Client-Hook Driven Live Streaming of Game Events with Transparent Token Rewards

Johan B

2025-07-01

Abstract

Competitive online games increasingly offer token-based rewards, but third parties rarely have reliable, real-time access to tamper-evident event data from official game servers. This paper presents Uppercut, a server-agnostic system that records important in-game events on the client and sends signed reports to an external aggregator. The goal is to validate events without privileged server access, so that rewards, analytics, and match reconstruction can rely on a clear standard of evidence. We describe two deployment options: a simple supermajority-based aggregator for low-cost, scalable use, and a Byzantine fault tolerant variant for high-stakes settings. Both tolerate partial data coverage and dishonest clients. The design shows that secure, real-time event verification can be added on top of existing games and can support token rewards and other monetization schemes without changing the game servers.

1 Introduction

This work specifies and evaluates a system for capturing, verifying, ordering, and distributing in-game events from matches that run on official, publisher-operated servers. We refer to this system as *Uppercut*. The starting point is simple. Popular games do not expose real-time server logs to third parties, but the wider ecosystem wants trustworthy signals to power rewards, analytics, and fair-play checks. Running custom game servers is one workaround, but it splits the player base and is often rejected by communities that prefer official queues. Uppercut addresses this by moving data capture to the client, by using only sanctioned local event feeds and title-provided local APIs where possible, by binding reports to platform identities in the live roster, and by finalizing events through cross-endorsement and explicit ordering rules instead of privileged server access.

The system model follows constraints seen in real titles. Event visibility is asymmetric across roles: for example, a killer and a victim may both see a kill, while some teammates or spectators do not. Some signals are ephemeral, such as a one-shot kill callback. Others are persistent, such as a scoreboard value that can be sampled multiple times. Anti-cheat software limits generic code injection, so data capture must prefer publisher-permitted interfaces or title-specific local APIs when available. Network issues and client restarts produce gaps in coverage. A single client's view is therefore incomplete, and correctness must come from correlating multiple perspectives, applying endorsement rules, and checking consistency against persistent fields.

The core mechanism is narrow and explicit. A lightweight overlay subscribes to a title's sanctioned local event feed. For each match, the client derives a per-match signing key from a long-lived account key using HKDF. Every captured event is encoded as a canonical payload that includes the platform identity, match identifier, event type and role tuple, and a coarse time bucket. The client signs this payload and submits it to an external aggregator. The aggregator verifies signatures, groups reports that refer to the same underlying occurrence, and requires endorsements from distinct eligible observers until a threshold is met. Eligibility is tied to platform identities visible locally and to presence rules that require a participant to appear in the roster for a minimum fraction of the match.

Because events have different rates and dependencies, the aggregator maintains multiple per-type counters and a directed acyclic graph (DAG) that records per-client, per-counter sequencing and title-level constraints, such as a round start preceding kills in that round. The DAG can be topologically sorted when a linear replay is needed. If a title does not present the same event to all observers, the aggregator assigns a global event identifier on first sight and links later endorsements to that identifier, so all signers refer to the same label.

Two ordering and trust envelopes are supported. In the first, an operator runs a service that applies endorsement and ordering rules, publishes finalized timelines, and periodically anchors manifests in decentralized storage. In the second, event proposals and endorsements go to on-chain logic on a fast BFT stack so that ordering and finality follow from consensus. The first model keeps operational cost low while still providing transparency through open formats and anchors. The second model removes trust from the operator at the cost of tighter latency budgets and more complex integration. In both models, persistent fields such as scoreboards are sampled by multiple clients and used to detect impossible inflations. If an aggregated count exceeds recent snapshots beyond a configured tolerance in value and time, the implicated participant is marked ineligible until reviewed.

Access control for archived logs is handled off-chain. Finalized timelines and auxiliary snapshots are written to content-addressed storage and encrypted under multi-authority attribute-based encryption (MA-ABE) policies. Independent authorities issue attributes that represent roles such as referee, league administrator, or analytics partner, and negative constraints and revocation are expressed in the policy. A referee can decrypt raw event lines for a finished match, while an active competitor cannot decrypt a live feed. This keeps the on-chain footprint small, lets organizers control who can read which parts of the record, and avoids exposing platform identifiers in clear form by storing only hashed or salted versions at rest.

For publishers and organizers, the main benefit is the ability to support verifiable rewards, anti-cheat analytics, and creator tools on official servers without changing server code or distributing privileged keys. For developers, the paper acts as a concrete contract. It defines canonical payloads, domain separation strings, per-match key derivation, nonce and replay rules, endorsement and eligibility policies, counter and DAG semantics, failure handling including skip-forward conditions, wire formats for ingestion and status streams, storage manifests, and MA-ABE policy templates. It also states measurable targets for validation, such as the fraction of critical events captured via sanctioned feeds, median and tail latency from capture to finalization for a chosen endorsement threshold, and false-positive and false-negative rates in scoreboard consistency checks.

The contributions are practical. The design shows how to build a standards-of-evidence pipeline that works with anti-cheat constraints, reconciles partial and asymmetric views without server logs, expresses ordering using multiple counters and a DAG instead of a single global log, binds endorsements to real platform identities on public servers, and publishes archives that are both privacy-preserving and selectively auditable under multi-authority control. The rest of the paper motivates the threat model and compliance stance, defines the wire formats and state machines, explains the endorsement and ordering algorithms, specifies the storage and MA-ABE layer, describes both deployment options, and outlines an evaluation plan that measures coverage, latency, and robustness under induced failures.

1.1 Integration with Existing Gaming Ecosystems and Commercial Use

The system is designed to sit on top of tools and platforms that players, creators, and organizers already use, such as live streaming services and community servers like Discord. Streaming is a primary way to discover competitive play and follow live service updates, and community hubs keep interest active between broadcasts. Uppercut turns sanctioned, real-time match events into machine-readable signals that broadcasters can use to trigger overlays, rules, and audience rewards without screen scraping. The same signals let community tools post verified highlights, match recaps, and role updates, so that play naturally feeds back into sharing.

The system uses only publisher-sanctioned data surfaces, such as Overwolf Game Events Provider or title-provided local APIs, and does not inject code or scrape the screen. Commercial use follows a simple allowlist. Event-driven rewards that rely only on sanctioned APIs are allowed. Post-match or lobby overlays that show only the application's own interface, with no game pixels or logos, are allowed. Sponsor or ad elements may appear inside the application window or in pre- and post-game views once the user has given explicit consent. Streams or VODs may be monetized on platforms that allow it, when the publisher's policy for that title permits ads on web videos. Partner backends may receive signed events for drops, watch-to-reward campaigns, or analytics. Platform identifiers are stored in hashed or salted form. Clear disclosures are shown when required, for example stating that the tool is not affiliated with the game publisher.

A short list of disallowed uses keeps risk low. In-play commercial overlays that cover or obscure the game are disabled. Any non-sanctioned capture, reverse engineering, or code injection is out of scope. Publishing or monetizing game video, screenshots, or user interface is disabled where a publisher's policy forbids it. Game trademarks or key art are not used in ads without permission, and endorsement is never implied. If a title's policy is unclear, the client defaults to a mode without commercial overlays and records events only.

