ChatGPT

Deterministic Interactive Computation in the Browser

Outline

Introduction & Motivation

- **Core claim:** RedByte OS demonstrates that interactive simulations in a browser can be made **fully deterministic** at the logic level, enabling exact reproducibility of user-driven sessions.
- **Motivation:** Traditional interactive systems (e.g. circuit simulators, games) are ephemeral and unpredictable. RedByte OS was created to remove this ambiguity – to make every run replayable and debuggable.
- **Supporting artifacts:** A formal *Determinism Contract* defines the guarantees and scope [1], and a working implementation in a logic circuit simulator validates these guarantees.
- **Not claimed:** This work does **not** promise improvements in performance or UI/UX; it strictly focuses on semantics and reproducibility (determinism "as a semantic contract, not a marketing claim" [2]).

Problem Statement (Non-Determinism and Opaqueness)

- **The challenge:** Interactive applications are traditionally **non-deterministic** and **opaque**. Tiny differences in input timing, scheduling, or environment lead to divergent outcomes, and internal state changes are hidden from users.
- **Consequences:** Bugs in such systems are hard to reproduce; educational demos might not behave the same way twice. Without determinism, developers and researchers cannot reliably **replay or inspect** what happened in an interactive session.
- **Illustration:** For example, two users toggling circuit inputs in a simulator might get different results if the engine's scheduling or random seeds differ – a common issue this system aims to eliminate.
- **Scope of problem:** Sources of non-determinism include user input timing, concurrency, uncontrolled randomness, and unlogged state changes. These make interactive sessions **un-repeatable** and their internals untraceable, motivating a need for a deterministic framework.

Determinism as a Semantic Property

- **Core claim:** RedByte OS treats **determinism as a first-class semantic property** of the simulation. Formally, given the same initial state and the same sequence of events, the system will always reach the same final state [3].
- **Unit of determinism:** The execution *session* – defined by an initial circuit configuration and an ordered event log – is the unit over which determinism is guaranteed. The contract states this precisely (hash of execute(circuit, events) is invariant) [4].
- **Scope & boundaries:** Determinism is guaranteed **only** for the logic simulation state and event processing. Other aspects are explicitly out-of-scope: rendering, real-time timing, external I/O, etc., are **not** deterministic or not recorded [5] [6]. This boundary prevents overclaiming.

- **Supporting artifacts:** The *Determinism Contract* document codifies these guarantees and non-goals, ensuring clarity on what "deterministic" means in this system [7] [8].
- **Not claimed:** Cross-version determinism or cross-implementation determinism is **not** promised [9] [10] – if the engine changes or a different engine is used, results may differ unless explicitly maintained. Similarly, determinism here doesn't imply any security (no tamper-proofing) [11] or UI consistency.

Event-Bound Execution Model

- **Core claim:** The system adopts an **event-bound execution model**, chopping interactive execution into a sequenced log of discrete events. Each event represents an *intent-level action* (e.g. loading a circuit, toggling an input, a simulation tick) that causes a state transition.
- **Deterministic sequencing:** By recording and replaying **the exact sequence** of events in order, the simulation's behavior becomes reproducible. The replay runner applies events one by one (from an initial `circuit_loaded` through each input or tick) and guarantees the same result state for the same event sequence [12].
- **Rationale:** Treating events as atomic steps with well-defined boundaries means the system can capture state snapshots at each boundary. There are no hidden intermediate mutations; every state change corresponds to a logged event. This eliminates sources of nondeterminism like uncontrolled concurrent updates.
- **Supporting artifacts:** A versioned **event log** format (EventLog v1) was defined, containing an append-only list of typed events [13]. The recorder component captures events with timestamps in a canonical order, and the replay engine consumes them in strict sequence, ensuring purity (same log → same outcome) [14].
- **Not claimed:** The model does not attempt to preserve *real-time* temporal spacing of events – only order. Wall-clock timing is not replayed (timing differences don't affect logic results [15]). Also, no branching or alternative event orders are considered; the focus is on one linear event history at a time. Concurrency and parallel events are not in scope.

