ChatGPT

Deterministic Interactive Computation in the Browser

Outline

- **Introduction & Motivation:**
- **Core Claim:** Present a browser-based logic simulation system (RedByte OS) that achieves **deterministic interactive computation**, meaning the same user interactions produce the same outcomes every time, solving reproducibility and debuggability issues in interactive simulations.
- **Supported by:** RedByte OS's formal *Determinism Contract* defines that given identical initial state and sequence of inputs, the simulation always reaches the same final state [1]. This deterministic guarantee is *provably enforced* via the system's design and tests (e.g. cryptographic state hashes to compare runs).

- **Not Claimed:** This is **not** a claim of performance improvement or a "magic" new platform. The introduction avoids any hype; it simply motivates why determinism in interactive computing is valuable. No claims are made beyond documented guarantees (e.g. nothing about security or multi-user sync, which are out of scope [2] [3]).

- **Problem Statement (Why Interactive Systems Are Traditionally Non-Deterministic and Opaque):**

- **Core Claim:** Traditional interactive applications (e.g. simulations with user-driven events) often exhibit nondeterminism and lack transparency, making behavior hard to reproduce or inspect. This section frames the challenges: timing variability, uncontrolled scheduling, hidden state changes, etc., lead to *opaque execution* where bugs and outcomes can't be easily replicated or explained.
- **Supported by:** We note common sources of nondeterminism: UI rendering and animations, variable frame timing, asynchronous browser tasks scheduling, and timing of user inputs [4]. These factors mean that two "identical" interaction sessions might not yield identical results in ordinary systems. Without special handling, interactive runs are not *recorded or replayable*, leaving developers and researchers with an opaque execution history.

- **Not Claimed:** We do not claim that all nondeterminism can or should be eliminated in every context. Instead, the problem statement highlights why it's *undesirable in a specific domain* (browser-based logic simulations). It doesn't claim prior systems entirely lacked solutions, but underscores their limitations (e.g. logging or debugging tools exist, but typically without **semantic determinism** or built-in replay fidelity).

- **Determinism as a Semantic Property (Unit of Determinism, Scope, Boundaries):**

- **Core Claim:** Define determinism at the *semantic level* of the simulation. The **unit of determinism** in RedByte OS is an entire execution session: given the same initial circuit and the same ordered

sequence of events, the simulation's resulting state is guaranteed identical [5]. This section clarifies the *scope* of that guarantee and the strict boundaries (what is and isn't covered).

- **Supported by:** The Determinism Contract explicitly enumerates what parts of the system are deterministic (the logic engine, recorded event sessions, replay/time-travel) and what are not (UI rendering, wall-clock timing, external I/O) [6]. We cite the precise contract language to formalize the guarantee. We also explain the *mechanisms enforcing semantic determinism*: e.g. canonical state representation and elimination of hidden nondeterministic factors (sorting of iteration order, removing random sources, injecting timestamps, etc. per Contract §5) [7] [8]. These design choices make determinism a property of the simulation's *specification*, not just an accident of environment.

- **Not Claimed:** The determinism guarantee is deliberately scoped. We **do not** extend it to things like real-time performance, cross-version consistency, or UI behavior [9] [10]. Any aspects outside the logic state transitions are beyond this semantic contract. This section will clearly state those boundaries (e.g., "deterministic" here does *not* mean the UI animation will play the same way, or that different versions of the engine produce identical results unless versioned properly).

- **Event-Bound Execution Model:**

- **Core Claim:** Introduce the system's execution model based on *discrete events* and **event boundaries** as the stepwise progression of state. The simulation is driven by an append-only event log; each user action or simulation tick is an atomic event. This yields a deterministic ordering of state transitions at event boundaries.
- **Supported by:** The system's event log format and replay mechanism serve as evidence. For example, RedByte OS logs events like `circuit_loaded`, `input_toggled`, `simulation_tick`, etc., in sequence [11] [12]. On replay, the engine applies these events in order, from the same initial state, to deterministically reach the final state [13]. We will describe the event types and how they capture *intent-level* interactions (not low-level micro-operations) – an approach that ensures repeatability. This section will reference how the *Replay Runner* processes events one-by-one in a fixed order, maintaining purity (same events → same result) [14].

- **Not Claimed:** We are not claiming to capture every possible internal micro-timing; the model intentionally works at the level of logical events. For instance, *user input timing* within the same event is not recorded (only the fact of the input toggle is). We also note that branching or interrupting event sequences (e.g. alternate timelines) is not supported – events define a single, linear history. The execution model is deterministic **given** a complete event log; we do not claim to handle speculative or concurrent events beyond the single-threaded sequence.

- **Verifiable Replay via State Hashing:**

- **Core Claim:** The system provides a mechanism to **verify** that a given execution can be replayed to an identical outcome, using cryptographic hashing of state. In other words, determinism isn't just a principle – it's mechanically checkable by comparing the hash of final states between a live run and a replay.
- **Supported by:** The Determinism Contract guarantees "determinism is cryptographically verifiable via state hashing" [15]. Specifically, the implementation computes a SHA-256 hash of the simulation state after live execution and after replay, and checks they match [16]. We will cite the formal property (∄ difference in hash implies deterministic equality [17]) and describe the hashing procedure (canonical JSON serialization of state + SHA-256 via Web Crypto). The *verifyReplay()* command implements this

check, running the log through the replay engine and confirming `hash(live) === hash(replay)` [18]. The use of a collision-resistant hash and canonical state ordering ensures reliability [19].

- **Not Claimed:** This is not providing *cryptographic security* or tamper-proof guarantees – only equality checking. We clarify that while hashing is used, the logs are not signed (integrity/authorship is out of scope) [20]. Also, verifiability applies to the final state equality; it doesn't catch issues if two different internal trajectories somehow result in the same final state (unlikely given the state is richly hashed, but theoretically). We do not claim any performance benefit to using hashing – it's purely for verification.

- **Inspectable Time Travel as a Query Primitive:**

- **Core Claim:** RedByte OS makes the entire execution history **inspectable**. It treats recorded history as queryable data: one can jump to any event boundary and examine the system state at that point, or step forward/backward through events. This time-travel debugging is deterministic and *side-effect free* (a read-only operation on the log).
- **Supported by:** Contract §3.3 ("Stable Time Travel") guarantees that stepping forward or backward yields the same results every time [21]. The system includes a *State Inspector* API (`getStateAtIndex`) to retrieve the state after N events, and navigation functions (`stepForward`, `stepBackward`) to move through the log [22] [23]. We will describe how snapshots are produced by replaying up to a point, and how invariants hold (e.g. stepping forward then backward returns to the original state, demonstrating *invertibility* of navigation [24]). The system also provides a diffing tool to compare state snapshots [25] [26], aiding inspection of what changed between events. All these are developer-facing query primitives (surfaced in a Dev Tools panel).

- **Not Claimed:** Time travel in this system is **read-only** – there is no ability to alter past state or create alternate timelines. The contract emphasizes that time travel is a query and does not mutate the log [27]. We also don't claim this is a fully featured IDE debugger: it's limited to the specific domain of logic circuit state, not arbitrary program state. There is no "time-travel editing" or reverse execution beyond stepping in the recorded log. This section will explicitly note these non-goals (no branching, no modification of history, no use as a general debugger).

- **Operational Validation in a Browser Environment:**

- **Core Claim:** The deterministic execution system is not only defined on paper – it has been *implemented and validated in real browsers*. This section details how the guarantees were tested end-to-end in a live environment, proving that the theoretical properties hold under practical conditions (e.g. different browsers, async APIs, real user interactions).
- **Supported by:** We cite evidence from Milestone D (Operational Validation) where the team ran the full record→replay→verify→time-travel cycle in multiple browsers (Chrome, Firefox, Safari) to confirm everything works as specified [28]. For example, using the dev panel, a user recording of a circuit simulation produced matching live/replay hashes on each browser [29]. The Web Crypto API hashing was confirmed to produce consistent results across environments [30] [31]. The system's automated test suite plus manual cross-browser tests give high confidence that determinism holds outside of a lab setting. We will highlight specific validation steps (like exporting a log, re-importing in a fresh session, and still getting a match [29]) as proof points.

- **Not Claimed:** We do not claim *universal* operational coverage – e.g., older or mobile browsers were not extensively tested [32]. The focus is on modern desktop browsers. Also, operational validation doesn't mean the feature is user-facing in production (it's available in a dev mode panel, not broadly deployed to end-users yet). We make no claims about performance in the wild; tests were about correctness. This section simply asserts that everything promised (within scope) has been observed to work in practice.

- **Comparison to Prior Work (Debuggers, Replays, Simulation Logs, Teaching Tools):**

- **Core Claim:** Position RedByte OS in context. While elements of this system resemble prior concepts (time-travel debugging, game replay logs, circuit simulators, educational tools), *none provided all these guarantees together as an integrated, verified system*. We outline key differences and similarities to illustrate the contribution.
- **Supported by:** We compare to **debuggers**: Traditional debuggers (and even "time-travel" debuggers like rr) often operate at the low level (instruction or source line) and require heavy instrumentation; RedByte's approach works at the semantic event level for domain-specific state, and it's *deterministic by design* rather than by recording nondeterministic effects. Moreover, RedByte's time travel is read-only and specialized, as explicitly noted (it's not a general-purpose debugger) [3]. Next, **game replay systems**: deterministic lock-step simulation is known in gaming (where replays are reproduced by feeding recorded inputs into the game engine). RedByte OS is analogous in concept (it records user inputs to a logic simulator), but goes further by formally hashing and verifying state equivalence. Also, unlike many game replays, it exposes a UI for stepping through the timeline and inspecting state, akin to a debugger. We will reference that our event log approach was inspired by those systems, but with additional emphasis on verification and introspection. For **simulation logs** in scientific or logic simulations: typically, logs capture outputs or traces but not full interactive sessions with guaranteed reproducibility. Our system ensures that an exported log *plus initial state* is a complete artifact for reproduction [33] [34], which is not common in simple logging tools. Finally, **teaching/education tools**: Many educational circuit simulators let users interact and maybe undo/redo, but RedByte OS uniquely provides a *proven-correct replay* and shareable execution logs. This means instructors or students can exchange a JSON log and be confident it will replay exactly the same (no manual re-setup needed). We'll note that this capability was not present in most prior teaching tools, framing it as an opportunity for more reproducible teaching demos.