Deployment follows directly from these rules. Low-latency, signed events reduce the friction of working with streamers and tournament organizers by offering simple, auditable signals for on-air graphics, co-stream rules, and audience rewards. Community features such as quests, community cups, attribution for creator codes, and highlight prompts are driven by signed events rather than screenshots, so operators can attribute outcomes with confidence and without operating custom servers. At runtime, a per-title policy gate enforces these constraints. The Uppercut client loads a signed policy for the active game, enables only the permitted surfaces for that title, keeps any sponsor interface out of gameplay unless explicitly allowed, requires user consent for commercial elements, and disables capture-to-publish when media reuse is restricted. The result is a system that can support engagement and monetization built on top of official servers while staying within publisher terms.

2 A Server-Agnostic Hooking Approach

Modern multiplayer titles often expose client-side hooks or callback APIs that allow external overlays to receive real-time notifications about kills, score updates, or other in-game triggers [1, 2, 3]. A lightweight application on the player's machine subscribes to these hooks and collects signed event records as they occur. This avoids the need for privileged access to game servers: engine-confirmed actions are translated into a simple, verifiable event stream. Each event is signed locally (using a per-match key derived from a long-lived account key and stored in an OS keystore when available) and sent to an external aggregator, which reconciles input from multiple participants before finalizing the match log.

This model has to cope with partial coverage. Some event categories may be missing if the API is incomplete or a patch temporarily disables callbacks. Players may also enable the overlay late, crash mid-match, or lose connectivity, so their streams contain gaps. By comparing reports from several players, the aggregator still recovers most of the session. Events with endorsements from multiple participants are treated as confirmed; lone, uncorroborated reports can be downgraded or discarded under policy.

To finalize a match record, the aggregator observes the incoming streams until all participants have ended the session or have been silent for a configured interval. It then marks the log as complete or partial, credits honest players for the events that did meet policy, and withholds rewards from participants whose data is too sparse to support them. When the surrounding system uses cryptocurrency rewards or staking incentives, the aggregator's record is the anchor for payouts or penalties. In a play-to-earn setting, tokens are minted or unlocked based on confirmed events, without requiring players to move to custom servers.

Because this design only depends on client authorization and sanctioned local APIs, it remains usable even when server implementations are closed and cannot be modified. Tournament organizers, play-to-earn platforms, and community projects do not need to host their own servers. Any player who installs the overlay can contribute verifiable data; cross-verification across clients keeps any single perspective from dominating the outcome.

2.1 Event Feeds, Anti-Cheat, and Compliance

Uppercut prioritizes sanctioned or publisher-permitted data sources. Frameworks such as Overwolf's Game Events Provider expose curated event streams for specific titles, while some games publish local APIs (e.g., CS:GO/CS2 Game State Integration; Riot's Live Client Data API) that allow read-only state. Steamworks provides a first-party overlay and services for *developer-owned* titles; it is not a general third-party hooking substrate. Low-level interception (e.g., binary detouring) is powerful and historically relevant [3] but may conflict with modern anti-cheat policies. In production we recommend: prefer sanctioned feeds; fall back to game-local APIs when available; use low-level interception only where it is explicitly permitted and tested. A per-title mapping of available surfaces and identity bindings is provided in Appendix A (Table 1).

3 Threat Model and Assumptions

We assume adversaries ranging from a single dishonest client to colluding groups (e.g., killer and victim), Sybil identities attempting to outvote honest players, and transiently faulty aggregators. Each match has a roster of eligible participants (issued by the organizer). Only rostered identities may endorse events. Clients derive a short-lived per-match signing key from an account key; private keys are kept in OS keystores when available. Aggregator time is the clock of record for ordering; client timestamps are auxiliary. Visibility varies by title: some events are visible to all, others only to specific roles (killer, victim, spectators). Policy specifies eligible observer sets and thresholds per event type.

In the *lightweight* deployment envelope (Section 8), we treat the aggregator as *honest-but-curious*: it is expected to verify signatures, apply policies, and publish accurate manifests, but it may observe and log all submitted data. The *direct on-chain* envelope (Section 9) instead assumes the aggregator role may be fully Byzantine and shifts ordering and finality to a BFT consensus layer. This separation lets operators choose between simpler infrastructure with a semi-trusted service and a stronger, fully decentralized model.

3.1 Identity Binding and Roster Verification

On private or tournament servers, organizers issue a roster that binds wallet or platform accounts to participants. On public servers, a credible binding remains possible using platform-level identities exposed by local interfaces (e.g., SteamID via CS:GO/CS2 GSI, Summoner identity via LoL Live Client). Uppercut ties each per-match signing key to the verified platform identity and `match_id`. Endorsement eligibility checks require presence in roster for at least a configured fraction of the match, matching platform identity to the event's observer set.

4 Detecting and Classifying Incomplete Data

Even robust hooking frameworks may fail to capture every in-game event, whether due to network disruptions, version mismatches, or partial client disconnections. These issues result in incomplete event logs and complicate efforts to reconstruct a precise match history.

4.1 Causes of Hook Failures and Partial Coverage

Hooking failures can arise from several practical circumstances. For instance, delayed hook registration may cause early match events to go unrecorded if the hooking interface has not finished initializing before play begins. Mid-session failures also occur when clients crash or lose connectivity, abruptly interrupting their data streams and leaving gaps in the event log. In addition, version mismatches between game patches and hooking software can prevent certain categories of events from being recognized at all. Lastly, some users may tamper with the hooking interface by disabling or modifying it, thereby reducing the visibility of key gameplay actions.

4.2 Detecting Coverage Gaps and Scoreboard Mismatches

To identify partial coverage, our aggregator monitors the incoming event streams for unexpected drops in frequency or missing entries around known checkpoints, such as round transitions. Whenever persistent data like scoreboard updates conflicts with the recorded events, the aggregator flags this mismatch for further review. In some cases, a single client's missed updates can be inferred from corroborating reports or scoreboard tallies. However, if multiple clients fail to report the same ephemeral event, the data gap becomes irrecoverable in the final timeline.

4.3 Classifying and Finalizing Partial Matches

If coverage gaps persist until the end of a match, the aggregator compiles all confirmed events in chronological order and annotates intervals where data is incomplete. Clients that have repeatedly failed to transmit events or gone offline are recorded as partially active or unresponsive. Depending on the policy in effect, these clients may be ineligible for certain rewards or flagged for additional inspection. Despite imperfect reporting, this process ensures that legitimate events remain recognized and traceable, enabling fair post-match analysis and consistent rule enforcement.

5 Security Measures and Cheating Mitigation

Ensuring that reported events are both complete and free of manipulation is vital to our design. Although technical failures account for some missed data, malicious actors may also attempt to forge or suppress events for personal gain. Our system combines threshold endorsements, cryptographic checks, and incentives to deter cheating while preserving an auditable record.

5.1 Threshold Endorsements and Cross-Verification

The aggregator compares event submissions from multiple participants, requiring a minimum number of endorsements before finalizing each event. For a kill, this may entail confirmation from both killer and victim, or a quorum of present players. When an event does not meet its endorsement requirement, it is discarded. Persistent scoreboard tallies further bolster data integrity: if the scoreboard indicates a total that exceeds the number of confirmed kill events, the aggregator highlights potential missing data, while additional kill submissions that cannot be reconciled are flagged as suspicious.