Verifiable Replay via State Hashing

- **Core claim:** RedByte OS enables **verifiable replay** – any recorded session can be cryptographically verified to replay identically. After replaying, the system produces a hash of the final state and compares it to the hash from the original run; a match proves the runs are identical [16].
- **Mechanism:** The final circuit state is reduced to a **SHA-256 hash** (using a canonical serialization of state). If `hash(liveExecution) === hash(replayedExecution)`, determinism holds by definition [17]. This hash comparison is collision-resistant and order-sensitive, giving high confidence in equality of states.
- **State normalization:** To make hashing reliable, the system normalizes the state before hashing (e.g. sorting collections, stable key order) to eliminate spurious differences [18]. This ensures that semantically identical states yield the same hash, addressing risks like iteration order non-determinism [19].
- **Supporting artifacts:** The implementation provides a `verifyReplay()` API which runs the recorded events on a fresh engine, computes hashes on both live and replayed outcomes, and checks equality [20]. The Determinism Contract explicitly includes hashing as the proof mechanism for identical execution [21].
- **Not claimed:** The hashing mechanism is used **only as an identity check**, not a security measure. The contract notes that logs are not cryptographically signed; a matching hash confirms identical

behavior, but does not guarantee the log wasn't tampered with [11]. In other words, we verify *equality*, not authenticity. Additionally, this method doesn't prevent someone from constructing a different sequence that by chance leads to the same final state (practically infeasible due to SHA-256, but no stronger integrity guarantees are made).

Inspectable Time Travel as a Query Primitive

- **Core claim:** The system makes execution history **inspectable**: time travel is provided as a pure *query operation* on the event log, allowing developers to examine any intermediate state without altering the execution [22]. It's as if the simulator's timeline is a database you can query by event index.
- **Capabilities:** Given a recorded event log, one can retrieve the circuit state after event *i* (for any i in 0…N) deterministically. Functions like `getStateAtIndex(initialState, log, i)` replay the first *i* events and return a snapshot [23]. Navigation primitives `stepForward` and `stepBackward` move through the timeline by one event in either direction [24] [25].
- **Stability:** These time-travel operations are stable and repeatable. Stepping forward then backward (or vice versa) returns you to the same state, demonstrating no side-effects or hidden state accumulation [26]. Each snapshot is immutable and derived solely from the log prefix, so the query can be repeated arbitrarily and yield the same result.
- **Diff and inspection:** The system also supports *state diffing* – comparing two snapshots to highlight changes (e.g. which node's output flipped between event i and j) [27] [28]. This helps users pinpoint the effects of each event. A developer-facing UI (the "Determinism Panel") allows interactive use of these features (navigate with buttons, view state hashes, etc.), making time-travel debugging accessible.
- **Supporting artifacts:** Milestone C delivered the inspector APIs and a minimal UI for time travel, which is integrated as a developer tool (Ctrl+Shift+D in the app) [29] [30]. The Determinism Contract guarantees that time-travel snapshotting is consistent (state(t) → state(t+1) is stable, and reversible) [31], treating the log as an immutable record.
- **Not claimed:** Time travel in this system is **read-only**. We do not support modifying history or forking new execution branches from the past – no "alternate timeline" editing is possible (the log itself is never altered during inspection). Also, this feature is not optimized for performance on huge logs (there's no sophisticated indexing or caching yet, so stepping through very large event histories may be slow [32]). It's intended as a correctness aid, not a fully polished end-user feature at this stage [33].

Operational Validation in a Browser Environment

- **Core claim:** Determinism isn't just a theoretical guarantee – it has been **validated in real browsers** with the full system in operation. The entire record→replay→verify→time-travel loop runs reliably in a live browser setting, confirming that our guarantees hold under practical conditions.
- **Browser compatibility:** The implementation was adjusted to use browser-native facilities (e.g. Web Crypto API for hashing) and tested on multiple browsers. For instance, replacing Node.js's sync SHA-256 with `crypto.subtle.digest` (async) required ensuring all async calls are awaited properly [34]. This change was validated by computing hashes in the browser and checking they remain consistent and deterministic [35].
- **End-to-end test:** A sample workflow was executed in Chrome and Firefox: a user recorded a session (building a small logic circuit, toggling inputs), stopped recording, then verified the replay via the UI. The live run's hash and the replay's hash matched exactly (e.g. `f718ecf89915...` for both) and the UI indicated "✓ Deterministic" [36]. The log was exported to JSON, the page reloaded, and the log re-imported; the replay still produced the same hash, proving portability and consistency.

