHA-256, HMAC_DRBG, ChaCha20; zero dependencies; browser / Node / workers; ESM + CJS
TypeScript	`@rain/rng-session`	2.1.0	The ceremony with persist-before-reveal and decision-before-reveal enforced; pipelining, rotation, `RainSpinClient` / `spinSeed`, transports; `RngAnchor.sol` + adapter
TypeScript	`@rain/rng-node`	2.0.0	Operator-runnable house node: WAL or Postgres, HTTP + WebSocket, `/verify`, `/healthz`, `/selftest`, `/metrics`, self-tests, HA compose
Python	`rain_rng` (`packages/rng-py`)	—	Stdlib-only, byte-identical mirror of core + draw layer for server-side studios and laboratories
Solidity	`RngAnchor.sol`, `ChannelManagerAA.sol`, `EngineV2RulesVerifier.sol`	—	On-chain commitments and the canonical `seedOf` / `outcomeOf` formulas (legacy path)

Test inventory: `rng-core` 21 offline tests (live-engine vectors; 32 CAVP HMAC_DRBG cases; FIPS/RFC KATs; statistical smoke; cross-language vectors) + 2 live against Arbitrum; `rng-session` 12 offline + 1 Solidity in a local EVM; `rng-node` 7 unit + 3 end-to-end (500 rounds over HTTP and WS; two workers on one store) + 1 crash test (SIGKILL before and after persist, torn WAL tail); `rain_rng` 26 pytest. The packages are **not yet published** to the public npm registry; the SDK is distributed from the repository.

Source: `sdk-v2/README.md` (package table); `sdk-v2/packages/*/package.json`; `sdk-v2/PUBLISHING.md`.

11. Live Status — Measured versus Target

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. The v2.1 draw layer, pipelining and rotation are implemented in the SDK and exercised by the load test and the laboratory programme; the live money deployments still run the v2.0 keccak engines and 4,096-element chains.

11.2 Measured

Item	Value	Conditions / source
Ceremony `r_k` formula live and verified against on-chain code	Byte-identical: core vectors vs `seedOf` on Arbitrum via `eth_call`	`npm run test:live`
Session-seed recompute from a live `SessionSeedCommitReveal`	tx `0x1c62012f…ba9e`, block 504,537,592 — matches	§7.3
Independent replay of live engine rounds	**540 / 540** re-derived OK, 0 bad	8 Sep 2026, `tools/engine-v2-audit.mjs`
Local wire test of Engine V2	250 spins × 3 games; 2,259 checks each, 0 fails	`services/house-node/test-engine-v2-local.cjs`
End-to-end spin latency, browser ↔ house node (topology d, un-pipelined)	**p50 92–104 ms, p95 99–176 ms** across engine games; max 125 ms in a 491-spin run	public deployment, `REAL-PLAY.md`
House node in-process RTT (money protocol)	p50 11 ms	same
In-browser round derivation (engine replay)	≤ 14 ms	same
On-chain transactions per spin	0	live deployments
Pipelined RNG cost on the `/spin` critical path — remote node, 40 ms RTT	**p50 0.35 / p95 0.50 ms** (pipelining on) vs 40.9 / 41.1 ms (off)	1,000 spins, loopback, simulated RTT, 60 ms animation gap, Node 24, JSONL WAL — `examples/slot-spin-endpoint/README.md`
Pipelined RNG cost — in-process node	p50 0.38 / p95 0.55 ms (on) vs 0.50 / 0.72 ms (off)	same run
DRBG cost	≈ 55 µs per HMAC-DRBG draw, ≈ 4.5 µs per ChaCha20 draw; three reels ≈ 0.2 ms	`docs/RNG.md` §2.5
Chain length 65,536 with automatic rotation at 90 %	Implemented, tested (`spin.test.mjs`, `session.test.mjs`); ≈ 1.3 s to build a chain	`@rain/rng-session` 2.1.0
HMAC_DRBG / ChaCha20 known-answer tests	32 NIST CAVP HMAC_DRBG cases + FIPS 180-4, RFC 4231, RFC 8439 vectors pass; 11 KATs also run at node startup / daily / `/selftest`	`rng-core/test`, `rng-node/src/selftest.ts`
Cross-language equivalence	TS ↔ Python: 12 streams × 7 indices, `intBelow`, `shuffle`, ceremony vectors; 26 pytest	`drbg-vectors.json`
Reproducible build	201 files, two consecutive clean builds byte-identical (digest `ca7e4b96…`)	`lab/BUILD.md`, `FINGERPRINTS.json`