5.2 Event Identity and Multi-Signer Endorsements

To ensure that endorsements refer to the same underlying occurrence, each event is bound to a shared identity `eid`. Depending on policy, `eid` is either (i) *aggregator-assigned* upon first report, after which observers reference the issued `eid`, or (ii) *content-addressed*, where clients compute `eid` $= H(\mathsf{domain} \,\|\, \mathsf{canonical\_payload})$ that includes `game_id`, `match_id`, event type, roles (e.g., killer and victim), and a coarse time bucket. Endorsements aggregate over **distinct rostered identities** for the same `eid` until the threshold is reached; role-aware policies (e.g., killer and victim plus a third-party observer) can be enforced when the title exposes such visibility. We use domain separation, e.g., `"UPPERCUT|eid|v1"`. In practice, `eid` is instantiated with a 256-bit hash function so that collisions are negligible for the lifetime of the system.

5.3 Cryptographic Checks

To combat tampering during transmission, each client signs a canonical encoding of its event fields (excluding the signature itself) with a private key. The aggregator verifies these signatures upon receipt, discarding any submissions with invalid signatures, altered payloads, or reused nonces for the same client and match. This mechanism ensures the final match record remains verifiably authentic, as each event can be rechecked or anchored to a public ledger. When feasible, per-match signing keys are derived from a long-lived account key and stored in the OS keystore or TPM; replay protection is enforced with per-match nonces that the aggregator tracks per client. We recommend HKDF [10] with `match_id` as salt to derive per-match keys under EdDSA [9].

5.4 Deterring Fraud Through Incentives

When participants submit misleading or conflicting data, the system can impose penalties ranging from collateral forfeiture to disqualification from token rewards. Repeated failures to provide accurate event logs may prompt external audits or exclusion from competitive matches. By aligning financial rewards with truthful reporting and imposing tangible costs for dishonest behavior, our framework upholds data quality across diverse game servers and play-to-earn environments.

5.5 Game-Specific Event Mapping and Multiple ID Counters

Many modern hooking frameworks deliver game events at widely varying rates and from distinct sub-systems. One title may send a "kill" notification to all participants, while another only notifies the killer and victim. Some produce numerous minor updates (like health changes) alongside critical milestones (like round transitions). In a single aggregated timeline, forcing every event to share the same localID counter can lead to large jumps or out-of-sync sequences, especially if one category (e.g. kills) is much rarer than another (e.g. health ticks).

To manage this complexity, the aggregator can load a `game_id`–specific configuration that details exactly which event types warrant their own counters and how each should increment. For instance, a "killCounter" might track lethal actions, while a "healthCounter" increments on each health-related update. When a client-side hooking overlay receives an event from the game, it checks the relevant configuration (potentially retrieved from IPFS[6] or a local module), then increments or omits the appropriate counter before transmitting the event to the aggregator.

On the aggregator side, each counter can have distinct finalization rules. One counter may require two signers (killer, victim) for a kill to become valid, while another might need a majority endorsement for a scoreboard update. A counter can also be designated as "non-blocking," allowing subsequent IDs in the same category to finalize even if an earlier one is incomplete. If multiple counters exist, each forms a parallel sequence in the aggregator's DAG. Events in different counters do not necessarily block or depend on one another unless a specific game rule (e.g. "scoreboard updates must follow all kills") imposes a cross-categorization edge. This modular approach ensures that concurrency and partial coverage are handled on a per-event-type basis, yielding a more robust final timeline.

5.6 Partial vs. Total Ordering for Local IDs

Even with multiple counters, individual event sequences may face uncertainties such as missing endorsements. A kill at localID 2 may remain stuck in a "pending" state if it never attains enough signatures, yet localID 3 could reach threshold and be ready to finalize. One method is to allow a *partial order*: the aggregator's DAG can finalize localID 3 without waiting for localID 2, reflecting that some events stall indefinitely. This is often sufficient if the game logic only needs each event's eventual confirmation, without insisting on strict increments.

However, some games require **strict sequencing** within each local ID stream, so the aggregator cannot finalize localID $n + 1$ until localID n is either confirmed or explicitly skipped. This approach yields a **total order** for that sequence, ensuring that missed events block later ones unless the aggregator (according to game-specific rules) forcibly discards them as "unrecoverable." Such strict ordering is useful for titles that rely on a guaranteed incremental progression (e.g. awarding a special bonus to the "*third* confirmed kill" of the match).

In practice, different counters may adopt different ordering rules. Kills might be strictly sequential, so the aggregator never skips an unconfirmed kill ID, while health updates can finalize out of order. At a higher level, if the game or an external analytics system desires a single linear timeline of all events across all counters, the aggregator can periodically merge them. A topological sort of the DAG—maintained in near real time—assigns each newly confirmed event an absolute position in a global "match timeline," breaking ties deterministically if events are concurrent. This process allows partial concurrency in the DAG internally while still producing a coherent total order that can be published for replay, tie-breaking, or advanced cross-event logic (e.g. "the first kill after a round transition is worth double points"). By mixing partial or total ordering, multiple counters, and game-specific thresholds, the framework can handle extremely diverse hooking behaviors under a single aggregator architecture.

5.7 Aggregator-Assigned IDs

While our main design relies on client-side increments (`localID`) to track each client's event sequence, certain titles or external systems do not consistently present the same events to every participant. In such cases, the aggregator can assign global identifiers to each newly reported event, rather than trusting the client to increment a local counter. Under this approach, when a client reports an event, the aggregator confirms the authenticity of the submission but does not finalize it immediately. Instead, it issues a globally unique label (for example, an integer counter or UUID) and temporarily stores the event in a pending state.

After assigning this global ID, the aggregator informs all relevant clients or observers—such as the killer and victim if it is a combat-related event—that "Event #37 has been proposed." Those who see or wish to endorse this event sign off on it by referencing the aggregator's global ID. Rather than referencing (`clientID`, `localID`), each confirmation now directly cites "Event #37." This ensures all parties share a single label for the same underlying action, regardless of whether each client originally recognized the event or incremented an internal counter.

Aggregator-assigned IDs can be activated on a per-game basis, especially for scenarios in which partial coverage is common or where multiple clients might hold inconsistent local counters. A game that already provides stable, universal feeds can retain client-side increments with no problem, while another with more fragmented or delayed event notifications may benefit greatly from a single aggregator-issued label. Each client that confirms the event references the same label, preventing confusion if one client's local IDs differ from another's.

Implementation Note. Aggregator-assigned IDs are for cross-client *event identity*; they do not replace per-client sequencing. Clients *must continue* to maintain and include per-counter `localID` values so the aggregator can enforce strict or partial ordering per client/counter stream. The aggregator uses the global `eid` solely to unify endorsements across observers, while ordering rules continue to reference (`clientID`, `counter`, `localID`).

6 Detailed Algorithmic Workflow

This section outlines the core algorithms of the event capture and aggregation framework. The goal is to show how the system keeps per-client sequencing, validates authenticity, and synchronizes event data under realistic match conditions.

6.1 Client-Side Hook Capture

First, the client checks hook availability for all required event types and registers callbacks. If available, it snapshots persistent state, then on `matchStart` it loads the per-game configuration that maps event types to counters and records whether event identities will be content-addressed by the client or assigned by the aggregator.