- **Not Claimed:** We acknowledge that each of these prior domains has rich histories – we are not claiming to have "invented determinism" or "the first ever replay system". The contribution lies in **combining** these ideas (deterministic replay, hashing, time-travel UI) in a browser-based logic simulation context with formal guarantees. Also, we do not claim our system covers use cases of a full debugger (e.g., you cannot set an arbitrary breakpoint or modify state) nor the scale of a multiplayer game replay (our focus is single-user, single-machine, as the contract disclaims network sync [35]). Any performance or security advantages that some prior systems focus on are explicitly out of our scope. The comparison is careful to respect those boundaries – for example, we don't claim to replace version-control or notebook-style reproducibility in all scenarios, just to contribute a well-bounded solution in this space.

- **Limitations & Explicit Non-Goals:**

- **Core Claim:** Enumerate the known limitations and what the system deliberately does *not* do. This section serves as an "honest accounting" to avoid overclaiming. It reiterates that RedByte OS's determinism is limited to certain conditions and features.
- **Supported by:** We directly draw from the **"What Is Not Guaranteed"** list in the Determinism Contract [9] [10]. Key points include: no guarantees about real-time performance (the system ensures consistency of logical results, not timing), no promise of cross-version determinism (a change in the simulation engine could change outcomes, which is why the event log is versioned [36]), no cryptographic integrity (logs can be altered; we assume trust in the log source [20]), no determinism for the UI or rendering layer (only the simulation state is deterministic [37]), and no support for multi-user or network-consensus determinism (the scope is a single-browser, single-instance simulation [38]). We also mention that maliciously crafted event logs are not defended against – the system trusts the log inputs (non-goal: security hardening) [39]. Each of these will be briefly explained. For example: *"Determinism applies to logical state, not wall-clock time"* [9] clearly limits the claim to functional results, not performance.

- **Not Claimed:** Essentially this whole section is about what is not claimed. We ensure the reader understands, in each case, that if something is listed here (e.g. performance or cross-browser compatibility beyond what was tested), the authors explicitly *do not guarantee or focus on it*. We also highlight any trade-offs made: e.g., the decision to use the Web Crypto API means Node.js environments aren't directly supported in this implementation (browser-only by design) [40] – not a fundamental limitation of determinism, but a pragmatic scope choice. By listing non-goals, we reinforce that the paper is not selling a cure-all, but a specific, well-scoped contribution.

- **Implications for Education and Research:**

- **Core Claim:** Discuss why having deterministic, replayable interactive computation is valuable for *education* (e.g. in teaching digital logic or programming) and for *research* (e.g. experiments in HCI, reproducible computing research, programming language semantics). This is a more forward-looking discussion of how the system can be used now that it exists.
- **Supported by:** For education: We point out that instructors can record an interaction (like demonstrating a logic circuit in action) and share it with students, who can replay exactly the same sequence and even step through it themselves. The system's designers have already envisioned this use: one roadmap item is to graduate the dev-only record/replay panel into a user-facing feature for sharing "demos" in the logic playground [41]. Because the execution is deterministic, these shared replays are reliable teaching artifacts (no "it worked on my machine" issues). For research: a deterministic interactive platform allows controlled experiments. Computing education researchers could use it to gather exact replays of student interactions and analyze them. Programming languages and systems researchers might see this as a case study in bringing rigorous determinism to the frontend/web domain – it provides a concrete implementation to study or build upon. Moreover, the existence of a formal contract and tests makes it easier to reason about and extend the system for research purposes [42]. For example, research on debugging or program synthesis could leverage the stable replay of user actions as a dataset or baseline.

- **Not Claimed:** We avoid claiming any immediate educational or research outcomes that haven't been realized. These are implications and potential uses, not results. We don't, for instance, claim that using RedByte OS will improve student learning outcomes (that would require separate study). We also note that making it truly accessible to non-developers would require future UI work (since currently the feature is dev-only). In research terms, we stop short of claiming generality beyond our

domain – e.g., this approach might not directly apply to nondeterministic concurrent systems without similar constraints. This section stays speculative in tone, suggesting possibilities rather than promises.

- **Future Work (Speculative):**

- **Core Claim:** Outline the avenues for future enhancements or research building on this system, clearly marked as beyond the current implementation. This includes both system improvements and broader research questions.
- **Supported by:** We base future work on the project's own roadmap hints and natural extensions. For instance, **User-Facing Features**: transitioning the deterministic replay from a developer tool to a polished feature (so that end-users can save and share their sessions easily) [41]. **Security Enhancements**: adding cryptographic signing of event logs to ensure authenticity, if needed, since currently integrity isn't guaranteed (a known non-goal). **Scalability & Performance**: exploring optimizations for very large circuits or long event logs, given that performance was not a focus (perhaps adding indexing for faster time-travel queries, caching, etc.). **Multi-User Collaboration**: while explicitly out of scope now, one could research how to extend determinism to collaborative scenarios (this would likely involve a consensus algorithm or CRDT approach, clearly a complex future research topic). **Formalization**: providing a more formal semantics or even a proof of determinism correctness (the current evidence is tests and hashing; future work could use formal verification techniques to prove determinism from the contract's formal properties). In fact, a suggested next step is writing a research paper (the one we are outlining) to formalize event log semantics and relate this work to existing literature [43]. We'll cite the roadmap to show these ideas come from the team's vision: e.g., mentions of "collaboration" and "bug report logs" as potential features [44].
- **Not Claimed:** We clearly state that these are not implemented yet. No claim is made that any of these will definitely happen or that they're trivial. For example, we don't claim that multi-user deterministic sync is solved – only that now, with a strong single-user foundation, it could be attempted with caution. We also refrain from promising any specific timeline or success. This section may discuss challenges associated with each future idea, reinforcing that they require careful design (and likely, new contracts or versions as needed to maintain determinism guarantees). Each point is framed as a possibility opened up by the work done so far, not as a capability of the current system.

First-Pass Draft (Preliminary)

Introduction & Motivation

Interactive computer systems (such as simulation tools and educational environments) traditionally suffer from **nondeterministic behavior** – the same sequence of user actions can lead to different outcomes due to hidden state, timing, or environmental factors. This paper presents **RedByte OS**, a browser-based logic simulation environment that achieves *deterministic interactive computation*. In RedByte OS, if a user provides the same sequence of inputs on the same initial circuit, the system guarantees the same final result every time [1]. Our motivation for building this system is to make interactive simulations **reproducible and transparent**. Determinism in this context is a *semantic property*: it's about the logic of the simulation, not the low-level execution on hardware. By eliminating sources of nondeterminism and recording interactions

as formal events, we address long-standing pain points in debugging, teaching, and reasoning about interactive computations.

Why is this important? In conventional interactive apps, outcomes can be *opaque*. If a student or developer finds a surprising behavior in a circuit simulator, reproducing that exact scenario can be difficult – minor differences in timing or environment might lead to a different state. Traditional GUIs and simulators don't log user intent at a semantic level, and factors like GUI framerate or asynchronous callbacks can introduce variability. As a result, these systems are often "write-only": once you perform an interaction, you cannot reliably reconstruct what happened or why. Our goal is to invert that: to make each interactive session a **first-class artifact** that can be analyzed, verified, and shared.

To achieve this, RedByte OS was built from the ground up with determinism as an explicit goal. The system's design revolves around a formal **Determinism Contract**, which precisely defines what "deterministic" means in our context and – equally important – what it does not mean [2] [3]. This contract (summarized in Section "Determinism as a Semantic Property") prevents ambiguity and overclaiming. For example, we do *not* claim to make the entire browser deterministic – rendering and real-time user timing are outside our scope. Instead, we focus on the logic simulation results: given the same logical inputs, the outputs are the same, provably so. In practice, RedByte OS can record a user's session (the series of actions they took in the simulator) and later replay that session step-for-step, arriving at an identical final state, with the system automatically verifying that sameness via a cryptographic hash comparison.

By providing determinism in an interactive setting, we unlock new possibilities. Developers can debug by replaying exact scenarios, knowing that any divergence would indicate a real bug. Researchers and educators can share interactive "experiments" or tutorials, confident that others will see the same behavior. Overall, our work is driven by the belief that **interactive computing need not be an unpredictable black box** – it can be made as reliable and examinable as batch computations, without sacrificing the rich learning and debugging benefits of interactivity. The rest of this paper details how we realized this vision in a real system and what guarantees it provides.

Problem Statement: Non-Determinism and Opacity in Interactive Systems

Interactive systems – from web applications to simulations – are notoriously prone to nondeterminism. Unlike a pure function which gives the same output for a given input, an interactive session's outcome can depend on myriad extrinsic factors. Consider a student using a logic circuit simulator: the timing of their clicks, the scheduler of the browser's event loop, or even random number generation for certain components could alter the result. Traditional systems do little to mitigate this. **User interface events, rendering, and scheduling are generally not deterministic** [4]. Two runs "doing the same thing" are rarely truly identical in state by the end, because *when* and *how* things happen can differ in subtle ways.

This nondeterminism leads to **opacity**. If the student sees an unexpected output, they cannot easily replay the session to inspect where things diverged, because the platform didn't record the exact semantics of their actions, or because re-running might not follow the identical execution path. Debugging such scenarios often relies on luck and logging. Likewise, instructors or developers cannot *guarantee* that a demonstrated behavior can be reproduced elsewhere – a significant obstacle for teaching and for collaborative development.