Item	Value	Conditions / source
Crash safety of persist-before-reveal	SIGKILL before / after persist and torn WAL tail: same `hRev` replayed for the same commit, no double reveal	`rng-node/test/crash.test.mjs`
NIST SP 800-22 STS 2.1.2 — 1,000 sequences × 10^6 bits, all 15 tests, both mechanisms	**188 / 188 result lines inside the proportion confidence interval; 0 uniformity P-values < 10^{-4}** — for `chacha20` **and** for `hmac-drbg`	`lab/REPORT-STATISTICAL.md` §4
Dieharder 3.31.1, full `-a` battery, unbounded stream (`-g 200` , cannot rewind) — `chacha20`	**90 PASSED / 5 WEAK / 0 FAILED** on 95 result lines; 61 of 80 test/ntuple combinations run; ≈ 73 GB consumed; plus 15 extra `sts_serial` replicate loops: 444 PASSED / 6 WEAK / 0 FAILED	§3.2 there; WEAK ≈ 2 % expected by design
Dieharder, stream — `hmac-drbg`	**17 PASSED / 2 WEAK / 0 FAILED** on 19 result lines; 18 of 80 combinations run (partial — HMAC_DRBG streams at ≈ 1 MB/s per core); run continuing at time of writing	§3.2 there
TestU01 1.2.3 SmallCrush, file-backed without rewind, both mechanisms	**15 / 15 statistics passed** — `chacha20` p ∈ [0.06, 0.96]; `hmac-drbg` p ∈ [0.09, 0.92]; 2.27 × 10^8 words each	§6 there
Scaled outcomes (final outcome output, GLI-19 §3.2.2): dice(100), roulette(37), coin(2), five-reel slot(20), 52-card shuffle	`chacha20` : **10^8 outcomes per game**, 2 × 10^6 decks; `hmac-drbg` : 2 × 10^7 per game, 4 × 10^5 decks — **every total-distribution χ^2 passes at α = 0.01**; blocked-χ^2 KS passes on every honest set; all 20 reel-pair independence tests pass; 52 × 52 position table p = 0.10 / 0.27	§5 there
Adversarial player inputs — all-zero reveals; fixed secret across sessions; 16-candidate grinding	10^8 (chacha) / 2 × 10^7 (hmac) dice outcomes each; **indistinguishable from the honest run on every aggregate statistic**; three isolated single-statistic exceedances at the 1 % level across ≈ 15 statistics × 16 data sets, of the count and magnitude expected and not shared between sets	§5.1, §5.3 there
Fisher–Yates shuffle	52 × 52 card × position χ^2 (2,601 d.f.) p = 0.1000 (chacha) / 0.2708 (hmac); no position with p < .01; adjacency and first-card tests pass	§5.2 there
Laboratory checklist (20 artifacts)	HAVE 13 · PARTIAL 5 · MISSING 2 (from 4 · 8 · 8)	`LAB-SUBMISSION-REQUIREMENTS.md` §7

The Dieharder and NIST STS runs feed the real ceremony (`RainRngClient.commit → RainRngHouse.reveal → settle`) through `drbg()` ; nothing in the collection tools re-implements the generator. Two earlier Dieharder runs on finite files that Dieharder silently rewound are retained in the package as quarantined non-evidence, with the reason documented. The numbers above are those of the report at 04:23 UTC on 14 September 2026; the HMAC Dieharder battery was still running (32 of 80 test processes complete, 0 FAILED) and the report regenerates from the logs with `bash lab/tools/report-tables.sh` .

Source: `sdk-v2/lab/REPORT-STATISTICAL.md` §0, §3–§7, §11; `sdk-v2/lab/results/dieharder/{chacha,hmac}-stream/SUMMARY.txt`; `bash sdk-v2/lab/tools/status.sh` at 04:26 UTC; `sdk-v2/examples/slot-spin-endpoint/README.md`; `sdk-v2/README.md`; `playmarket/fe-ux/docs/engine-v2/REAL-PLAY.md`.