Algorithm 1 Client-Side Hook Capture (Part 1: Steps 1–4)

```
 1: procedure LocalEventCapture(pk, sk, A, gameID)
 2:     /* Step 1: Hook Version Support */
 3:     for all hook in {match, round, kill, ...} do
 4:         if Unavailable(hook) then
 5:             report “HOOK VERSION ERROR”
 6:             return
 7:         end if
 8:     end for
 9:     report “HOOK VERSION OK”
10:     /* Step 2: Register Hooks */
11:     if RegisterHooks() = Fail then
12:         report “HOOK REGISTRATION ERROR”
13:         return
14:     else
15:         report “HOOK REGISTRATION OK”
16:     end if
17:     /* Step 3: Optional Snapshot */
18:     if SupportsGetInfo() then
19:         snapshot ← GetInfo()
20:         SignAndSend(snapshot, sk, A)
21:         report “SNAPSHOT OK”
22:     else
23:         report “SNAPSHOT NOT AVAILABLE”
24:     end if
25:     /* Step 4: Match Start Check */
26:     if not ConfirmMatchStart() then
27:         report “MATCH START ERROR”
28:         return
29:     else
30:         localCounters ← empty map with default 0
31:         cfg ← FetchConfig(gameID)
32:         report “MATCH START OK”
33:     end if
34: end procedure
```

Algorithm 2 Client-Side Hook Capture (Part 2: Steps 5–6, continuing from Part 1)

```
 1: procedure LocalEventCapture_Loop(pk, sk, A, gameID)
 2:     $\mathcal{Q} \leftarrow \varnothing$
 3:     while MatchOngoing() do
 4:         rawEvent ← NextHookEvent()
 5:         type ← rawEvent.type
 6:         cName ← MapToCounter(type, gameID)
 7:         localCounters[cName] ← localCounters[cName] +1
 8:         $E$.localID ← localCounters[cName]
 9:         $E$.eventType ← type
10:         $E$.gameID ← gameID
11:         $E$.payload ← rawEvent.payload
12:         $E$.timestamp ← LocalTime()
13:         $E$.nonce ← NextNonce()
14:         $E$.signature ← Sign(CanonicalEncode($E$ without signature), sk)
15:         $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{E\}$
16:         Send(E, A)
17:         if MatchEndConfirmed() then
18:             report “MATCH END OK”
19:             Cleanup()
20:             break
21:         end if
22:     end while
23:     if not MatchEndConfirmed() then
24:         report “MATCH END ERROR”
25:     end if
26:     return $\mathcal{Q}$
27: end procedure
```

6.2 Aggregator Threshold Logic

Each event is validated, bound to policy, and aggregated under a shared `eid`. Sequencing rules are enforced per counter; confirmed events populate a DAG that can be topologically sorted.

Algorithm 3 Aggregator Ingestion and Multi-Signer Finalization

```
1: procedure AggregatorThresholdIDSync($\mathcal{C}$, policyDB)
2:     for all $c \in \mathcal{C}$ do
3:         MaxConfirmedID[$c$] ← empty map with default 0
4:         HighestReportedID[$c$] ← empty map with default 0
5:         SeenNonces[$c$] ← empty set
6:     end for
7:     trackerMap ← ∅                    ▷ $eid \mapsto$ payload, signers, first_seen, status
8:     pending ← map with default empty sets per ($c, cn$, localID)
9:     while Running() do
10:        $e$ ← GetNextEventRecord()
11:        if $e$.clientID $\notin \mathcal{C}$ or not VerifySig($e$) then
12:            report "INVALID EVENT"; continue
13:        end if
14:        if $e$.nonce ∈ SeenNonces[$e$.clientID] then
15:            report "REPLAY DETECTED"; continue
16:        else
17:            SeenNonces[$e$.clientID] ← SeenNonces[$e$.clientID] ∪{$e$.nonce}
18:        end if
19:        $p$ ← policyDB[$e$.gameID][$e$.eventType]; $cn$ ← $p$.counterName
20:        $eid \leftarrow \begin{cases} \text{AssignGlobalID}(); & p.\texttt{aggregatorAssignedIDs} \\ H(\text{domain} \,\|\, \text{CanonicalPayload}(e)); & \text{otherwise} \end{cases}$
21:        if $eid \notin$ trackerMap then
22:            trackerMap[$eid$] ← {payload: CanonicalPayload(e), signers: ∅, first_seen: Now(), status: Pending}
23:        end if
24:        trackerMap[$eid$].signers ← trackerMap[$eid$].signers ∪{$e$.clientID}
25:        pending[$e$.clientID][$cn$][$e$.localID] ← pending[$e$.clientID][$cn$][$e$.localID] ∪{$eid$}
26:        HighestReportedID[$e$.clientID][$cn$] ← max (HighestReportedID[$e$.clientID][$cn$], $e$.localID)
27:        TryAdvance($e$.clientID, $cn$, $p$, trackerMap, pending, MaxConfirmedID, HighestReportedID)
28:        if |trackerMap[$eid$].signers ∩ EligibleObservers($eid, p$)| ≥ $p$.threshold then
29:            $S_{\text{loc}}$ ← pending[$e$.clientID][$cn$][$e$.localID]
30:            if $\exists eid' \in S_{\text{loc}}$ with $eid' \neq eid$ and trackerMap[$eid'$].status = Final then
31:                report "CONFLICTING FINAL EID"; continue
32:            end if
33:            trackerMap[$eid$].status ← Final; BuildAndOrderDAG(. . . )
34:        end if
35:    end while
36: end procedure
```

Algorithm 4 Strict/Partial Sequencing with Hybrid Skip-Forward

```
1: procedure TryAdvance($c, cn, p$, trackerMap, pending, MaxConfirmedID, HighestReportedID)
2:     currentID ← MaxConfirmedID[$c$][$cn$] +1        ▷ MaxConfirmedID counts IDs that are Final or Skipped
3:     while Exists(pending[$c$][$cn$][currentID]) do
4:         $S$ ← pending[$c$][$cn$][currentID]
5:         if ExistsFinal($S$, trackerMap) then
6:             MaxConfirmedID[$c$][$cn$] ← currentID; currentID ← currentID +1; continue
7:         else if $p$.seqMode = "partial" then
8:             break
9:         else
10:            $e^\star$ ← Oldest(S)
11:            if (Now() − trackerMap[$e^\star$].first_seen ≥ MIN_WAIT_TIME) and (HighestReportedID[$c$][$cn$] ≥ currentID + SKIP_GAP) then
12:                trackerMap[$e^\star$].status ← Skipped
13:                MaxConfirmedID[$c$][$cn$] ← currentID; currentID ← currentID +1
14:            else
15:                break
16:            end if
17:        end if
18:    end while
19: end procedure
```

6.3 DAG Construction for Confirmed Events

Once events are confirmed, they are woven into a coherent Directed Acyclic Graph (DAG)[8]. Nodes correspond to `eid`s; edges encode per-client per-counter sequencing and global rules.