Several specific issues underline the problem:

- **Hidden State & Unlogged Inputs:** Many interactive systems maintain hidden internal state that isn't fully exposed or logged. A simulation might have internal clocks or caches that influence behavior. Without a log of all *relevant* state changes, reproducing a run is guesswork.
- **Timing Variability:** In browsers, things like layout rendering, garbage collection, or user input timing (e.g., how fast a user toggles a switch) can vary. Traditional event handling might treat these differences as insignificant, yet they can lead to different sequences of operations. Since performance and animation are not typically part of a program's functional spec, developers often consider them nondeterministic by default [4].
- **Lack of Checkpointing:** Even if an interactive tool allows you to undo actions or reset, it usually doesn't let you jump into the middle of a past execution. There's no timeline you can scroll through. If a bug manifested around "step 5" of some process, you must manually try to recreate that scenario – an effort prone to human error.

In summary, interactive systems are "traditionally non-deterministic and opaque" because they weren't designed with replay or semantic logging in mind. The consequence is that we treat the behavior of such systems as ephemeral. This has been acceptable for casual use, but it's a serious limitation for **reproducibility** (in education, one cannot ensure students or graders see the same results) and **debugging** (developers can't rely on exact replays to diagnose issues). Our work tackles this problem by introducing determinism as a first-class goal in a browser-based simulation environment. We do so in a carefully *scoped* way: only the logical computation is deterministic, whereas inherently unpredictable aspects (like UI timing) are explicitly left out, to keep the problem tractable.

The problem, therefore, is not claiming that "all nondeterminism is bad" – rather, that in the context of a logic simulator (or similar tools), the lack of deterministic replay and introspection is hindering. By addressing this problem, we aim to make interactive computing more like running a pure function: predictable and repeatable, without sacrificing interactivity. The next sections describe how we define the semantics of determinism in RedByte OS and how we built a system to guarantee it.

Determinism as a Semantic Property

To make determinism meaningful in an interactive setting, we define it at the level of the **simulation's semantics**. In RedByte OS, determinism is not about low-level CPU instructions or exact timing of events, but about the logical *outcomes* given a specific history of interactions. We formalize this via the **unit of determinism**:

> **Given the same initial circuit state and the same ordered sequence of recorded events, the simulation produces the same resulting state** [5].

This definition, drawn from the RedByte Determinism Contract, captures the essence of our guarantee. It says that if you start a simulation in a known state and feed it the exact same sequence of user intents (e.g., "toggle this switch at time X, advance simulation by one tick, etc."), you will always end up with an identical state. Identical state is determined structurally (we will later use hashing to concretely check identity). All other aspects – how long it took, what was drawn on screen, which thread ran first – are deliberately abstracted away. They do not factor into the semantic model of the simulation's behavior.

It's important to clarify **scope and boundaries** of this determinism. According to the contract, RedByte OS guarantees determinism for the **logic simulation engine** and any execution captured in its event log [6]. The logic engine is the part of the system that evaluates circuit behavior (propagating signals through gates, etc.). The event log is the sequence of interactions and simulation steps (we'll detail that in the next section). **Replay and time-travel inspection** of those logs are also deterministic operations – meaning if you replay half the events, you get the exact state at that midpoint every time [6].

However, RedByte OS explicitly does *not* guarantee determinism for things outside that scope. For example, the UI layer (how the circuit is rendered in the browser, animations, widget states) is *not* deterministic by our definition [45]. Two runs might have slightly different animation timing or layout, and that's acceptable because it doesn't affect the circuit logic state. Similarly, performance and timing are not covered: determinism applies to the logical result, not to how quickly or slowly it was obtained [9]. The scheduling of browser tasks (which can be influenced by the runtime or OS) is outside our control – we ensure that even if tasks interleave differently, the *end state* remains the same, because our engine doesn't rely on nondeterministic ordering of operations. We also exclude any external integrations or network events from the determinism promise. If the simulation, say, fetched data from a server, we don't guarantee replaying that unless the fetch results were recorded or stubbed (our current implementation doesn't do network calls in the core logic engine at all).

One way to think about our determinism guarantee is as a **contractual promise** between the system and the user (or developer using it): *if you stay within these bounds, you get determinism; if you go outside, all bets are off*. This is why we call it a Determinism Contract and consider it binding [46]. The contract lists both guarantees and non-goals. For example, it notes that cross-version determinism is not promised [36] – if we upgrade the simulation engine in an incompatible way, a log from an old version might not produce the same state under the new engine. To handle that, the event log carries a version and the engine declares compatibility or not. Similarly, the contract does not promise anything about cryptographic security of the log (no built-in tamper prevention) [20], since our focus is on *functional determinism* (we trust the log's contents for replay).

Within the agreed scope, RedByte OS enforces determinism through several **semantic invariants** and design choices:

- **Deterministic State Representation:** We ensure that the representation of the circuit state is canonical. For instance, JavaScript's iteration over object properties or `Map` is not guaranteed to be in insertion order across engines, so we explicitly sort keys and map entries in a normalized form [47] [48]. By defining an unambiguous ordering for all components of the state, we ensure that two logically identical states *look identical* to the engine and to our hashing function. This removes a common source of nondeterminism.
- **Elimination of Hidden Sources of Entropy:** We avoid using any API that would introduce randomness or time sensitivity. Functions like `Math.random()` or uncontrolled use of `Date.now()` do not appear in the core logic engine. When the simulation needs a timestamp or a random seed, those are passed in as part of an event (so they're recorded and can be replayed) or otherwise handled in a deterministic fashion [49] [8]. For example, if a "simulation_tick" event needs to indicate a time delta, that delta is part of the event data, not computed from the system clock at replay time.
- **Pure Functions and No Ambient State:** The logic engine is designed to be as pure as possible. State evolves only through the application of events. There are no background threads modifying the

circuit state and no dependency on external global state. This means the outcome is a pure function of (initial_state, event_sequence). The contract formalizes this as a quasi-mathematical property: `hash(execute(initial_state, events))` is invariant [17].

By viewing determinism at the semantic level, we align our guarantees with what users actually care about – the correctness and reproducibility of the *simulation results*. We explicitly do not tie ourselves to lower-level determinism (e.g., making sure each CPU instruction happens the same way) because that's both infeasible in a browser and irrelevant to the simulation's logical behavior. Instead, we say: if the logic differs, we catch it. If the presentation or timing differs but logic is same, that's fine.

In closing this section, we reiterate: **Determinism in RedByte OS is a well-defined, testable semantic contract.** It's not a vague promise or a side-effect of writing single-threaded code; it's an intentional property that we specify and verify. The next sections will explain how events are used to structure execution, and how we record and replay those events to fulfill the contract.

Event-Bound Execution Model

RedByte's execution model is centered around **discrete events** that partition the interactive session into well-defined steps. This design is key to achieving determinism because it imposes an order and structure on what might otherwise be a continuous stream of stimuli. Each event represents an *intentional action or a logical tick* in the simulation, and the system's state only changes at these event boundaries. By recording the sequence of events, we capture a complete story of the session that can be exactly replayed.

What is an event in this context? We define a small set of event types that cover all interactions relevant to the logic simulation's state:

- `circuit_loaded` – occurs at the start of a session, recording the initial circuit configuration (the network of logic gates and their initial values).
- `input_toggled` – records a user action toggling an input (e.g., flipping a switch or changing a input pin from 0 to 1). It includes which node and which input port was toggled, and the new value [50] [51].
- `simulation_tick` – represents advancing the simulation by one step or a time delta (for circuits that have clocked behavior or propagate signals over time). It might include a tick count or delta time.
- `node_state_modified` – records a change to a node's internal state (if the simulation allows modifying component parameters or if a component's state is changed during simulation).

These events are deliberately chosen to be at the level of *user intent and simulation logic*. We do not, for example, have an event for "redraw the screen" or "garbage collector ran" because those don't impact the logical state. By keeping events high-level, we reduce the log size and keep the replay focused on meaningful state transitions.

The **Event Log** is simply an ordered list of such events, with each event stamped with a timestamp and any necessary data (like which node was toggled, etc.) [52]. The log is an append-only record; once an event is added (during recording), it is never altered. If the user performs actions A, B, C in that order, the log will contain [A, B, C] in that exact order. This seems obvious, but it's worth noting that some systems log things out of order or coalesce events; we do not – we keep the exact sequence.

Now, given this event log, the **Logic Engine** can perform a **Replay**. The replay process is deterministic by design: we start from the initial state and apply events one by one in sequence [13]. Concretely:

1. **Initialize state:** Take the `circuit_loaded` event's payload (the circuit definition) as the starting circuit state.
2. **Apply each event in order:** For each subsequent event:
3. If it's `input_toggled`, the engine sets the specified input node's value and updates the circuit state accordingly (propagating any immediate combinational logic that results).
4. If it's `simulation_tick`, the engine invokes the simulation's tick function (advancing the simulation by one time-step or performing one iteration of propagation, depending on the simulator design).
5. If it's `node_state_modified`, the engine applies the state change to the given node (for example, if a flip-flop's internal state was changed).
6. **Continue until all events are applied**, then output the final state of the circuit (and optionally the final state of the simulation engine, though for hashing we primarily focus on the circuit state).

Throughout this process, because the engine is pure and event application is deterministic, the final outcome is entirely determined by the input log. Replay is essentially a *pure function*: `replay(initial_circuit, event_log) -> final_circuit`. The contract explicitly notes this purity: the same log always produces the same result [14].

One might wonder, what about intermediate states? Because we apply events sequentially, we can talk about the state after each event as well. Those are the points at which our time-travel system can snapshot the state (described in the next section). Importantly, the states after each event in replay will exactly match the states that occurred during the original live session at the end of those corresponding events. This is by design: the simulation during recording essentially does the same state transitions, just in real time and possibly with UI updates in between. Since the UI doesn't feed back into the logic (no event is triggered by a pure UI occurrence without user intent), the recorded events encapsulate all the influences on state.