11.3 Not yet measured, not yet done

Item	Status	What closes it
Pipelined latency over a **real network** (operator server and RAIN node in different data centres, real players)	**Target** — the < 30 ms added-p95 goal is met only under loopback with simulated RTT	Deploy `examples/slot-spin-endpoint` against a remote `rain-rng-node` ; publish the percentiles (§12)
TestU01 **Crush / BigCrush**	**Not run** — 2^{35} / 2^{38} words (≈ 137 GB / 1.1 TB); stream runner shipped	`bash lab/tools/run-testu01-crush-stream.sh <mech> crush 8` — ≈ 2.5 h (chacha) / ≈ 19 h (hmac) on an idle 8-core host
Dieharder `rgb_lagged_sum` ntuple 14–32 (chacha)	**Not run** — ≈ 200 GB of stream	resumable runner skips finished tests
Dieharder full battery for `hmac-drbg`	**Partial** — 18 of 80 combinations at report time	same runner, continuing

Item	Status	What closes it
HMAC scaled outcomes at 10^8 (run at 2×10^7)	**Partial**	`R=1000000` in `run-outcomes.sh`
`RngAnchor.sol` deployment on a public chain; contract bytecode fingerprints	**Not done** — exercised in a local EVM only	deploy; `solc --metadata-hash none` digests into the manifest
Two-node HA under real Docker (`ha-test.sh`) and Postgres store against a live server	**Not executed** — Docker unavailable where the node was written; HA semantics proven in-process only	run on a Docker host; attach output to `HOSTING-HA-DR.md`
npm publication of `@rain/*`	**Not done** — scope to be decided	`PUBLISHING.md`
External smart-contract and reference-code audit	**Not done** — internal review only	third party (§12)
Penetration test of a reference deployment	**Not done**	third party
Laboratory (GLI-19 class) evaluation	**Not done** — design mapping (§9.2) and evidence package (§9.4) prepared	pre-submission call, then submission (§12)
Real-money deployment	**Not done** — play-money tokens only	after audit and laboratory evaluation

11.4 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 channel house that holds money is not in the SDK.** `@rain/rng-node` is the operator-runnable *RNG* node (party B); bond management, on-chain open submission and account-abstraction sponsorship remain in the Playmarket production node.
7. **Reference node not hardened** (no HSM, plaintext secrets at rest, no built-in rate limiting), and **same-entity deployments** need published commitments to keep the two-party argument (§8.3).
8. **Legacy `% N` reductions** for the on-chain quick games are byte-frozen and outside the v2.1 certification scope; their bias bound is documented (§4.3).

12. Roadmap

Milestone	Content	State
v2.1 — Standard candidate	HMAC_DRBG + ChaCha20 draw layer with CAVP vectors; 65,536-chain rotation; pipelined commit; `RainSpinClient` / `spinSeed`; `@rain/rng-node` with WAL/Postgres, `/verify`, self-tests; Python mirror; `INTEGRATION-OPERATOR.md`; statistical programme; laboratory package	**Done** (13–14 Sep 2026) except the items in §11.3
Finish the statistical programme	Dieharder `hmac-drbg` battery to completion; TestU01 Crush (both mechanisms), BigCrush if budget; HMAC scaled outcomes at 10^8; lagged-sum tail	Runners shipped; CPU time

Milestone	Content	State
Real-network latency	`slot-spin-endpoint` against a remote `rain-rng-node` across data centres; publish p50/p95 with and without pipelining, replacing the loopback figures in §5.2	Next
Docker HA execution	`ha-test.sh` under real Docker (worker killed mid-run), Postgres store against a live server; attach output to `HOSTING-HA-DR.md`	Next
npm publication	Choose the scope, `npm version` all packages together, publish in dependency order (`PUBLISHING.md`)	Owner decision
Security audit	External audit scoped to the RNG product — `RngAnchor.sol`, the `ChannelManagerAA` seed-ceremony functions, `EngineV2RulesVerifier.seedOf`, `@rain/rng-core`, `@rain/rng-session`, `@rain/rng-node` — with the inputs auditors expect (frozen commit, threat model, invariants, coverage, deployment scripts); timelock/multisig for admin keys; money-path contracts as a second scope	To commission
Penetration test	Annual third-party vulnerability assessment and pen-test of a reference deployment (GLI-19 App. B.9)	To commission
Laboratory pre-submission call	Ask how the lab classifies player-supplied entropy under §4.5.2(e)/AGCO 4.26; whether `% N` with bias `< N/2^256` passes source review for the legacy games or must be rejection-sampled; draw counts per game; whether contracts are in scope	Before submission
Lab engagement	Submit the RNG layer (this paper, `lab/`, vectors, reference code) to a GLI-19-class laboratory for evaluation independent of any game; `hmac-drbg` and `chacha20` as two implementations	After audit inputs are frozen
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	Ongoing
v3 exploration	Threshold house (multiple operators contribute `hRev`), hardware-backed chain secrets (HSM), a compact on-chain verifier for DRBG-derived outcomes; removal of `seed32Compat`	Later