Algorithm 5 DAG Construction for Confirmed Events

```
1: procedure BuildAndOrderDAG(eidIndex, rules)
2:     DAG ← empty
3:     for all $eid \in$ Finalized() do
4:         AddNode(DAG, $eid$, Payload($eid$))
5:     end for
6:     for all client $c$ do
7:         for all counter $cn$ do
8:             $L \leftarrow$ ConfirmedLocalIDs($c, cn$); Sort($L$)
9:             for $i \leftarrow 1$ to $|L| - 1$ do
10:                AddEdge(DAG, eidIndex($c$, $cn$, $L[i]$) $\rightarrow$ eidIndex($c$, $cn$, $L[i{+}1]$))
11:            end for
12:        end for
13:    end for
14:    for all rule $\in$ rules do
15:        ApplyRuleEdges(DAG, rule)
16:    end for
17:    linearOrder ← TopologicalSort(DAG)
18:    return (DAG, linearOrder)
19: end procedure
```

7 Storing Game Events in Decentralized Storage

Most popular blockchains lack the bandwidth and may cause an extreme overhead in storage size to store full gameplay logs on-chain, making it necessary to rely on off-chain or decentralized storage systems such as IPFS. This section shows how attribute-based encryption (ABE) [7] can protect these logs at scale—keeping only a concise reference (hash/CID) and policy link on-chain. We illustrate with a zero-knowledge extension of ABE (ZK-DABE), but any ABE variant that handles complex policies (e.g. with negative constraints) can be substituted.

Overview of the Storage Flow. After each match, the aggregator (or the game clients themselves) serializes event data into manageable chunks. These chunks might be JSON lines of kills, scoreboard updates, or partial coverage flags. Each chunk is then encrypted under an ABE policy specifying who can read it (e.g. holders of `Referee` or `LeagueAdmin`) and who cannot (`NOT(CheaterFlag)`). Once encrypted, the chunk is uploaded to IPFS, generating a unique content identifier (CID). On-chain, only a minimal record is stored:

$$\{\texttt{matchID},\ \texttt{chunkIndex},\ \text{CID},\ \text{policyRef},\ \text{signature}\},$$

where `policyRef` points to the relevant ABE policy or DAG for more detailed rules. In a typical deployment, `policyRef` is a hash or identifier of a JSON policy document itself stored on IPFS, and that document encodes which Attribute Authorities (AAs) may issue attributes and how policies compose. This small on-chain "anchor" ensures tamper-evident referencing without incurring prohibitive fees.

Decryption and Access Control. To retrieve a chunk, an authorized user (e.g. a referee) fetches the CID from on-chain, downloads the encrypted file from IPFS, and presents a zero-knowledge proof of attribute ownership (and possibly non-ownership) to the appropriate Attribute Authority. If verification succeeds, the AA issues a partial decryption key so that the user can locally run ABE.Decrypt on the chunk. If the user is flagged (e.g. `CheaterFlag`), then any policy with `NOT(CheaterFlag)` blocks them from obtaining the partial key. Since each new chunk is encrypted under the latest policy, revoking an attribute on-chain immediately impacts future logs without requiring re-encryption of old data. The on-chain contract or registry only needs to store policy references and CIDs, while the AAs enforce the actual access decisions.

Example Use Cases. *Referee Access.* Only users with attribute `Referee` can review raw kill feeds. *Streamer vs. Competitor.* A negative constraint `NOT(Competitor)` ensures that active players cannot decrypt live data intended for broadcast. *Analytics Firms.* Aggregated logs might be partially anonymized with policy `AnalyticsPartner` and `NOT(PersonalData)`. *Cheater Revocations.* Once someone is flagged with `CheaterFlag`, the partial keys for new match chunks will not be issued to them.

On-Chain Footprint vs. Off-Chain Scale. Because only the hash/CID and a short policy reference go on-chain, the system can handle thousands of events per second without bloating blocks or incurring high gas fees. IPFS can store arbitrarily large logs, so partial coverage or incomplete data does not pose a problem; each chunk is simply labeled as `partial` or `complete` and published under the same policy. If an aggregator merges multiple clients' event streams, the final per-match timeline is chunked and encrypted the same way.

Conclusion. This ABE-based approach to storage offers granular, dynamically updatable access control over extensive game data archives. By aligning each IPFS chunk with a policy enforced by an Attribute Authority, tournament organizers, referees, or community auditors can retrieve only the logs they are entitled to see, while players' private data remains protected. Integrating zero-knowledge proofs (as in ZK-DABE) adds privacy for both the attribute holders and any sensitive game records, creating a flexible, scalable foundation for decentralized game-event storage.

8 Lightweight Live Event Aggregation - Main Proposal

A lightweight aggregator orders incoming streams of game events using a simple supermajority rule. It does not run a full Byzantine fault-tolerant protocol among many aggregator nodes; instead, it assumes one service (or a small cluster) that verifies signatures, applies policies, and publishes manifests. This suits cases where the cost and operational overhead of a fully decentralized network are not justified. Signed data from game clients is consolidated in one place, finalized, and periodically anchored to a tamper-evident repository.

A typical use case is a gaming community that wants to offer token rewards for achievements but does not plan to run its own chain or consensus cluster. As long as the aggregator exposes its policies and timelines, third parties can replay the final log, check that thresholds were met, and see which events were rejected. The focus is on a straightforward first line of defense against cheating and obvious manipulation.

We treat the lightweight aggregator as *honest-but-curious*: it can see all incoming data, but is expected to apply policies correctly, avoid censoring events, and publish accurate manifests and anchors. Transparency measures such as an open implementation, independent replay tools, and periodic anchoring to IPFS or a base chain make misbehavior detectable after the fact. If operators or use cases need stronger guarantees, the blockchain-based model in Section 9 is a better fit.

8.1 Rationale and Basic Flow

In the lightweight model, clients register for real-time hooks within the game, sign each captured event, and send the signed data to the aggregator. The aggregator then merges these events into a growing dataset, checks their validity (for example, by verifying digital signatures), and uses a supermajority threshold to decide when an event should be marked as finalized. The threshold can be configured to ensure that if a certain fraction of clients sees or supports a reported kill, score increment, or other major milestone, that event is presumed genuine.

Once finalized, an event is inserted into a data structure that preserves a logical ordering. This might be a simple sequential log or a Directed Acyclic Graph (DAG) that accommodates parallel updates from different clients. Because a single aggregator (or small group of cooperating nodes) makes all ordering decisions, event finalization does not require multiple communication rounds among a large cluster. The aggregator's job is primarily to confirm authenticity, check for enough endorsements, and maintain a coherent timeline that it can publish to outside observers.

This system works smoothly when there is a reasonable level of trust that the aggregator itself will not censor incoming data or rewrite past events. The aggregator can still employ signatures and references from multiple clients to minimize the risk that any one client's forgery slips through undetected. If the aggregator is operated by a reputable entity, or if there are clear legal or contractual obligations preventing it from altering records, the lightweight approach can be more than sufficient. The result is a cost-effective mechanism for building an auditable record of in-game actions that supports a variety of reward or analytics applications.

8.2 Operational Considerations and Performance

The lightweight aggregator model is well suited to scenarios in which the number of participants per match is manageable. Each client's event submissions must be signed and delivered promptly to the aggregator. Because finalization depends on a majority or supermajority, the aggregator needs to ensure that it has reliable information about which clients are active, how many clients have endorsed a particular event, and whether partial coverage from certain participants can be handled without invalidating the entire match log.

Periodic anchoring to IPFS or a minimal chain is typically performed at intervals that make sense from a cost and latency perspective. A small match might finalize thousands of events in a single data block, which the aggregator then publishes at the end of the session. Larger matches or events with more critical timing demands can anchor data more frequently to reduce the window in which any tampering might go unnoticed. The frequency of these anchors is an operational choice: publishing them too often may increase fees, while publishing them too infrequently risks leaving valuable data in a pending state for a longer period.