Example: Suppose a user loads a circuit, toggles input X, then toggles input Y. The event log might look like: `[circuit_loaded(circuit=…), input_toggled(node=X, value=1), input_toggled(node=Y, value=1)]`. If there are some combinational logic consequences (e.g., toggling X causes some gates to recompute outputs), those consequences are part of the *circuit state* but they don't generate separate events – they are a direct result of the input toggle and the engine's logic. On replay, when we apply `input_toggled(X,1)`, the engine will produce those same consequences internally. By the time we then apply `input_toggled(Y,1)`, the state of the circuit is exactly where it was in the live run before Y was toggled. Thus, after replaying all events, the final output matches the live run's final output by construction.

This **event-bound model** is crucial because it provides **clear checkpoints** in execution. Each event boundary is a point where the state is well-defined and can be hashed, compared, or inspected. It compartmentalizes the execution into chunks that can be reasoned about. Contrast this with a system where inputs could arrive continuously or concurrently – reasoning about determinism there is much harder, because you'd have to consider race conditions or partial orders. We avoid that by having a single, totally ordered log of events. Even if two inputs happened "at the same time" in the user's perception, one will be recorded before the other (perhaps just milliseconds apart, but order is preserved based on when the event loop handled them).

It's worth noting that our *Recorder* component is what captures these events during a live session. The recorder attaches to the simulation engine and UI controls. For example, when the user toggles a switch in the UI, the recorder's `recordInputToggled()` method is called with that node's ID, port, and new value [12]. The recorder timestamps it and pushes it into the log. The design ensures the recorder itself doesn't interfere with the simulation (recording is side-effect free [53]). Essentially, it just listens and logs.

In summary, by structuring execution into a series of logged events, RedByte OS creates a deterministic skeleton for the interactive session. The event log serves as the authoritative history that can drive a replay. No matter how many times we feed that log into the engine, or on which machine, we should get the same results, as long as the engine version is the same and complies with the contract. The next section will discuss how we *verify* that sameness via hashing (making this claim concrete), and subsequently, how we use these event boundaries for time travel.

Verifiable Replay via State Hashing

Logging and replay alone provide *determinism in principle*, but how do we know the replay truly mirrored the live execution? RedByte OS addresses this through a **cryptographic state hashing** mechanism that allows automatic verification of replay correctness. The idea is simple: if two execution runs (one live, one replayed) have identical final states, and we have a reliable way to represent those states, then we can check equality. We use SHA-256 hashes of the normalized state representation for this check.

The Determinism Contract explicitly states determinism is *provably checkable* in this way: *"A replayed execution can be proven identical to a live execution by comparing SHA-256 hashes"* [15]. Here's how it works in practice:

When a user finishes recording a session (say they stop the recorder after doing some interactions), we have an event log in memory. At that moment, the simulation has a certain *live state* (the state at the end of the user's interactions). We then invoke a **verifyReplay()** function, which does the following:

1. **Compute live hash:** Normalize the current live circuit state (ensuring things like order of keys, etc., are consistent) and compute a SHA-256 hash of that normalized state. This gives us a digest (a 256-bit number, usually represented as a hex string) representing the end state.
2. **Replay in parallel:** Using the recorded event log and the initial state (which we also have from the `circuit_loaded` event), run a fresh instance of the simulation engine through the exact sequence of events. This is like starting a brand-new copy of the simulator and feeding it the log. At the end of replay, compute a hash of the replayed final state in the same manner.
3. **Compare hashes:** Check if the two hashes are identical [16]. If they are, then by implication the states are identical (assuming negligible chance of SHA-256 collisions, which is a reasonable assumption given its collision resistance [54]).

If `hash_live == hash_replay`, we declare the run **verified deterministic**. In the RedByte OS dev UI panel, this is indicated by a "✓ Deterministic" status, along with showing the two hash values for transparency. The contract formalizes this as: `hash(live_execution) === hash(replayed_execution) ⇒ deterministic` [18].

There are a few subtleties under the hood worth mentioning:

- **Canonical State Representation:** We touched on state normalization earlier. This is vital for hashing. We convert the entire circuit state (all nodes, connections, and their values) into a canonical JSON structure such that two equivalent states produce the exact same JSON text. For instance, we sort arrays of nodes by a stable ID, we sort object keys alphabetically, etc. This eliminates false mismatches – e.g., JavaScript might list object properties in insertion order which could differ run-to-run; our normalization prevents that. The hashing process always uses this normalized form. The evidence of this approach appears in our Milestone A tests: identical circuits produce identical normalized state and thus identical hashes [55].
- **Async and Browser-Safe Hashing:** We use the Web Crypto API in the browser to perform SHA-256 hashing, which is an asynchronous promise-based API (for security and performance reasons, browsers don't expose direct synchronous hashing). So our `hashCircuitState()` function is asynchronous [56]. It collects the normalized state, calls `crypto.subtle.digest('SHA-256', data)`, and returns a hex string of the hash. This is important because it means the verify replay process also has to be async (we have to await the hash results). We ensured that making these operations async did not introduce nondeterminism – the promise resolution order is well-defined in our single-threaded context, and we don't intermix multiple concurrent hashing operations that could race. The hashing algorithm (SHA-256) is *deterministic* given the same input, obviously; we just had to ensure we feed it the same input in replay vs live.
- **Two-Path Execution:** In implementation, we actually run two code paths in verifyReplay: (a) the "live" path which is basically no-op (we already have the live state in memory from the running session, but we conceptualize it as one path), and (b) the "replay" path which instantiates a fresh engine and replays events [57]. Both end in a state that gets hashed. We did this to mimic how the events originally occurred – the live path is essentially re-running any finalization logic on the existing engine, whereas the replay path reconstructs from scratch. In an ideal scenario, both approaches yield the exact same state. If there were a bug (say some part of state wasn't logged and so didn't replay), then the hashes would differ, alerting us to a determinism breach.
- **Hash Comparison Outcome:** If the hashes differ, it means something went wrong – either a nondeterministic behavior crept in or there's a bug in the replay. In testing, we actively check that mismatched hashes are detected [58] [59]. In normal usage, a mismatch would indicate to the user/developer that the replay is not faithful, which is essentially a contract violation (this should not happen if our system works as intended). The UI would show a "✗" or some error in that case. During our development, such mismatches helped catch issues. For example, early on we discovered floating-point rounding differences could cause mismatches, which led us to avoid floating point in the core logic (only integers and booleans for logic signals).

Using hashing to verify replay has some attractive properties: - It's **unambiguous** and automated. The developer or user doesn't have to visually diff entire circuit states or trust their intuition; a matching SHA-256 means the states are bit-for-bit equal (given our consistent serialization). - It's **efficient** enough for our needs. Hashing even a complex circuit state (a JSON structure) is usually milliseconds of work, and doing it twice (once live, once replay) is fine for interactive use. We trade a tiny bit of performance to get a strong guarantee. - It's **cryptographically strong**. While our aim is not security per se, it doesn't hurt that we use a collision-resistant hash [54]. This means it's astronomically unlikely for two different states to produce the same hash. So if the hashes match, one can be confident the states really were the same (not just accidentally, but structurally).

One might ask: why not directly compare objects in memory? The reason is that the circuit state may contain complex structures (graphs of objects), and doing a deep comparison would be another approach – however, by hashing a normalized serialization, we get a single comparison point and also have the option to store or communicate that hash easily (e.g., in logs or UI). It also simplifies debugging: if someone reports a mismatch, they can share the event log and we can recompute and see where it diverged.

Another benefit is **offline verification**. Because we can export the initial circuit and event log as JSON (see Exportable Reproducibility below), someone else (or the same user at a later time) can run the replay and verify the hash matches some expected value. This could be used in a continuous integration test or a regression test: if a new version of the engine changes behavior, a previously recorded log might yield a different hash – alerting maintainers to a potential unintended change in semantics.

To sum up, verifiable replay is our answer to the question "how do I *know* it was deterministic?" We don't expect users to manually scrutinize outputs; we give them a one-click (or one-call) proof. Determinism becomes a tangible thing – a "✓" in the UI that says "yes, this run can be reproduced exactly." This bridges the gap between theory and practice: the contract's promise is enforced by code.

It's important to clarify again: this hashing-based verification is about *equality* of end states, not a cryptographic guarantee of authenticity. If someone maliciously edited the event log JSON to a different sequence, the system would faithfully replay that different sequence – the hash might match some expected value if they cleverly orchestrated it, or not, but we wouldn't know that the log was tampered with. In other words, we are verifying determinism, not preventing tampering. Authenticity or signatures on logs could be future work but are intentionally not part of the current system's guarantees [20].

In the context of related work, this approach is somewhat analogous to the use of checksums in networking (to verify data integrity) or the use of hashes in build systems (to verify reproducible builds). We're applying it to interactive sessions.

Having established that we can record and verify sessions, we now turn to the next powerful feature built on top of the event log: **time travel debugging** – the ability to inspect any step in the event sequence.

Inspectable Time Travel as a Query Primitive

One of the most novel aspects of RedByte OS is treating *time travel* in the execution as an **inspectable query** rather than a control flow gimmick. With the deterministic event log in hand, we can not only replay the whole session, but also examine the state at intermediate points. This is akin to a debugger's ability to set breakpoints, but here we don't need the foresight to set a breakpoint – we have the entire timeline recorded and can jump to any event boundary after the fact. Moreover, because of determinism, the state at each point is stable and repeatable.

How is time travel implemented? The core is the `getStateAtIndex(initialCircuit, eventLog, index)` function [60] [61]. This function returns a **CircuitStateSnapshot** representing the state of the circuit after the event at position `index` in the log has been applied. If `index = 0`, that would typically correspond to after the `circuit_loaded` event (initial state). If `index = N` (the last event), it yields the final state (which should match the live final state if the log came from a live run). Under the hood, `getStateAtIndex` simply replays the first `index` events from the log (starting from a fresh initial circuit

each time) and then stops, returning the resulting state. Thanks to determinism, it doesn't matter how many times or when we call `getStateAtIndex` – the result will be the same for the same `index`. The operation is **idempotent** and side-effect-free: it doesn't change anything about the original log or any global state [22]. It's a pure function mapping (initial state, partial event sequence) to a snapshot state.