Appendix A — Test Vectors

Ceremony vectors are generated from the live front-end engines by `scripts/gen-vectors.mjs` and shipped in `packages/rng-core/test/vectors.json`; they are re-checked against the deployed verifier on Arbitrum by `npm run test:live`. Draw-layer vectors are generated from the TypeScript reference by `scripts/gen-drbg-vectors.mjs` and asserted by both the TypeScript and Python suites. 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])) = 0x360d7049f585d3e6b6632096f0574664817be4e3e38b687176364fb73c26a6bb
c[0]   = keccak256(abi.encode(c[1])) = 0xc7c00c62e44af0940af4fa895d5098e9341995caeb062e1882c73b18ce239197  (roo
t)
```

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

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

A.4 Round outcome (canonical `r`, legacy reductions)

```
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 (legacy keccak domain separation)

```
pRev        = 0xf1c2fbe645543fcdec88b890bba0b0b59391dc287e4fa1396e85cf0e09a50033
hRev        = 0xa5a64e3eca9beafc3ec32842f3be63350fc67d64ab7605172bd311293014b3f6
sessionSeed = 0x162cbbbb4586b29e2b91267f1e9f9e92cb9cf5b7f96e1c325fcdb44c1d9e1ee2
channelId   = 0xf22aa1fc519a4d1de879fd583318033d707c770d5c41a5bfe7cf08894784e1ef
k           = 3170
gameId      = "foundry"   keccak256("foundry") = 0x4eb2f10301a3ed7f2c31091074ca429f73cb8c51539e1a0005e132f70b8b
b74a
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)               = 0x875a10a61e1f13fb65dfca35c4b560661d666acc070e83b18517ca3094a0
d41a
sessionWalletPk = keccak256(base ‖ "session")  = 0x05c055bd9033f342eba27f5bf22ec1adaed89b9e6b9a08f05a76f3298315
8633
openNonce       = 4560430705875
chainSecret     = keccak256(base ‖ "chain:4560430705875") = 0x49afd08131355b891f8a4de4349beba6f70bee20e13644e5f
8a0cc16d5d814ec
playerSeed      = keccak256(base ‖ "seed:4560430705875")  = 0xb10f6d3d7c6db68aa723b9f608b0f5fda14bfb69ab9f9da06
02d291ceac0bd23
```

A.7 Draw-layer vectors (v2.1) — shipped, not reproduced here for length:

- `packages/rng-core/test/cavp-hmac-drbg-sha256.json` — 32 NIST CAVP HMAC_DRBG SHA-256 no-reseed cases (personalization ∈ {0, 256 bits} × additional_input ∈ {0, 256 bits}), asserted by `test/drbg.test.mjs` and by the Python suite.
- `packages/rng-core/test/drbg-vectors.json` — 12 RAIN streams (both mechanisms, several `gameId`s) × 7 indices: `key`, `draw(i)`, `uint32`, `uint53`, `float`, `intBelow(i, n)` with words consumed, `shuffle` of a 52-element array, plus ceremony vectors; generated by the TypeScript reference, asserted byte-for-byte by `rain_rng` (`packages/rng-py/tests`).
- FIPS 180-4 SHA-256, RFC 4231 HMAC, RFC 8439 §2.3.2 / §2.4.2 / A.1 ChaCha20 vectors in `test/drbg.test.mjs`; the same 11-KAT subset runs inside `rain-rng-node` at startup.

Source: `sdk-v2/packages/rng-core/test/`; `sdk-v2/packages/rng-py/README.md`; `sdk-v2/lab/RNG-DESCRIPTION.md` §10.