While this approach does not provide the same resilience to malicious aggregator nodes as a fully decentralized byzantine-fault-tolerant system, it can still deter cheating if participants trust that there is minimal incentive for the aggregator to behave dishonestly. The aggregator's entire workflow, from verifying client signatures to calculating a final reference hash, can be made transparent through open-source implementations and by publicly disclosing the real-time flow of events. If any suspicious patterns arise—such as high numbers of single-client reports that lack corroboration—the aggregator can flag these for further review before finalizing them.

8.3 Summary of Benefits and Trade-Offs

The lightweight aggregator design provides an effective compromise for many gaming contexts. It is relatively simple to implement, requires limited infrastructure, and leverages existing decentralized storage or minimal blockchain tools to anchor a final, tamper-evident record. On the other hand, it assumes that the aggregator itself acts in good faith or at least has no strong reason to misrepresent the event data. It also depends on enough client endorsements to filter out most forgery attempts, while acknowledging that a more determined collusion between the aggregator and multiple clients could remain harder to detect.

Still, for the large number of multiplayer games and smaller competitive events that do not warrant a fully distributed blockchain-based solution, this lightweight approach can prove both practical and secure enough to ensure that token-based rewards are disbursed accurately. Organizers can rely on game-engine hooks to gather real-time data from participants, confirm it through a supermajority rule, and anchor the outcome in a record that no single party can secretly alter. Subsequent sections discuss how more robust solutions can be designed when the aggregator node itself must be distrusted, but for many use cases, the lightweight model strikes an optimal balance between simplicity and reliable verification.

9 Direct Blockchain-Based Ordering of Game Events

A more decentralized variant removes the lightweight aggregator and inserts events directly into a blockchain, relying on its consensus (e.g. BFT [5]) for ordering and finalization. Instead of sending signed events to an off-chain service that batches them, clients submit signed payloads to on-chain logic (a smart contract or native module). The chain provides global ordering and can enforce the same endorsement rules in its state.

Basic Workflow. In this design, each hooked event becomes an on-chain transaction or message. The client signs the payload (e.g., "client c claims a kill against player d at time T") and submits it to the blockchain's mempool. Block producers gather these pending transactions and confirm them in the next block. If the underlying chain provides 1-second block times and can handle upwards of 1000 transactions per second, real-time event flow becomes feasible, though each new block only finalizes events once per second.

Threshold Agreement On Chain. One major challenge is that many events require multiple signers or a supermajority threshold. One way to handle this is to model each event as a "proposal" that needs additional endorsements. When a kill event is first submitted, it enters a pending state in the contract's storage. Other clients who observed this kill then send "endorsement" transactions referencing that same event ID. Once the contract's state shows that enough unique signers have endorsed the event (meeting the threshold policy, such as 3 of 5 players in the match), the contract finalizes it on chain. If at least one block has elapsed (allowing time for endorsements), a new block can contain the finalization logic that marks the event as confirmed. The resulting ledger-based ordering is guaranteed by the chain's consensus, removing the need for an off-chain aggregator node to serialize events.

Data Structure in the Smart Contract. A contract can store events in a map from `eventID` to a record containing:

$$\{\ \texttt{payload, setOfSigners, requiredThreshold, status}\ \}$$

Each new kill or scoreboard update becomes a record with `status = pending` and an empty `setOfSigners`. When players endorse it, the contract appends their addresses to `setOfSigners` and checks whether `requiredThreshold` is met. If so, `status` transitions to `finalized`. Since each transaction is automatically placed in a specific block, the chain's block height (or timestamp) yields the event's final ordering relative to other events. Here `eventID` plays the same role as `eid` in the off-chain aggregator model.

Partial vs. Total Ordering[12]. Depending on the blockchain's execution model, events may arrive in arbitrary sequence. If two kills appear in the same block, the chain can either record them in the order of transaction IDs or as "simultaneous" within that block. This results in a partial order that only becomes strictly linear after the chain's consensus finalizes block boundaries. For games demanding a strict internal ordering (i.e. "kill #2 cannot finalize before kill #1"), each new event may reference the block or ID of the previous event. Alternatively, the contract could enforce a per-client localID so that kill $n + 1$ for a given client is only valid if kill n is already finalized on chain. We deliberately reuse the same (`clientID`, `counter`, `localID`) triple as in the off-chain aggregator so that sequencing semantics remain consistent across both deployment envelopes.

Incentives and Fees. Whereas the lightweight aggregator can batch events off-chain and anchor the result, a direct on-chain approach means each submission or endorsement is a separate transaction. This introduces the standard constraints of blockchain fees: if transaction costs are high, repeatedly submitting ephemeral data can become prohibitive. Some fast, high-throughput chains (with near-zero fees) may be better suited. Additionally, paying fees in the game's native token or awarding fee reimbursements to honest players could incentivize real-time participation.

Scalability and Network Load. Achieving 1000 transactions per second is within reach of certain modern blockchains but remains a non-trivial load. High-volume matches with many ephemeral events (e.g. health ticks) may overwhelm a chain unless off-chain filtering or aggregation is performed first. One practical approach is to rely on the chain for critical events (like kills or round transitions) while minor updates remain off-chain or are batched. The contract can store only the final aggregated counts, ensuring that the on-chain overhead stays manageable even under heavy gameplay.

Open Questions and Future Work. Much like the aggregator-based model, direct on-chain insertion still requires robust cross-verification logic. The threshold endorsement checks must be atomic, ensuring partial coverage does not stall every subsequent transaction. Furthermore, integrating advanced concurrency (e.g. partial vs. total ordering) within a smart contract can become complex if the chain's execution model is sequential. Future research may explore sidechains or layer-2 solutions that confirm game events at high speed and periodically anchor batches to a main chain, combining real-time performance with tamper-evident finality. Despite these open challenges, the direct on-chain approach, once feasible, would remove reliance on any single aggregator node and deliver a fully decentralized method for finalizing in-game actions.

10 Economic Model and Incentive Mechanisms

This section describes how the aggregator-based architecture integrates with a token-inflation system to reward honest gameplay and deter cheating. By staking collateral, participants expose themselves to financial risk if they attempt to submit falsified or contradictory data. Meanwhile, newly minted tokens are used to compensate players who produce legitimate, consensus-approved event logs. The design accommodates partial matches and hooking failures by withholding rewards from incomplete participants without penalizing them through slashing unless actual fraud is detected.

10.1 Token Inflation and Reward Distribution

The game ecosystem mints new tokens each period, denoted $\Lambda(t)$, from which a fraction θ is reserved for gameplay incentives. This allocation $\Gamma(t) = \theta\,\Lambda(t)$ is subdivided across the matches that occur in period t. Once a match's aggregator-based timeline is finalized, participants in that match become eligible for a share of the match's local reward pool $R_i(t)$. To sustain the incentive structure over time, the ratio θ and related parameters (such as the total inflation rate) can be tuned based on player populations or community preferences.