- **Locking the guarantees:** Upon such validation, the project "locked in" the determinism guarantees as a stable foundation. The event log format (v1) is now fixed – no changes will be made that break determinism without introducing a new version and updating the contract [37]. In effect, the **Determinism Contract is now binding** for the released system, meaning any future changes must respect these same guarantees or explicitly version off. This provides confidence that determinism won't regress accidentally [38].
- **Supporting artifacts:** Automated test suites accompany this validation (covering hashing, replay, time-travel, etc.), and a developer-facing Determinism Panel serves as a live demo of the guarantees (showing hash comparisons in real time). Together, these give a high degree of trust that "deterministic interactive computation" is not just an idea but a working reality in the browser.
- **Not claimed:** We do not claim *universal* browser support or extreme stress testing at this point. Only modern, standards-compliant browsers (Chrome, Firefox, Safari recent versions) were explicitly validated; older or uncommon browsers have not yet been verified [39]. Performance under very large circuits or logs is not guaranteed – our testing focused on functional correctness with reasonably sized sessions [40]. Also, the determinism tooling is currently a developer tool (dev-mode panel) rather than a polished user-facing feature [41].

Comparison to Prior Work (Debuggers, Replays, Logs, Educational Tools)

- **Deterministic replay in context:** The concept of replaying executions is not new – from software debuggers to video games, various systems have tackled it. Traditional *debuggers* (e.g. GDB) typically execute programs stepwise but usually don't enforce determinism unless carefully controlled (some advanced debuggers like rr can record and replay program executions, but at a very low level and high overhead). By contrast, RedByte operates at a higher semantic level (logic circuit events), which is more tractable and domain-specific, allowing built-in determinism with less overhead.
- **Game replay systems:** Many simulation games (for example, strategy games) use deterministic logic and input logs to enable replay and networking. RedByte's approach is analogous: like a game that records user inputs and random seeds to replay a match, we record circuit inputs and ticks to replay a simulation. The key difference is **generality and inspection**: game replays are often opaque (a "demo" plays out without letting the user intervene or inspect state deeply), whereas our system provides a general mechanism to inspect and verify each step. Also, we formalize the guarantee (via a contract and hashing) which game systems typically do not publish openly.
- **Simulation logs & provenance:** In scientific simulations or data processing pipelines, logging every step for reproducibility is a known practice. Our work brings that rigor to an interactive GUI-based simulation. Unlike typical log files that just record outputs, we capture all *inputs* (events) that lead to outcomes, which is a stronger form of provenance – anyone with the log can replay the entire session and get identical results, not just analyze what happened after the fact. In prior systems, one might have to instrument code or use specialized tools to achieve this level of replay; here it is a built-in feature of the environment.
- **Time-travel debugging:** Environments like Elm and Redux (in web development) have popularized *time-travel debugging*, where a history of user actions is recorded and can be replayed or stepped through. Our system is philosophically similar, but whereas those frameworks assume pure functions and are limited to app state, RedByte OS deals with a potentially complex simulation with internal state. We had to ensure purity by design (through careful state handling and event logging). We also add the dimension of **verifiability** (cryptographic hashes), which is not a standard part of time-travel debuggers. Additionally, our explicit contract distinguishes our approach – we're not just saying "it usually works," we have a precise definition and tests proving it always works within stated conditions.

- **Educational tools:** Prior educational tools (e.g. visual circuit simulators, algorithm animation software) often allow stepping through an example execution, but rarely do they allow a user's arbitrary session to be recorded and perfectly replayed elsewhere. RedByte's determinism and exportable logs fill this gap: an instructor can perform a live demo and then give the students the exact log of that demo, which they can replay deterministically on their own. This goes beyond traditional "save file" or "undo history" mechanisms by guaranteeing the replay will mirror the live demo state-for-state. In comparison to typical classroom tools, this could be seen as merging the benefits of a recorded video (exact reproducibility) with an interactive model (students can pause, inspect, etc.).
- **Summary of novelty:** In sum, while elements of our work echo prior ideas (record/replay, state diffing, etc.), the integration of **deterministic replay, time-travel inspection, and formal verification in a single browser-based system** appears to be unique. We leverage lessons from prior work (e.g. using event logs like games do, using state immutability like functional debugging) and combine them with new parts (a formal contract and hashing verification) to push the boundary of what a general-purpose interactive simulation can guarantee.