Appendix B — API Surface

`@rain/rng-core` **2.1.0** (pure functions, zero dependencies)

```
keccak256(bytes|hex) · keccakWord(x) · keccakWords(...w) · gameIdHash(gameId)
houseSeedCommit(houseSeed) · sessionSeed(houseSeed, playerSeed, channelId) · channelId(manager, player, operato
r, 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) → bytes32 · outcomeUint(...) · outcomeMod(..., N) [legacy % N] ·
modN(r, N)
gameSeed(pRev, hRev, sessionSeed, channelId, k, gameId) [legacy Engine V2] · outcomeWith(pRev, hRev, sessionSee
d, channelId, ...extra)
verifyRound({pRev, hRev, sessionSeed, channelId, k, expected?, N?, gameId?, pAnchor?, hAnchor?}) → {ok, r, valu
e, matches, pChainOk, hChainOk}
deriveMsg(manager, chainId) · materialFromSig(sig) · chainSecretFor(base, openNonce) · playerSeedFor(base, open
Nonce)
randomBytes32() · badge({url?, dark?}) → {text, html, markdown, svg, attribution, url, version}

drbg(r, gameId = "", { mechanism?: "hmac-drbg" | "chacha20" }) → RainStream
  RainStream: r · gameId · mechanism · key · draw(i) · drawBytes(i) · uint32(i) · uint53(i) · float(i)
              intBelow(i, n) · intBelowEx(i, n) → {value, used} · shuffle(i, arr) → {result, used} · cursor(sta
rt?) → RainCursor
  RainCursor: index · next() · uint32() · float() · intBelow(n) · shuffle(arr)
deriveDrawKey(r, gameId) · HmacDrbg · chacha20Block(key, counter, nonce) · sha256 · hmacSha256
RNG_DOMAIN_PREFIX = "RAIN-RNG-v2.1|" · DEFAULT_MECHANISM = "hmac-drbg"
seed32Compat(r)   — DEPRECATED, development only, warns
```

`@rain/rng-session` **2.1.0** (the ceremony, transport-agnostic)

```
new RainRngHouse({persist, chainLen? = 65536, maxInFlight? = 2, rotateAt? = 0.9, houseSeed?, chainSecret?, meta?})
  .getTerms() → Terms · .open(Open) → Opened · .reveal(Commit) → Reveal · .rotateTerms() → Terms · .rotate(Open) → Opened
  .needsRotation · .roundsLeft · .rotations[] · .history[] · .transcript()
new RainRngClient({chainLen? = 65536, maxInFlight? = 2, playerSeed?, chainSecret?})
  .open(Terms) → Open · .opened(Opened) → sessionSeed · .commit(decision) → Commit · .receive(Reveal) · .settle(Reveal?) → RoundResult
  .rotate(Terms) → Open · .rotated(Opened) → sessionSeed · .abort() · .fairness(result) · .latencyStats() → {count, p50, p95, max, mean}
localOpen(house, client) · localRound(house, client, decision, N?) · localRotate(house, client) · percentiles(xs)

RainSpinClient(transport, {chainLen?, maxInFlight?, decision?, onRotate?})           — one-call ceremony for operators
  .open() → sessionSeed · .spinSeed(k?, decision?) → {r, k, proof, latencyMs, decision, stream(gameId, opts?)}
  .prefetch(decision?) → k · .rotate() · .needsRotation · .latencyStats() · .close()
RainHouseEndpoint(house, {onRotate?}) · inProcessTransport(ep) · wsTransport(ws) / wsServe(ep, sock) · httpTransport(url) / httpHandler(ep)
RngAnchorAdapter(address, signer): commit · bindPlayer · revealSeed · submitReveals · session(sessionId)   (RngAnchor.sol — not deployed)

Wire (v: 2): Terms{houseSeedCommit, houseChainRoot, chainLen, meta?} · Open{playerSeed, playerChainRoot, chainLen, termsHash}
             Opened{sessionId, houseSeed, sessionSeed} · Commit{sessionId, k, decision, pRev} · Reveal{sessionId, k, hRev, claimed?}
             SpinRequest: terms | open | commit | rotateTerms | rotate
```

`@rain/rng-node` 2.0.0 (operator-runnable house node, party B)

```
POST /v2/terms             → Terms            POST /v2/open/:termsId  (Open) → Opened
POST /v2/reveal  (Commit)  → Reveal           — WAL append + fsync (or Postgres COMMIT) before hRev is returned; same (k, decision, pRev) → same hRev; different → 409
GET  /v2/reveal/:sid/:k    → Reveal           — recovery: the reveal for an already-persisted round
GET  /verify/:sid/:k       → public commitments, persisted commit, reveal, recomputed r + chain checks, status revealed | committed_not_revealed  (CORS *)
GET  /healthz              → 200/503 {ok, store, chainHeadroom, kat, selfTest{...}, attribution}
GET  /selftest             → runs the 11 KATs + fingerprint check now; 503 on failure (RNG endpoints inhibited)
GET  /metrics              → Prometheus        WS /v2/ws — same operations as JSON envelopes
Env: RAIN_OPERATOR_ID · RAIN_DATA_DIR | Postgres · RAIN_CHAIN_LEN (default 4096 in the node) · RAIN_OPERATOR_KEY[_FILE] · RAIN_ANCHOR_KEY
     RAIN_FINGERPRINTS[_STRICT] · RAIN_KAT_FAIL_EXIT · (RAIN_CRASH_AT, RAIN_KAT_FAULT — tests only, refused in production)
HA: N stateless workers behind nginx sharing one Postgres (docker-compose.ha.yml; ha-test.sh — not yet executed under real Docker)
```