10.2 Collateral Staking for Cheating Deterrence

Players lock collateral $S_i(t)$ at the beginning of each match, reflecting a core principle of decentralized finance. By putting up a stake, each participant signals confidence in the authenticity of their submitted gameplay events, knowing that proven fraud could lead to forfeiture. The probability p of detecting dishonest actions, combined with the maximum illicit gain r, guides the rational calculus:

$$U_{\text{cheat}} = (1 - p)\, r \; - \; p\, S_i(t).$$

Cheating is unprofitable if $(1 - p)\, r \leq p\, S_i(t)$. The aggregator nodes use cross-verification (comparing partial orders from multiple clients) and other detection mechanisms to identify suspicious or contradictory records. If a participant is conclusively found to have forged data, a slashing penalty is applied. However, no participant is slashed simply for hooking failures or incomplete logs caused by external factors. Incomplete or late-joining clients instead forfeit the chance to earn rewards but do not lose their stake.

10.3 Example Calculation of Cheating Profitability

The interplay of inflation, stake collateral, and detection can be illustrated with a hypothetical scenario. Assume $\Lambda(t) = 10{,}000$ tokens per period, $\theta = 0.5$, and $N(t) = 100$ matches. Each match thus has $R_i(t) = \frac{5000}{100} = 50$ tokens in its local pool. With a collateral ratio $\rho = 0.2$, each participant's stake is $S_i(t) = 0.2 \times 50 = 10$. If $p = 0.40$ and r denotes the potential gain from cheating, the expected utility of cheating is $U_{\text{cheat}} = 0.60\, r - 4$, which is negative for $r \leq 6.67$. Under these parameters, minor fraud attempts yield less than the risked stake, discouraging cheating. If hooking failures cause partial logs or early disconnections, participants labeled as incomplete simply do not earn from this match, avoiding a slash unless an audit reveals active fraud.

10.4 Reward Distribution and Slashing

This procedure completes the economic loop by assigning newly minted tokens to honest participants, returning stake collateral to those who disconnected or arrived late, and penalizing any confirmed forgers. It relies on the aggregator's trusted timeline to classify each participant as *complete*, *incomplete*, or *lateJoiner*, and to identify suspected cheaters. Once the match ends, the aggregator parses its final event log to determine which players genuinely contributed valid hooking data and which failed to meet the required coverage or were caught forging events.

Participants flagged as *suspects* face a final audit to confirm whether they indeed submitted contradictory or fraudulent records. If cheating is proven, their collateral stake is slashed—destroyed or redistributed according to policy—and they are disqualified from any rewards. Those who are merely incomplete or late-joining receive their stake back but no additional tokens. Only fully compliant players, whose event data was consistent and complete, receive token payouts from the match's local reward pool. By referencing the final aggregator timeline (or its linearization) for kills, objectives, or other achievements, this algorithm calculates each qualifying player's share of newly minted tokens via ScoreToShare, which can be implemented as a function of confirmed kills, objectives, or other per-title metrics with caps to prevent farming. The resulting payout is then added to their returned stake. Once distribution is complete, the aggregator produces a transparent summary indicating which participants were rewarded, who was slashed, and which stakes were simply returned without reward.

Algorithm 6 Reward Distribution and Slashing

```
 1: Input: Final aggregator timeline $T_{final}$ for match $i$, local reward pool $R_i(t)$, collateral stake $S_i(t)$ for each participant, detection
    results.
 2: Output: Payouts to honest participants, refunds to incomplete players, slashing for confirmed fraud.
 3: Parse $T_{final}$ to identify participants $P_i$ and status: $\{complete, incomplete, lateJoiner\}$. Retrieve flagged suspects.
 4: for all $p \in P_i$ do
 5:     if $p \in$ suspects and AuditConfirmsFraud($p$) then
 6:         Slash $S_i(t)$; mark $p$ as disqualified.
 7:     else if $p.status \in \{incomplete, lateJoiner\}$ then
 8:         Refund $S_i(t)$; no rewards.
 9:     else
10:         share ← ScoreToShare($T_{final}$, $p$)
11:         Payout($p$, $S_i(t) + \text{share} \cdot R_i(t)$)
12:     end if
13: end for
14: return DistributionReport()
```

11 Event Validation

11.1 Persistent vs. Ephemeral Game Data

Game data captured by hooking interfaces can be broadly categorized into two distinct types: ephemeral events and persistent data fields. Ephemeral events, such as a kill notification or a round transition, are generated as one-time callbacks that capture specific, discrete moments during gameplay. Once these events occur, they are delivered immediately and, if not captured at that instant, are lost forever.

In contrast, persistent data fields represent continuously updated information, such as a player's current health, score, or ranking. Unlike ephemeral events, persistent data remains available over the duration of a match and is updated only when changes occur. This ongoing stream of data provides a stable context that can be used to verify or supplement the transient nature of ephemeral events.

11.2 Validation of Aggregated Event Counts Against the Scoreboard

In our system, the aggregator collects real-time event data from client-side hooking interfaces. Since the persistent scoreboard serves as the authoritative record of official event counts, it is essential to validate that the aggregated data does not exceed these official counts. To reduce false positives due to scoreboard lag, we apply a small tolerance window in both value and time and require multiple independent snapshots.

Algorithm 7 Validate Reported Event Counts Against Scoreboard with Grace Windows

```
procedure ValidateEventCounts($\mathcal{P}$, $\mathcal{E}$, $EventCountAgg$, $ScoreboardSnapshots$, $\delta$, $\Delta t$, $q$)
    for all player $p \in \mathcal{P}$ do
        for all event type $e \in \mathcal{E}$ do
            $(sb_1, ts_1, S_1) \leftarrow$ most recent consensus snapshot for $(p, e)$ with $|S_1| \geq q$    ▷ $S_1$ is the set of reporting clients
            $(sb_0, ts_0, S_0) \leftarrow$ previous consensus snapshot within $[ts_1 - \Delta t, ts_1]$ with $|S_0| \geq q$, or $\perp$ if none exists
            $aggCount \leftarrow EventCountAgg[p][e]$
            if $sb_0 \neq \perp$ then
                if $aggCount > sb_1 + \delta$ and $aggCount > sb_0 + \delta$ then
                    FlagIneligible($p, e$); Log potential inflation.
                end if
            else
                if $aggCount > sb_1 + \delta$ then
                    FlagIneligible($p, e$); Log potential inflation.
                end if
            end if
        end for
    end for
end procedure
```

11.3 Detection and Handling of Non-participating Clients

Algorithm 8 Detecting, Removing, and Notifying Non-participating Clients

```
procedure RemoveAndNotifyNonParticipants($\mathcal{C}$, $MissingCount$, $missingThreshold$)
    for all client $c \in \mathcal{C}$ do
        if MissingCount[$c$] $\geq$ missingThreshold then
            MarkNonParticipant($c$); NotifyClient($c$, “Removed due to insufficient reporting.”)
        end if
    end for
end procedure
```

12 Evaluation Plan

Measure hook coverage of critical events across at least two titles using sanctioned feeds. Report end-to-end latency (client capture to finalization) under uplink constraints and with induced failures (late joins, overlay restarts). Quantify false-missing and false-positive rates against persistent scoreboard snapshots with the grace windows in Algorithm 7. Compare finalization latency and operator cost between the lightweight aggregator and an on-chain endorsement prototype. Concretely, candidate scenarios include 10–20 player FPS matches (e.g. CS2 via GSI, LoL via Live Client) at typical home uplink bandwidths, with metrics such as median and 95th-percentile capture-to-finalization latency, per-event-type coverage (kills, round transitions), and sensitivity of scoreboard-inflation detection under controlled injection of synthetic fake events. For the on-chain prototype, measure achievable throughput and block times on a fast chain while restricting on-chain events to critical milestones (e.g. kills and round transitions).

