Oblivious Computation

Solid-State Distributed Oblivious Computation

Introduction: Computing Without Remembering

Most computer systems work by remembering everything. They pass messages along chains, record who said what and when, and store long histories so they can reconstruct the past in order to decide what is true now. This works, but it comes with a cost. As systems grow, they become heavier. More memory, more coordination, more bookkeeping. Eventually, the effort spent remembering outweighs the effort spent computing.

Oblivious Computation (OC) begins with a simple question: **what if correctness did not require remembering the past at all?**

A useful analogy comes from math. Before the concept of zero, numbers only moved forward. You could count higher and higher, but you could not subtract, cancel, or express absence. Zero made subtraction possible, then negative numbers, then entire new kinds of mathematics. Zero was not powerful because it added something, but because it allowed things to be taken away.

OC applies the same idea to computing systems. Instead of building truth by adding messages and history, it builds truth by *removing* everything that does not survive. Correctness is not negotiated, voted on, or reconstructed. It is simply what remains after all competing possibilities are erased.

A simple way to picture this is three frogs sitting on a log. In most systems, Frog A whispers to Frog B, Frog B whispers to Frog C, and the meaning depends on order, memory, and whether anyone made a mistake. OC skips the whispering. The frog jumps up and croaks once so everyone hears it at the same time. There is no chain, no path, and no need to remember where the message began.

At this point, a natural objection arises: **what if a frog lies?**
That question matters in traditional systems. In OC, it turns out not to matter in the same way.

Explanation: Why Lying Doesn't Matter, and Why it's so light

Each frog can only report its own state—for example, how many flies it has eaten today—and the system only cares that the total state makes sense. If a frog lies, that lie either disappears or becomes the truth, but it cannot spread, compound, or corrupt anything else. Byzantine actors do not break the system; they simply do not matter.

This works because Oblivious Computation (OC) does not try to prevent bad information from appearing. It does not punish it, track it, or argue with it. Bad states are treated exactly like good states: they compete. If they lose, they vanish completely. There is no replay, no appeal, and no memory of the disagreement. What survives is the state. Everything else is gone.

Traditional distributed systems are slow and complex because they assume correctness requires memory. Messages are daisy-chained, order is tracked, logs are stored indefinitely, and constant coordination is required to keep histories aligned. Most of the work is not computation at all—it is bookkeeping. The system spends its time remembering the past instead of settling the present.

OC exists because this assumption is not always necessary. If messages are not chained, there is no path to track. If histories are not merged, they do not need to be stored. If conflicting states are erased instead of reconciled, the system becomes lighter by construction. Computation shifts from accumulation to elimination. Truth is not built up; it is what remains.

Formalization: Oblivious Computation as a Solid-State Arbiter

Formally, Oblivious Computation (OC) operates by exchanging **complete candidate states** rather than incremental messages. Each participant may propose a full representation of the system's current state. Multiple candidate states may exist transiently, but they are never merged or reconciled.

When multiple states are observed, a deterministic selection rule is applied. Exactly one state survives. All others are erased permanently. Participants overwrite their local state with the surviving state and continue. No record of discarded states is retained.

Because full states are shared between peers and selection is deterministic, the system behaves as a **distributed arbiter**. In hardware, an arbiter selects one outcome from many without negotiation or memory of rejected inputs. OC exhibits the same behavior at the level of global state. There is no voting, no quorum, and no agreement protocol. Selection is enforced by overwrite and erasure.

This behavior is **solid-state** rather than conversational. Many candidate configurations may exist briefly, but only one stable configuration remains observable. Stability emerges through elimination, not coordination. Discarded states leave no trace.

The system is **memoryless by design**. Once a state is erased, it has no influence on future behavior. There is no rollback, no audit trail, and no historical reconstruction. Correctness is defined operationally: **the surviving state is the truth**.

OC does not attempt to provide fairness guarantees, ordered histories, auditability, or Byzantine agreement. These are intentional non-goals. The model trades historical accountability for immediacy, simplicity, and erasure-based finality. False or adversarial states are not prevented from appearing, but they are not privileged. They either survive or vanish.

This places OC in a different class from consensus protocols, leader election, and eventual consistency systems. It is not agreement-based. It is selection-based.

Conclusion: Truth Through Erasure

Oblivious Computation (OC) reframes distributed correctness as a process of selection and erasure rather than accumulation and agreement. By removing the requirement to remember the past, it eliminates entire classes of coordination, bookkeeping, and historical overhead that dominate traditional systems.

Just as zero enabled subtraction and negative numbers, OC enables a form of deductive computation—computation by removal. The system does not ask which history is correct. It asks which state survives.

The result is a lighter, simpler, and more direct way to compute. Not faster because of optimization, but faster because it refuses to carry what it does not need. Sometimes truth is not found by remembering more, but by eliminating everything that does not remain.

Final Note

Oblivious Computation is intentionally not a universal replacement for history-based systems. It is best suited for environments where immediacy, simplicity, and finality matter more than auditability, fairness, or long-term accountability. Systems that require detailed provenance, reversible decisions, or post-hoc inspection are better served by memory-heavy approaches. This paper presents Oblivious Computation as a foundational primitive, not a finished product. If the idea is wrong, it is wrong cleanly; if it is right, it opens a new design space where computation no longer depends on carrying the past forward.