Building on `getStateAtIndex`, we implemented **navigation primitives** for convenience: `stepForward(...)` and `stepBackward(...)` [62] [63]. These take a current snapshot and move one event forward or back, returning a new snapshot (or null if you're at the boundaries) [23]. Internally, `stepForward` just calls `getStateAtIndex` with index+1 (since it knows the current snapshot's index and total events), and `stepBackward` calls with index-1. We also provide `canStepForward()` and `canStepBackward()` as predicates so the UI can enable/disable buttons appropriately [64] [65]. The key property of these navigation functions is **stability**: if you step forward and then immediately step backward, you will return to the exact same snapshot you started with [24]. This might sound obvious, but it's only guaranteed because our replay of a prefix of events is deterministic. In other systems, stepping backwards often involves storing a previous state or undo logs explicitly; here we simply recompute by replaying, which is straightforward given our log and determinism.

Time travel is **read-only**. We emphasize that when you call `getStateAtIndex` or use the dev UI to inspect a past state, we are not rewinding the actual live simulation or altering history; we're querying a copy. The original recorded log remains intact and is always the source of truth. This approach aligns with the contract's note: "Time travel is a query operation, not a mutation" [27]. It ensures we don't accidentally diverge from the recorded timeline or create confusion about the current state. If the user, for instance, has a circuit playing and they go back to an earlier event to look at something, the system doesn't suddenly branch the simulation from that point. They are simply inspecting. They can always "resume" real time by returning to the final event.

We also provide a **state diffing** capability (`diffState(beforeSnapshot, afterSnapshot)`) which produces a structured description of what changed between two snapshots [25] [66]. This is useful to highlight the effect of an event or a series of events. For instance, if toggling input X caused three other signals to change and one new node to turn on, the diff would list those changes (e.g., "node A output changed from 0 to 1"). We categorize diffs by type: value changes, node added/removed, connection changes, etc. [26]. This is not a guarantee in the contract per se, but a convenience tool – it helps the user understand the *semantics* of the difference between states. It's important to note that our diff is structural and not semantic beyond that; it won't tell you "because you toggled X, Y changed" in English, but it will pinpoint *what* changed. This has proven useful in debugging: you can step through events and see exactly what each step did.

From an interface perspective, we built a **Determinism Dev Panel** in the browser UI that exposes these controls (though only in a development mode) [67] [68]. The panel has buttons for "Start Recording" / "Stop Recording", "Verify Replay", and once a replay is verified, it allows "Initialize Time Travel". Upon initialization, the panel is basically using `getStateAtIndex` for index 0 to grab the initial snapshot and preparing for navigation. Then "Step Forward"/"Step Back" buttons call the respective functions and update the displayed state. The panel displays the current event index out of total (e.g., "Event 5 of 10") [69] [70]. It also shows the hash comparison results after verification (so the user can see the live vs replay hash if they want). The design is minimal – it's not meant for end-users in production, but for developers to validate and explore the determinism features. We took care to ensure the panel itself doesn't introduce nondeterministic

behavior. It doesn't inject anything into the simulation; it just calls the deterministic APIs and renders results.

Now, **what does the contract guarantee about time travel?** It guarantees *stability* and *repeatability* of stepping through time [21]. This has been tested by scenarios such as stepping forward 10 steps and then backward 10 steps ends up where you started [71]. The contract's wording "any recorded execution can be inspected at any event boundary" is fulfilled by `getStateAtIndex` and the navigation functions. It also states that re-initializing time travel (i.e., if you start over and inspect again) yields the same snapshots for the same indices [72], which is inherently true because we always derive from the log.

One limitation to acknowledge: because we rely on replay to generate past states on demand, if the event log is very long or the circuit is very large, jumping to a late event means replaying a lot from scratch, which could be slow. We have not yet implemented any optimization like checkpointing states to speed this up. In our tested scenarios (tens or hundreds of events, moderate circuit size), the performance has been fine, if not instant. We explicitly chose correctness over optimization for this phase. So while time travel is functionally available, it's not tuned for huge data sets (something noted in our limitations).

It's worth comparing our approach to **traditional debuggers**: Traditional debugging often requires you to decide ahead of time where to break, or to use reverse debuggers that log execution at a very low level (which can be extremely slow or generate massive logs). Here, because our domain is constrained (logic circuits and a known set of events), we get the benefits of a reverse debugger *with relatively low overhead*. We don't log every CPU instruction; we log high-level events that are infrequent relative to, say, instruction count. And we leverage our deterministic replay to regenerate intermediate states rather than storing them all. In essence, we're trading some recomputation for not storing every state (space-for-time tradeoff), which works well given that the recomputation is deterministic and relatively fast in our domain.

Finally, beyond debugging, we see **pedagogical value** in time travel. In a classroom or learning context, a student can step back and forth through a circuit's behavior to understand cause and effect. The diffing can help highlight exactly what changed, reinforcing learning (e.g., "Toggling this input flips these two outputs"). The guarantee that this can be done any number of times (idempotently) without altering the circuit is crucial – students can experiment in the time-travel mode without fear of messing up their live session, then resume live simulation if needed by exiting the time-travel view.

In summary, inspectable time travel transforms the recorded log from a static playback into an interactive timeline that the user or developer can query. This capability rests entirely on determinism: only when the execution is deterministic can we trust that the snapshots mean something (and aren't artifacts of a particular run). RedByte OS thus offers a lightweight but powerful form of time-travel debugging tailored to the domain of logic simulations.

Operational Validation in a Browser Environment

All the guarantees and features described so far were validated in an actual browser environment to ensure that our deterministic model holds up outside of theoretical design. RedByte OS is implemented in a web tech stack (TypeScript, running in browsers), which introduces real-world considerations: the JavaScript runtime, the Web Crypto API, different browser behaviors, and user interaction patterns. This section describes how we **validated the system end-to-end** and what constraints that imposes.

Cross-Browser Testing: We tested RedByte OS in multiple modern browsers (Chrome, Firefox, Safari) to ensure that our determinism guarantees are not accidentally broken by browser-specific quirks. For example, we needed to verify that the Web Crypto API's SHA-256 implementation yields consistent results across browsers (it should, by specification, but it's good to confirm) and that there were no differences in JavaScript's sorting or JSON stringification that could slip through. Our Milestone D included a checklist for browser compatibility: open the identical circuit and run the same recorded interactions in Chrome and Firefox, and confirm both produce the same final hash [73]. We found that as long as we used the normalized state and our defined logic, the browsers behaved uniformly – which is expected, since we're not relying on any undefined language behavior.

One particular adaptation we had to make was switching from Node.js's `crypto` module (which we used in early tests) to the Web Crypto API in the browser [74]. The Node crypto allowed synchronous hashing, whereas Web Crypto is async and slightly different in usage. This change was part of making the system truly browser-native, and our validation included ensuring that this did not introduce any race conditions. We wrote tests to confirm that calling our `sha256()` function twice on the same input yields the same result both times (which it did) [75] [76], and that no errors occur when using it in the browser context (no missing polyfills, etc.). This was critical because a hashing mismatch could break everything. The result: **browser-native hashing works correctly** and deterministically in all tested browsers [31].

End-to-End Workflow Testing: We conducted full scenario tests akin to user behavior to validate that every piece works in concert. For instance, one manual test procedure was: 1. Open the RedByte Logic Playground in the browser. 2. Create or load a known circuit (we often used a simple 2-to-1 multiplexer as a test case). 3. Start a recording (via the dev panel). 4. Perform a series of interactions: flip some inputs, maybe toggle through a few simulation ticks. 5. Stop the recording. 6. Click "Verify Replay" in the dev panel. 7. Check that the panel shows `liveHash === replayHash` and a green check mark (✓ Deterministic). 8. Click "Export Log" to download the session as a JSON file. 9. Refresh the page, import that JSON (we have a way to load an exported session). 10. Run verify on the imported session again to make sure it still passes (and ideally, even re-play some interactions, although imported logs are static).

We got a successful outcome in these tests: the verification showed identical hashes, and the exported log when re-imported also verified identically [29]. This is strong evidence that determinism is preserved not just in memory but also through the serialization/deserialization process (which is expected because the export is just the JSON form of the event log and circuit).

Additionally, we tested the **time-travel navigation** in a live browser context to ensure the UI controls correctly call the underlying functions and that, for example, stepping forward then backward in the panel indeed returns to the original state with no errors. An interesting edge case we validated was behavior at the boundaries of the event list: if you try to step forward at the end, the system should indicate no more steps (we disable the button via `canStepForward = false`), and similarly at the beginning for stepping back. We confirmed that at the final event, "Step Forward" returned null/no action and didn't break anything, and at the first event "Step Back" did nothing (with no state change). The panel correctly kept the user from going out of bounds [77].

Performance considerations: While performance is not a guarantee, we did take note during operational tests of how responsive the system is. For the moderate sizes of circuits and logs we tried (like under 100 events, circuits with under ~50 components), everything was near-instantaneous – recording overhead was negligible, verification hashing took a fraction of a second, and time-travel stepping was quick enough to

feel interactive (usually less than 0.1s to compute a new state). This gave us confidence that the system can handle typical usage in a browser without special optimization. We didn't formally push the limits in these tests; that lies in future performance evaluations. However, we explicitly recognize that performance under extreme conditions (very large circuits or thousands of events) is not yet validated [78] – so if someone tried that and found it slow, it wouldn't contradict our contract (since we don't promise performance).