Limitations & Explicit Non-Goals

- **Performance**: The system makes **no promise of real-time performance or timing accuracy** [15]. Determinism is about final state and logical consistency, not about running at the same speed or meeting deadlines. For instance, replay might run faster or slower than the original session; this does not affect correctness of outcomes.
- **Cross-Version & Compatibility**: We do **not guarantee determinism across different engine versions** [9] or across alternative implementations of the engine. The guarantee holds when using the same version of the RedByte logic engine that declares compliance with the contract. If we update the engine in ways that affect simulation logic, those changes might produce different outcomes (hence the event log is versioned, and old logs may not work identically on a new engine unless compatibility shims are provided).
- **Security**: Our determinism guarantee is **not a security guarantee** [11]. We assume the event log and initial state are trusted – the system does not attempt to detect maliciously altered logs. Hashes will reveal if a replayed log yields a different state than expected, but they won't stop someone from modifying a log or guarantee who created a log. There is no signing of logs or encryption in place. In short, we ensure consistent execution, not tamper-proof execution.
- **UI and Rendering**: The **user interface is out of scope** for determinism [42]. Graphical or interactive elements (animations, widget states) might not replay exactly, and we do not attempt to capture or replay UI behavior. The focus is the underlying simulation state. This means, for example, that the position of the cursor or timing of a button highlight during recording are not recorded. Such non-determinism doesn't affect the circuit logic, which is why we exclude it.
- **No Multi-User Sync**: The system is single-user and local. We do not tackle **networked determinism or collaborative consistency** (that is a non-goal per the contract) [43]. If two people were to try to replay a session together, our contract does not cover divergence between their views. Any future collaborative features would need a separate protocol; for now, determinism is assured only in a single-node context.
- **Dev-Tool (Not Product Feature)**: The deterministic replay and time-travel features currently live in a developer tool panel. Making this a slick end-user feature (with polished UI, persistence, etc.) was *not* a goal of the core system milestones. The panel is functional but minimal, intended for proof-of-concept and internal use. User-friendly extensions (like a one-click "replay my session" feature in the app) are left for future iterations.

- **Implicit Non-determinism**: We assume the underlying JavaScript runtime doesn't introduce hidden nondeterminism. Issues like floating-point precision or garbage collection timing are beyond our scope; we avoid them by design (e.g. using integer logic values only, no timing-based logic) [44]. However, if the host environment had a bug causing nondeterministic behavior at the hardware/VM level, that's outside what we address.
- **Summary**: By explicitly listing these non-goals, we ensured from the outset that we focus on delivering the core deterministic experience without misleading about areas we haven't solved. These boundaries (no performance guarantee, no cross-version promise, etc.) are a conscious part of the contract to keep it *honest* and achievable.

Implications for Education and Research

- **Reproducible teaching modules**: With this deterministic execution system, an educator can create an interactive lesson (e.g. a logic circuit demo) and be confident that every student who replays it will see the exact same sequence of states and outputs. This is akin to providing a code example that always runs the same, but applied to interactive behavior. It could greatly enhance remote learning or flipped classroom models – students can "play back" the instructor's live demo on their own, step through each event, and inspect how a conclusion was reached. The *intent-level* log preserves the decision-making process, not just the end result.
- **Lowering the barrier to experimentation**: Students can also record their own sessions and share them with instructors or peers for troubleshooting. Instead of "it's not working on my computer" (common in labs or assignments), the exact interaction can be shared and replayed. This could shift how debugging and help sessions are done in educational contexts – the log becomes a shareable artifact of the learning process.
- **Research on debugging and PL**: For programming languages (PL) and systems researchers, RedByte OS offers a working example of *deterministic interactive computation*, which is a rarity. It shows that, by carefully constraining scope and using formal logs, one can bring the rigor of reproducible batch computation to an interactive GUI scenario. This might inform new research on debugging tools (e.g. integrating similar deterministic replay into IDEs or live programming environments) or on *live programming* systems where understanding program state over time is critical.
- **Reproducible research demonstrations**: Often, when researchers publish interactive systems or UI techniques, it's hard for others to experience them exactly as described. A system like this could allow researchers to publish **interactive artifacts** along with papers – a recorded log of an interactive experiment that readers can load up and replay exactly. This would be a boon for reproducibility in HCI or systems research where setups are interactive and typically hard to package.
- **Provable properties**: The strong semantic guarantees also mean we could, in the future, prove properties about the interactive session (e.g. invariant checking over the event log). For now, the system is tested but not formally verified end-to-end. However, having a deterministic model could enable model checking of interactive workflows or formal verification of certain safety properties (because nondeterminism often complicates formal reasoning). In education, this could mean automatic grading of interactive lab exercises by comparing student-submitted logs/hashes to an expected outcome.
- **Broader impact**: The success of this approach in a browser environment suggests that even web-based interactive tools (often seen as impossible to fully determinize due to browsers' sandbox and scheduling) can be made reproducible. This might inspire similar determinism efforts in other domains – for example, deterministic web-based data visualization tools or game engines that

guarantee replayable user sessions. It reinforces the notion that **determinism is a design choice**: by designing the software with a clear contract and avoiding certain sources of variability, we can achieve reproducibility where it wasn't previously expected.