`rain_rng` (Python, stdlib only)

```
rain_rng.core: house_seed_commit · session_seed · channel_id · hash_chain · chain_root · chain_ok · verify_chain_element
               outcome · outcome_mod · game_seed · outcome_with · verify_round · material_from_sig · chain_secret_for · player_seed_for
rain_rng.drbg: drbg(r, game_id, mechanism) → RainStream: draw · uint32 · uint53 · float · int_below · shuffle · cursor
               HmacDrbg · chacha20_block · derive_draw_key · seed32_compat (deprecated)
```

On-chain (Arbitrum One)

```
ChannelManagerAA 0x60743006c2a5Dd5b9907CA37e7381854ff860959 — openChannel, submitReveals, demandReveal (60 s), forceClose, challenge
EngineV2RulesVerifier 0x5d6048AB261e6151AB44fE6053e7AEb35Cb5C1B1 — seedOf(...) pure, engineHash(gameIdHash) view
RngAnchor.sol (not deployed; local-EVM tested) — commit, bindPlayer, revealSeed, submitReveals, seedOf, outcomeOf, gameSeedOf, verifyChainElement
```

Source: `sdk-v2/packages/rng-core/src/{index,drbg,rng}.ts`; `sdk-v2/packages/rng-session/src/{index,spin,anchor}.ts`; `sdk-v2/packages/rng-node/src/{server,house,config}.ts`; `sdk-v2/packages/rng-py/README.md`.

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 and seed halves (`rotateTerms → rotate → rotated`) before the current chains are exhausted; default at 90 % of 65,536.
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.
Draw layer	`drbg(r, gameId, {mechanism})`: the per-(round, game) keyed generator and its scaling functions; the object of laboratory certification.
DRBG / HMAC_DRBG	Deterministic random bit generator (NIST SP 800-90A); here HMAC_DRBG with SHA-256, instantiated from a protocol-derived 256-bit seed, or a ChaCha20 block function as the alternative mechanism.
Domain separation	Including a context string (version prefix, game id) in a key derivation so that different uses never share randomness.
`gameSeed_k`	Legacy Engine V2 seed: `r_k` extended with `keccak256(bytes(gameId))`; byte-identical to `EngineV2RulesVerifier.seedOf`.
GLI-19 / GLI CSR	Gaming Laboratories International standard for interactive gaming systems (Chapter 3: RNG); Composite Submission Requirements defining the submission package.
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).
`intBelow`	Unbiased integer in [0, n) by rejection sampling on 32-bit words; the certified scaling function.
`maxInFlight`	The number of rounds that may be committed but not yet settled (default 2); bounds pipelining on both sides.
Persist-before-reveal	House rule: the player's commitment is durably stored before `hRev_k` is released.
Pipelining / `prefetch`	Sending the commit for round *k+1* while round *k*'s animation plays, so the reveal is already held when `spinSeed(k+1)` is called.
`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 (STS).
SP 800-90A / 90B / 90C	NIST recommendations for DRBG mechanisms / entropy sources / RBG constructions. RAIN uses the 90A HMAC_DRBG *mechanism*; it does not claim 90B or 90C conformance.
`spinSeed(k)`	The whole ceremony for one round in one call (`RainSpinClient`), returning `r`, the proof and a `stream(gameId)` factory.
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.
`/verify`	`GET /verify/:sessionId/:k` on a RAIN node: the public inputs and recomputation of any round for independent verification.

RAIN RNG Whitepaper v1.1 · 14 September 2026 · pre-audit · © 2026 RAIN Risk Markets · Reference code: `@rain/rng-core`, `@rain/rng-session`, `@rain/rng-node`, `rain_rng` (MIT + attribution). Live: playmarkets.bet · maycasino.xyz · rainriskmarkets.com

RAINRISKMARKETS.COM

PLAYMARKETS.BET · MAYCASINO.XYZ

Powered by RAIN RNG