Locking Down Determinism: Once we saw all tests passing in the browser, we effectively "locked" the determinism features to prevent accidental regressions. In practice, this means: - The event log format (version 1) is now fixed. We freeze the schema of events so that any change would require a conscious version bump [79] [80]. This ensures that old logs remain compatible or we know when they won't be. - The Determinism Contract was marked as binding and any future changes must go through the contract versioning process [81] [82]. In our repository, we might even set up tests or CI checks that flag modifications to critical determinism code that aren't accompanied by contract updates. - The public APIs associated with determinism (like `createRecorder()`, `verifyReplay()`, `getStateAtIndex()`, etc.) are declared stable [83] [84]. We intend not to change their function signatures or semantics now that they are proven to work. This is analogous to an API freeze for this subsystem so that external tools or future components can rely on them.

The **operational validation** milestone effectively signaled that "this system works in reality as advertised." It's one thing to write tests that simulate things, but another to have it run in a real user scenario. For example, one real-world test we did: use the system during an actual interactive session of moderate complexity (like building a small logic circuit, running it, etc.) and ensure that enabling the determinism recording didn't break the user experience. The dev panel indicated recording was on (we show a red dot), and the user didn't perceive any differences in simulation speed or behavior. After stopping, the verification was instant and positive. This means our instrumentation (the recorder hooks, etc.) were lightweight enough not to interfere with usage.

Limitations observed: We did identify some limitations during operational testing that are worth noting: - We didn't test on mobile browsers or older browsers (e.g., IE11, older Safari) [32]. Mobile has additional constraints (performance, touch events instead of mouse events, etc.) which we haven't explored. So, determinism might hold on mobile in theory, but we can't vouch for it without testing. This is outside our current scope. - Extremely long sessions or very complex circuits haven't been tested in production scenarios. So, while the system should still be deterministic, the practicality (e.g., memory usage of the event log, time to replay) is unknown at those scales. - The UI we built for determinism (the dev panel) is a developer tool – we didn't focus on making it user-friendly for laypersons. Operational validation treated it as a means to an end (proving the concept works). For a polished user-facing feature, more design work would be needed (e.g., better visuals, possibly an ability to scrub through the timeline instead of step buttons, etc., and handling of error cases or log management). - We trust the environment the code runs in. If the browser is running on faulty hardware or someone has tampered with the JavaScript runtime, obviously determinism could break (but that's beyond any reasonable scope).

In conclusion, the operational validation confirms that **RedByte OS's deterministic execution model is not just a theoretical contract, but an actual property of a deployed system**. By running the system in real browsers with real user actions and obtaining consistent results (backed by hash comparisons), we demonstrated that all parts of the contract hold. This gives us confidence to proceed with using this system in broader contexts, and it provides a baseline for others to trust the system's behavior.

The process of operational validation also effectively turned our determinism features "on" for day-to-day use in development: the team can use the dev panel to catch regressions whenever new features are added to the simulator. It has become a tool in its own right – for example, if someone modifies the logic engine, we can run the determinism verification to ensure they didn't inadvertently introduce a nondeterministic change. In this way, the system is self-testing to a degree.

With the core functionality validated, we can now compare our approach with related efforts in other systems, to clarify how our contribution stands in context.

Comparison to Prior Work

Deterministic replay and time-travel debugging are not entirely new concepts; our work connects to ideas from debuggers, distributed systems, and educational tools. However, the unique combination of features and the specific context of browser-based logic simulation set RedByte OS apart. In this section, we compare and contrast to illustrate the novelty and limitations of our approach relative to prior art.

Versus Traditional Debuggers: Classic debugging tools (e.g., GDB for low-level code, or browser dev tools for JavaScript) allow stepping through code, setting breakpoints, and inspecting state. Some advanced debuggers and research prototypes offer *time-travel debugging* (also known as reverse debugging), which lets you step backwards. One example is Mozilla's `rr` tool, which records program execution at the instruction level so it can deterministically replay it for debugging. Compared to such tools, RedByte OS operates at a higher level of abstraction – we record logical *events* (user inputs, simulation steps) rather than every instruction or memory access. This makes our logs much smaller and more comprehensible, at the cost of not handling arbitrary code (we handle a specific simulation engine). We also integrate the record/replay directly into the application's logic, whereas `rr` and similar are external layers that intercept execution. This means RedByte's approach is more domain-specific but easier to use for that domain (no special setup or overhead, just click "Record").

Another key difference: most debuggers, when not in reverse mode, are not deterministic – if you run the program again, it might behave differently unless you controlled all inputs. RedByte ensures that the program (the simulation) *is deterministic by design*, which is a stronger condition than what typical debuggers assume. Also, our time-travel is read-only (you can't change state in the past and continue from there) whereas some debugging scenarios consider "what-if" modifications. We intentionally don't support that because it would break the connection to the recorded reality and complicate the determinism guarantee (we'd be exploring alternate timelines, which is a different problem space).

Versus Distributed System Replays / Game Replays: In distributed systems and online games, **deterministic replay** has been used to replicate bugs or synchronize state. For instance, some distributed databases can replay logs of transactions deterministically, and some online multiplayer games use lockstep simulation with input logs to keep all players in sync (each client runs the same simulation with the same sequence of inputs, achieving the same outcomes). RedByte's approach is spiritually similar to a game replay or a deterministic simulation in distributed computing. The notion of recording inputs and having a simulation produce the same outputs is exactly what games like StarCraft do for replay files (they save initial state and player commands and the game is deterministic enough that you can replay the whole match). Our contribution relative to that prior art is adding a layer of **formal verification and inspection** on top. Game replays typically assume determinism but don't prove it beyond perhaps CRC checks or just trust in the engine. We actively hash and compare to prove equivalence [15]. Additionally, game replay systems

usually don't allow arbitrary stepping and inspection by the user – you watch a replay like a video (albeit rendered by the engine). RedByte allows you to pause and inspect and even see a structural diff of the state, which games rarely expose (they might show scores or a timeline, but not internal state of all components).

However, our system currently is single-user. We are not dealing with the complexity of synchronizing multiple concurrent users in real-time. If one were to extend our approach to a multi-user collaborative simulation, it would resemble the multiplayer game determinism problem, which often requires strict network protocols and rollback mechanisms to handle lag – those are beyond our scope. We note that multi-user collaboration is considered a separate layer, and determinism would help (each user could share a log), but we haven't built a system for live synchronization. In fact, the contract explicitly marks "network-synchronized determinism" as out-of-scope [85].

Versus Simulation Logs in Scientific Computing: In fields like model checking, hardware simulation, or scientific computing, logging simulation data is common. For example, hardware description language (HDL) simulators produce waveform logs that show how signals change over time. Those are deterministic simulations by design (given the same input stimuli, the circuit behaves the same, similar to our core logic engine). The difference with RedByte OS is the **interactivity** and *intent-level logging*. Traditional simulation logs often log every little change (like every signal at every tick), which can be overwhelming. Also, those tools might not capture interactive "user did this" events because usually the stimulus is pre-scripted. We sort of hybridize these approaches: we capture the user's actions as the "stimulus log" for the simulation. In doing so, we don't log every signal change (we don't need to, since the engine can recompute them), which keeps our event log compact and focused.

Another angle is **education tools** (for example, block-based programming environments, or algorithm visualization tools). Many of those let the user step forward and backward through *pre-defined* examples or through their own actions via undo. Few, however, record a full session with the guarantee of replay. For instance, some coding education platforms have a "history" slider, but it might be limited to code edits, not program execution. One somewhat related concept is the idea of **notebook** environments (like Jupyter notebooks) ensuring reproducibility: there, if you run the same code with the same inputs, you get the same result. But notebooks don't typically log user actions beyond the code cells executed.

Unique aspects of RedByte OS in context: The combination of features we have – deterministic engine, event log, cryptographic verification, and interactive time travel – doesn't cleanly appear in prior single systems to our knowledge. Time-travel debugging systems like Elm's "time-traveling debugger" or Redux DevTools in the web development world do have similarities: Redux (a state management library for web apps) can log all dispatched actions and let developers jump back and forth in state. In fact, conceptually, RedByte OS's approach is akin to treating the circuit simulation like a pure reducer function and events like Redux actions. The Redux DevTools allow stepping through state changes, which is very similar to what we do. The main differences: we formalized the guarantees (Redux assumes your reducers are pure for time-travel to work but doesn't enforce it beyond developer discipline; we enforce purity through design and test every step) and we add the formal verification via hashing (Redux DevTools doesn't check a hash, it just assumes if you replay the same actions you get the same state – which can fail if you had any impure reducers). Also, Redux logs aren't usually exported/imported for cross-environment replay, whereas we explicitly allow exporting a session to JSON and re-running it elsewhere with assurance [33] [86]. In a way, RedByte OS could be seen as a domain-specific application of the principles behind Redux (deterministic state updates via actions) combined with the rigor of verified replay.

By comparing to these various domains, we illustrate that RedByte OS is **standing on the shoulders of prior techniques** but integrating them in a novel configuration: - Like *debuggers*, we allow state inspection and stepping, but we rely on a pre-recorded log and focus on domain semantics. - Like *replay systems*, we capture inputs and can reproduce outcomes, but we add strong verification and introspection. - Like *educational simulators*, we aim to help users understand system behavior, but we give them an exact, shareable record rather than just an on-screen animation.

It's also worth noting what we do *not* do, compared to some prior systems: - We are not providing a general solution for arbitrary code determinism (tools like `rr` or chess engine analyzers tackle that in their realms, often with performance overhead). - We are not a replacement for version control or literate programming (our logs are execution traces, not human-written narratives or editable code sequences). - We haven't integrated any AI or automated analysis to diagnose differences (some research debuggers try to explain causality; we provide raw diffs and let the user interpret them).

In summary, RedByte OS contributes an example of **deterministic, replayable interactive computation in the browser**, with a careful balance of techniques from various fields. This comparison underscores that while the ingredients (logs, replay, hash, time-travel) have been seen before, the recipe here – applied to logic circuit simulation, running live in a browser with formal guarantees – is, as far as we know, unique. We view it as a step toward bridging gaps: bringing the rigor of reproducible computation (more common in batch or backend systems) into the interactive and educational sphere.