Future Work (Speculative)

- **User-Facing Features**: A clear next step is to graduate the determinism capabilities from a dev tool to a polished feature for all users [45]. For example, adding "Save Session" and "Replay Session" buttons in the main UI would let users record their work and share or revisit it easily. This involves packaging the current recording/export under a friendly interface, adding proper file handling, and perhaps cloud integration for sharing logs. It's speculative but straightforward now that the core is stable.
- **Collaboration & Networking**: Enabling deterministic collaboration (multiple users) is an ambitious extension. The current system assumes one event stream from one user. A possible future approach is to merge multiple event streams in a controlled way or have a distributed protocol where each client's actions are logged and applied in a deterministic order (similar to how some multiplayer games use lock-step simulations). This would require careful design to avoid race conditions. While **not covered by the current contract** [43], exploring this could open RedByte OS to multiplayer or classroom collaboration scenarios – e.g. a teacher and student interacting with the same circuit and both having a consistent replay log of the session.
- **Performance Optimization**: As logs grow large, replaying from scratch for every query (time travel step) will become a bottleneck. Future work could introduce **checkpoints** or state caching. For instance, the system might periodically store a snapshot of state so that jumping to event 900 of 1000 doesn't require re-running all 900 events from the start. Any such caching would itself need to be deterministic and transparent. Another area is optimizing the hashing and normalization – perhaps using WebAssembly for speed or allowing selective hashing of changed parts of state for quicker verification.
- **Cross-Version Log Migration**: Right now, a log is only guaranteed to work with the same version of the engine. In the future, if we release engine v2 with new features, we'd need a way to migrate or still accept v1 event logs (or explicitly convert them). Planning a **log schema evolution** (with converters or dual-version support) is future work, guided by the rule that any semantic changes must increment the log version and preserve old behavior or document breaking changes [38]. This ensures that determinism can extend across versions in a controlled manner.
- **Formal Verification**: We have high confidence in determinism due to testing and hashing, but one could take it further. Future research work could formalize the semantics of the event log and the engine, and attempt a proof that for all possible event sequences, the engine's state transition function is deterministic. This could involve writing a model in a proof assistant or model checker. Achieving a machine-checked proof of determinism would be a strong validation (and relatively feasible given the single-threaded, discrete nature of the model).
- **Extended Scope (Cautiously)**: Another speculative path is expanding what's recorded to inch closer to capturing *full* user experience. For example, recording UI interactions or user timing to allow a "video-like" replay that also reproduces how the user saw it. This is tricky: it risks reintroducing nondeterminism and goes beyond our current contract (which deliberately left UI out [46]). If attempted, it would be as an **opt-in extension** with its own guarantees (e.g. recording pseudo-random seeds for any animations, or locking frame rates). This is largely outside the current plan, but mentioned as an idea for how the principles here could influence other layers of the stack.
- **Conclusion of future outlook:** Any future work will adhere to the philosophy that made this project successful: clearly define the guarantees and non-goals, and validate changes rigorously. The

foundation built in RedByte OS's deterministic engine (now locked and proven) makes these explorations possible – we can add features like collaboration or richer replay confident that the core will behave predictably [47] . In summary, deterministic interactive computation in the browser, once an unlikely proposition, is now a concrete reality and a springboard for further innovation.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [15] [16] [17] [19] [21] [22] [31] [38] [42] [43] [44] [46] CONTRACT.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/f22c383c085c019f2aec1177b7d72f57cad17303/docs/determinism/CONTRACT.md

[12] [14] [20] milestone-b.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/f22c383c085c019f2aec1177b7d72f57cad17303/docs/determinism/milestone-b.md

[13] [18] milestone-a.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/f22c383c085c019f2aec1177b7d72f57cad17303/docs/determinism/milestone-a.md

[23] [24] [25] [26] [27] [28] [29] [30] [32] [33] milestone-c.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/f22c383c085c019f2aec1177b7d72f57cad17303/docs/determinism/milestone-c.md

[34] [35] [36] [37] [39] [40] [41] [45] [47] milestone-d.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/f22c383c085c019f2aec1177b7d72f57cad17303/docs/determinism/milestone-d.md