Limitations and Non-Goals

No system is without limitations, and we have been careful from the outset to document what RedByte OS does **not** do. Being explicit about these non-goals is important to maintain a clear understanding of the system's guarantees and to avoid misapplication or overhyping of the results. Here we summarize the key limitations:

1. Performance and Real-Time Behavior: RedByte OS makes **no guarantees about real-time performance** [9]. The simulation's determinism is about logical correctness, not speed. This means, for example, if you replay a session, it might run much faster or slower than the original (since on replay we are not waiting for user input timing). We do not attempt to record or enforce any wall-clock timing. Frame rates, UI responsiveness, and latency are outside the scope. If someone needed a real-time replay (say to recreate exact timing of events), our system wouldn't ensure that – it would only ensure the sequence and outcomes are the same. This was a conscious non-goal: we prioritized reproducibility over real-time fidelity.

2. Cross-Version Determinism: The determinism guarantee holds **only within the same version of the simulation engine and event log format** [36]. If there's an update to the logic engine that intentionally (or unintentionally) changes how circuits behave, a log recorded on the old version might produce a different result on the new version. We mitigate this by versioning the event log and tying the contract to specific engine versions. But beyond that, we don't promise that, say, RedByte OS v1.0 and v2.0 will produce identical results for the same log – in fact, if v2.0 had a different feature, it likely wouldn't. To maintain determinism across versions requires either strict backward compatibility or a mechanism to handle legacy behavior. We've decided such cross-version support is a non-goal for now (except to detect the version and refuse or convert logs). Practically, this means that reproducibility is assured in a given snapshot of the system, but long-term archival of logs would need either freezing the engine version or eventually writing conversion tools.

3. Security and Integrity: Our system is **not a security or anti-tamper system**. The event logs are not cryptographically signed or protected; they are simply data. If someone maliciously modified a log file (or if a bug caused a bad log entry), the system would still replay it deterministically, but you might get a garbage-in, garbage-out situation. We do use hashing to verify equality, but that's different from ensuring the log wasn't altered. For instance, an attacker could record a session, then alter the JSON to do something else – on replay it will deterministically do that something else, and the hash will faithfully reflect that different outcome. We trust that the logs are *trusted inputs* [39]. If you need guarantee that a log came from a particular user or wasn't corrupted, you'd have to add a layer of digital signatures or checksums outside our current system. We explicitly listed "no cryptographic integrity or tamper detection" as a non-goal [20]. Also, we do minimal validation on event logs upon import. We assume they are well-formed and make sense. A sufficiently malformed log could conceivably cause the replay engine to throw an error or produce undefined behavior – we mitigate obvious issues with schema checks (e.g., ensure required fields exist), but we haven't focused on hardening against malicious inputs.

4. Scope of Determinism – UI and External Factors: As mentioned earlier, we do not cover UI behavior in our determinism contract [37]. That means the user interface – e.g., how the circuit diagram is drawn, or any animations or cursor blink rates – can vary and we don't record or replay that. The logic will run the same, but the presentation could be different. For example, if during recording the user sees an LED blink animation, we don't guarantee that on replay they'll watch the same blink timing (though the final on/off state of the LED at event boundaries will be the same). Similarly, any external interactions like network calls or file I/O are not recorded. Currently, the logic engine doesn't do any network calls by design. If we integrate something like that in the future, we'd have to consider it separately, but as of now, external integrations are not part of the deterministic core.

5. Single-Engine, Single-Thread Assumption: Our determinism is guaranteed only for the **reference logic engine implementation** we wrote, running in a single thread [87]. If someone wrote a different engine implementation (say a reimplementation in WebAssembly or a server-side version) and tried to replay logs there, there's no guarantee it would match byte-for-byte. They'd have to ensure that their engine is also deterministic in exactly the same way. We don't currently provide a formal spec beyond the contract for how the engine works, so cross-implementation determinism is a non-goal [87]. In practice, this hasn't been an issue since we have one engine, but it's worth noting as a limitation should the project expand (for instance, if a hardware-accelerated version of the logic engine was developed, one would need to verify it produces identical results).

6. No Multi-User Collaboration or Distributed Consistency: RedByte OS is inherently a single-user, local application (even if it's in a browser). We do not attempt to solve issues of multiple users editing a circuit simultaneously or sharing a live session. Collaboration features like real-time shared editing or networked simulation are explicitly called out as separate from our replay guarantees [85]. If in the future two users each have their own log of a session and then they try to merge them, that's beyond anything we handle. However, one could use our export feature to exchange logs (one user can send a log to another who can replay it). But real-time sync would require determinism across potentially divergent event streams and network delays, which is a known hard problem (similar to multiplayer game sync or Google Wave/ Operational Transform). We decided early that this is not in scope.

7. No State Editing or Alternative Timeline Exploration: Because time travel is read-only, we do not let users create a new timeline by altering a past state and continuing from there (like some advanced debuggers or sandbox games allow). This is a limitation in terms of functionality (it could be a cool feature),

but a deliberate choice to keep the model simple and deterministic. The moment you allow branching, you'd have to record new events that diverge from the original log, which becomes a different log essentially. We think it's cleaner to treat logs as immutable records; if you want a different scenario, you record a new session.

8. Node.js/Server-Side Incompatibility (Current Implementation): Our hashing relies on Web Crypto API, which is available in browsers. If someone tried to run our determinism code in a Node.js environment, Web Crypto might not be available or would need a polyfill. We flagged that as a known limitation [40]. It's not a fundamental one – we could use Node's crypto in that context – but currently the focus was the browser. So determinism is assured in browser usage; using the same code on a server might require slight modifications (e.g., using `crypto.createHash` instead of Web Crypto, and ensuring the same normalization logic works in Node, which it should). We mention this just to set expectations for anyone who might think to reuse our code outside the browser.

To ensure these limitations are well-understood, our documentation (the Determinism Contract and milestone docs) have dedicated sections listing them, often under "What is not guaranteed" or "Known Limitations". Throughout this paper, we have tried to respect those boundaries. For instance, we never claimed that using RedByte OS will make your simulations run faster, or that it can detect tampering, or that it supports collaborative editing.

Why consciously not address these things? Partly to keep the project focused and feasible. Each of those non-goals is either an orthogonal problem (like security or real-time performance) or something that would multiply the complexity of the project significantly (like multi-user determinism). By narrowing scope, we were able to actually build a working system that excels at its specific purpose. In future work, some limitations might be revisited (for example, adding optional log signing for authenticity if someone wanted to use this for serious assessment scenarios). But acknowledging them now is crucial for an honest evaluation of the system.

In conclusion, RedByte OS should be used with the understanding that it guarantees *deterministic logical outcomes* within one browser session and one version of the software. It does not solve problems beyond that scope. We believe being clear about these limits not only builds trust (we're not hand-waving about potential issues) but also guides appropriate applications of the system. In the next section, we discuss how, within its scope, the system can be leveraged in educational and research contexts, and we then outline where we might go from here in future work.

Implications for Education and Research

A deterministic, replayable interactive system like RedByte OS has a number of potential implications and uses for education and research, especially in fields of computer science education, software engineering, and programming languages.

For Computing Education (especially digital logic and programming): One of the challenges in teaching is demonstrating concepts in a way students can reliably reproduce and experiment with. Traditionally, an instructor might do a live demo of a circuit or program in class; students watch and try to follow along. But if a student misses a step or wants to experiment with a slight change, they often have to recreate the setup themselves from scratch, which can lead to confusion if they don't get it exactly right. With RedByte OS's

approach, an instructor could record a demonstration of a circuit in action – for example, showing how a flip-flop stores a bit or how a certain combinational logic produces a truth table. This recorded session (the event log plus circuit) can be exported as a **reproducible artifact** [33] and shared with students. Students can then load this artifact and replay the demo exactly, or step through it at their own pace using the time-travel controls. They can inspect the state after each step, which is like having a guided tutorial but under the student's control. Because the replay is deterministic, every student sees the same sequence of states, so they can collectively discuss "At step 5, why did this output change?" knowing everyone's step 5 is identical. This could enhance understanding and reduce the chance of misinterpretation that often comes when each student's environment is slightly different.

Furthermore, students can use the recording feature themselves as a form of **learning by teaching**: they could be assigned to create a deterministic recording that illustrates a concept, which forces them to think systematically about the sequence of steps and outcomes. The resulting logs can then be peer-reviewed or graded, providing a clear trace of the student's thinking process.

For educators evaluating students, determinism provides a sort of *ground truth* to evaluate against. For instance, in a lab exam setting, a student might be asked to design a circuit that does X and to provide a log of a test sequence showing it working. The instructor can replay the log to verify the circuit indeed behaved as expected. If the log is deterministic, any discrepancy in behavior is either a mistake in the circuit or an intentional aspect – there's no ambiguity of "maybe it was a glitch". It's also easier to reproduce bugs: if a student says "sometimes my circuit fails when I do this", they can provide a log of it failing, and the instructor can see it exactly.

For Research in Reproducibility and PL Systems: RedByte OS can be seen as a case study or tool in the broader discussion of reproducibility in computing. It touches on ideas common in research on determinacy in parallel/concurrent programs, deterministic simulation frameworks, and record/replay debugging. Researchers in programming languages (PL) might be interested in the formal contract approach we took – we specified a property (determinism) and enforced it through both engineering and lightweight formal methods (hashing, tests). It provides a concrete example of treating execution traces as *semantic units* that can be reasoned about.

One could imagine building on RedByte OS to explore **formal verification**: for example, proving that the hashing mechanism and event application actually enforce a refinement of an abstract deterministic specification. Or exploring what it would take to extend the determinism contract to more complex language features (if, say, the logic engine became Turing-complete or allowed user scripting – would the determinism still hold or what constraints would we need?).

For HCI (human-computer interaction) research, having a log of exactly what the user did is gold mine data. It removes uncertainty when analyzing user behavior, because you have a definitive record of actions and system reactions. If one were studying how users learn digital logic, for example, one could instrument RedByte OS to collect all student logs and then analyze them to see common patterns or mistakes. Because the logs are deterministic, any analysis script run on them is itself reproducible – another researcher could take the same logs and get the same results. This could help elevate the study of interactive learning tools from anecdotal to data-driven.

Collaboration and Communication: Deterministic logs could serve as a medium of communication between developers or between users. For instance, a user finds a bug in the logic simulator. Instead of

describing it in words ("when I connect this gate to that and click here, it sometimes goes wrong"), they can record a session that demonstrates the bug, and send the log to the developers. The developers replay it and see the bug exactly as it happened [88]. This dramatically reduces the "I can't reproduce this bug" problem that plagues software development. It's similar in spirit to how a failing test case pinpoints an issue – here the failing test is literally a captured user session. Our export format is JSON, which is language-agnostic and can be attached to issue trackers or emails easily.

For general software engineering research, this raises interesting possibilities: Could other systems incorporate this approach to improve bug reporting and debugging? We focused on logic simulation, but one might envision IDEs or other interactive tools adopting a form of event log recording for later analysis. RedByte OS demonstrates that it's feasible and beneficial at least in one domain.

Teaching Determinism and State: There's also a reflexive educational benefit: using the system teaches students (implicitly) about concepts like determinism, state, and hashing. For example, a curious student might ask, "What are those hex numbers (hashes) shown when I verify a replay?" – leading to an explanation of hashing and why it's used to verify equality. Or they might wonder why we need to press "Initialize" before stepping through time – leading to a discussion on how initial state and event logs define a computation. In essence, the tool itself can illustrate important CS ideas: that a program's behavior can be understood as a pure function from inputs to outputs (here, from initial state + events to final state), and that when it's not, weird things happen. If they ever see a determinism verification fail (maybe if they tinker with the code or there's a bug), that's a learning moment about nondeterminism.

Limitations of these implications: To be balanced, using RedByte OS for education assumes students and teachers have access to the tool (it's in a browser, which is good, but it's not a widely known platform yet). Also, the dev panel is currently a dev tool; an instructor might not want to require students to use a dev mode interface. One future improvement (discussed next) is making a more user-friendly version for exactly these educational uses.

From a research perspective, one must also note that our determinism is within a sandboxed environment (browser, single-thread). So while it's an existence proof of a deterministic interactive system, it doesn't solve determinism in general concurrent programming or distributed systems. Researchers in those fields might take inspiration but would still have to address their own complexities (like synchronization issues, which we avoid by design). But at least RedByte OS can be a testbed to try out ideas in a simpler context.

In summary, the advent of a system like RedByte OS could **improve the rigor and quality of interactive computing activities** in both educational and research settings. It enables *exact replay and inspection* where previously one had to rely on approximate methods or substantial manual effort. We hope it encourages more tools to be built with determinism in mind, especially when reproducibility is valuable.

Next, we will outline possible future work to extend this system, many of which directly address making it more accessible for education or exploring research questions raised by this approach.

Future Work and Speculations

While RedByte OS's deterministic execution framework is implemented and stable (as of this writing), there are several avenues for future work. We separate these clearly from the current guarantees – everything here is **speculative or aspirational**, not part of the present system's promises.

1. Formal Semantics and Verification: One direction is to take the informal contract we wrote and formalize it in a proof assistant or formal semantics. Currently, our confidence in determinism comes from testing and hashing. A future project could be to define a small-step semantics for the logic simulator and prove that, for example, replaying events in that semantics leads to an equivalent state. This would connect our work to programming languages research more directly, possibly framing it as a case study in mechanized semantics for interactive programs. It would also allow proving properties like "hash equality implies state equality" given the normalization function – something we assume but could be proven about the serialization. We could leverage the fact that our state and events are fairly data-structured (JSON-like) which are amenable to formal reasoning.

2. Wider Educational Deployment: As implied earlier, making the determinism features more user-friendly for general audiences is a logical next step. This might mean creating a polished UI for recording and replay that could be used in a classroom without enabling a dev mode. We could introduce features like: - **Save/Load Session** buttons in the main UI (with appropriate warnings about unsupported browsers if any). - Possibly a way to annotate or narrate replays, turning them into guided tutorials. - Integration with learning management systems, so that an instructor can embed a deterministic simulation in a webpage or assignment. - Running on more platforms (ensuring mobile browser compatibility, so students with tablets/phones can also use it).

3. Productizing for End-Users (Minimal Product Features): Outside of education, determinism can enhance general user experience in some subtle ways: - **Crash Recovery:** If the simulator or editor crashes (or the tab closes), a recording of the session could allow the user to restore to the last known state. Since we can deterministically replay, we could effectively "live save" the event log and if something goes wrong, just replay it after restart. We'd have to capture any nondeterministic things like random seeds or times if needed, but given our design, that's handled. Implementing this would involve writing the log incrementally to persistent storage (like browser IndexedDB or localStorage) and then detecting crashes. - **Enhanced Bug Reports:** As mentioned, we can integrate a feature for users to directly submit their session log when filing a bug report, making developer debugging far easier. - **Collaboration via Shared Logs:** Not real-time, but turn the export/import into a way to share "experiments." For example, a user could post a cool circuit along with a demonstration log on a forum. Anyone could load it and see it run exactly as on the original author's machine. This is almost like sharing a recording or a small video, but it's interactive and inspectable. It could foster a community resource of example circuits and their behaviors.

4. Extending to Collaborative Scenarios: While networked determinism was a non-goal, one could consider building on the solid single-user determinism to tackle multi-user. For example, if two people wanted to simulate together, one might explore a model where they take turns (so events are still linearized in one log). That's trivial deterministically (just merge inputs), but not really concurrent. A bigger challenge: allow divergence and merging (like each has a log of their own actions, how to merge them deterministically?). This edges into research on *operational transformation* or CRDTs (conflict-free replicated data types) for merging event streams. While we won't claim an easy solution, having the single-user log as

a base might help, e.g., by tagging events with user IDs and using some deterministic merge policy. This is clearly speculative and would require careful thinking about consistency vs. user experience.

5. Handling Nondeterministic Extensions: If in future we add features that inherently have nondeterminism (say, a random number generator for some circuit element, or an analog simulation with floating point rounding differences), we'll have to extend the determinism infrastructure. Future work could be to incorporate such nondeterministic features in a controlled way. For instance, if randomness is needed, we could generate a random seed as an event so that subsequent random choices are derived from that seed (making them replayable). Or for analog values, we might specify tolerances or quantization to maintain determinism. Each such extension would likely result in a new version of the determinism contract (as per our contract stability clause [82] [89]).

6. Performance Optimization and Scaling: Currently, determinism has been proven on relatively modest examples. Future work could try to optimize for scale: - Implementing state checkpointing to avoid replaying from scratch for very long logs (e.g., store every 100th state so `getStateAtIndex` can jump closer and then replay the last 100 events). This introduces complexity but could be optional. - Parallelizing parts of replay or hashing if circuits get huge (though in JS that's limited due to single-thread, but Web Workers could be used). - Investigating memory overhead of logs and perhaps introducing compression (the JSON could get large; maybe we allow binary encoding or just compress the stored JSON). - Ensuring the system remains deterministic under heavy load or extended runtime (sometimes long-running JS apps can have subtle issues like numeric precision over time – likely not an issue for us since logic values are just 0/1 and small integers).

7. Broader Applicability: Perhaps the most speculative idea is to apply the approach to different domains. For example, could we create a "deterministic interactive Python tutor" where student code execution is logged and replayable? Or a UI builder where user interactions with a UI prototype are logged so that designers can see exactly how testers went through their UI? RedByte OS provides a pattern: separate logic from UI, capture high-level events, ensure determinism of logic, verify with hash. That pattern might be reusable. It would be interesting to research what the requirements are to make an arbitrary interactive system deterministic – not all will fit our model, but many could with restrictions.

8. Community and Standardization: If deterministic replay of this sort catches on, one could imagine a standard format for event logs of interactive systems, or integration with version control (like capturing interactions as part of commit history for certain projects). We're far from that, but every field has to start somewhere. We might consider releasing RedByte OS's determinism framework as an open library or standard, so that other web-based STEM tools could adopt it. This would be more of a social/engineering effort than a research one, but it aligns with the goal of improving reproducibility in education and beyond.

In pursuing future work, we'll remain vigilant not to break the core guarantees. Any addition would go through the same rigor: update the contract, implement with tests, and allow turning off determinism checks to see if anything fails when they're on.

To wrap up, RedByte OS's deterministic interactive computation system is a **foundation**. We built the foundation first (milestones A–D) and now stand at a point where the fun part is figuring out what structures we can build on top of it. We foresee a richer educational platform, new research experiments, and maybe inspiring other tools to adopt similar ideas.

The key takeaway we want to leave is that **determinism in interactive systems, once thought unnecessary or too hard, is achievable and can unlock obvious benefits** now that it's available. Going forward, we anticipate refining this work and exploring those benefits in practice, while clearly delineating which aspects are proven versus which are exploratory.

---

1 2 README.md
file://file_00000000d22c71fd97cb56b97078d884

3 4 5 6 7 8 9 10 15 16 17 18 19 20 21 27 28 35 36 37 38 39 42 45 46 49 54 72 85 87 CONTRACT.md
file://file_000000005f0071fda83cb2d3252109c4

11 33 34 40 47 48 50 51 52 55 56 milestone-a.md
file://file_00000000d17471fdbb57de6a61083aa5

12 13 14 53 57 58 59 86 88 milestone-b.md
file://file_00000000c03871fdb9853fdce1025768

22 23 24 25 26 60 61 62 63 64 65 66 67 68 69 70 71 77 milestone-c.md
file://file_000000001c8071fdbce3ad6af213732a

29 30 31 32 41 43 44 73 74 75 76 78 79 80 81 82 83 84 89 milestone-d.md
file://file_00000000d39071fd92e51003dea1e0ce