13 Related Work

Client-facing event frameworks such as Overwolf provide curated, publisher-sanctioned signals for several titles [1], and Steamworks offers first-party overlay services for developer-owned games [2]. Binary interception via Microsoft Detours demonstrates feasibility on Windows but conflicts with many anti-cheat regimes [3]. For ordering, classical PBFT [5] motivates the lightweight versus fully decentralized design; DAG-ledger surveys [8] frame partial orders; attribute-based encryption provides expressive access control for off-chain archives [7]. We build on these strands by unifying sanctioned client event capture, cross-endorsement, DAG-based sequencing, and CP-ABE–style archival controls within a single implementation reference.

14 Architecture Figure

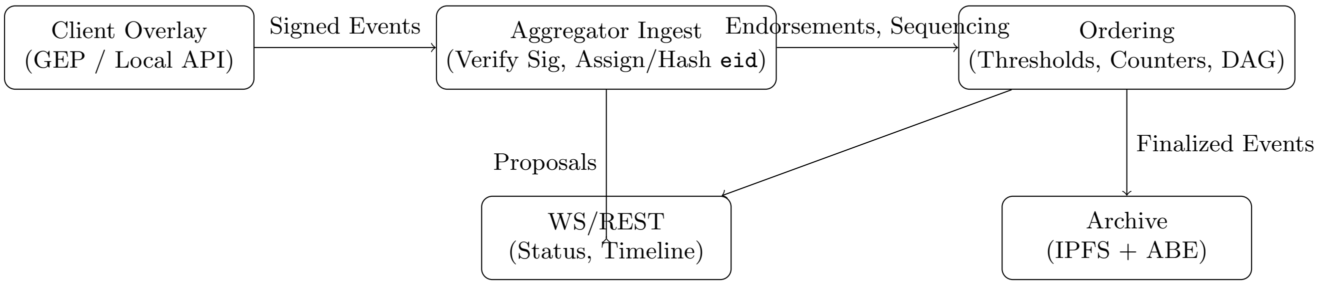

Figure 1: Uppercut data path and external surfaces.

15 Conclusion and Future Directions

This work describes a framework for capturing, validating, and distributing in-game events through client-side hooks, without relying on private server logs. By combining ephemeral callbacks and persistent fields on the client with signed event streams, a single aggregator (or an on-chain contract) can build a verifiable match timeline on top of existing titles.

Two deployment models cover most needs. The lightweight aggregator uses a supermajority rule and periodic anchoring to decentralized storage and is suitable when the operator is mostly trusted and infrastructure should stay simple. The blockchain-based variant pushes ordering and endorsement logic into a consensus layer, at the cost of higher network load and integration complexity.

A recurring concern is partial coverage and missing hooks. The proposed aggregation logic cross-checks multiple clients and compares counts against persistent scoreboards, flagging discrepancies such as reported kills that exceed official tallies. An economic layer with collateral and slashing provides an additional lever against outright fraud: honest clients recover their stake and share newly minted rewards, while proven forgers can lose theirs.

For long-term storage and controlled access, we outline how ABE (and ZK-DABE when needed) can protect archived match logs, letting referees and analytics partners read data that competitors cannot. This keeps sensitive logs off-chain while preserving a small, auditable footprint on chain.

Future work includes zero-knowledge proofs for derived predicates over event streams, reputation-aware staking, shared schemas that span multiple games, and higher-rate ingestion paths that use layer-2 systems with periodic anchoring. The aim is to keep the client-facing surface simple while making event verification and reward logic incrementally stronger over time.

A recurring theme throughout this paper is *managing partial coverage and unavailability of hooks*. Even the most advanced hooking frameworks can fail mid-match or omit key events, leading to incomplete timelines. Our proposed aggregator logic correlates overlapping data from multiple clients and cross-references it with persistent scoreboard fields. By detecting discrepancies—such as a reported kill count surpassing the scoreboard total—the system identifies potential forgeries and flags those participants. Furthermore, a collateral-based economic model incentivizes honesty: players stake tokens that may be slashed upon proven fraud, while those who provide complete, consistent data share in newly minted token rewards.

In addition to preserving data integrity, we also discussed off-chain and on-chain storage solutions. Attribute-Based Encryption (ABE), possibly paired with zero-knowledge enhancements (ZK-DABE), can selectively grant or deny decryption capabilities for archived match logs, ensuring confidentiality where needed (e.g. disallowing *competitors* from viewing sensitive live feeds) while still preserving transparency for referees and analytics partners.

Potential future directions include zero-knowledge attestations of event predicates, adaptive reputation-aware staking, standardized cross-game schemas, and high-rate ingestion with layer-2 anchoring.

A Indicative Per-Title Surfaces and Identity Mapping

Title	**Local Surface**	**Identity Binding**	**Core Event Families (examples)**	**Policy Gate Default**
League of Legends	Live Client Data API (local HTTP)	PUUID / Summoner (local)	Scoreboard snapshots, objective events (Dragon/Baron), limited kill feed, match state	Events allowed; in-play commercial overlay per title policy
CS:GO / CS2	Game State Integration (GSI)	SteamID64	`round_start`/`round_end`, kill, bomb plant/defuse (CS:GO), scoreboard	Events allowed; overlays per title policy
Dota 2	Game State Integration (GSI)	SteamID64	Kills, tower/objective updates, scoreboard	Events allowed
Fortnite	Overwolf GEP (where available)	Title-dependent	Eliminations, placements, match state	Events allowed; no in-play commercial overlay if disallowed
Overwatch 2	Overwolf GEP (where available)	Title-dependent	Eliminations, objective progress, match state	Same as above
PUBG: Battlegrounds	Overwolf GEP (where available)	Title-dependent	Kills/knockdowns, placements, match state	Same as above
Rocket League	Overwolf GEP (where available)	Title-dependent	Goals, assists, match start/end	Same as above

Table 1: Indicative, publisher-sanctioned local surfaces and identity bindings used by the system. Coverage and availability vary by title and over time. The runtime policy gate enforces which surfaces are enabled per title.

Note: This appendix is an implementation aid. Always verify current title policies and Overwolf/first-party surface status at runtime.

Copyright and Licensing Statement

References

[1] Overwolf Developers. Overwolf: A platform for in-game apps. Overwolf SDK Documentation, 2024. Available at: https://overwolf.github.io/

[2] Valve Corporation. Steamworks API: Game Overlay. Steamworks Developer Documentation, 2024. Available at: https://partner.steamgames.com/doc/sdk

[3] Hunt, G., and Brubacher, D. Detours: Binary interception of Win32 functions. Microsoft Research, 1999. Available at: https://www.microsoft.com/en-us/research/project/detours/

[4] Ethereum Foundation. ERC-1155: Multi-Token Standard for In-Game Assets. Ethereum Developer Docs, 2024. Available at: https://ethereum.org/en/developers/docs/standards/tokens/erc-1155/

[5] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance. Laboratory for Computer Science, MIT. OSDI, 1999. Available at: https://pmg.csail.mit.edu/papers/osdi99.pdf

[6] Benet, J. IPFS - Content Addressed, Versioned, P2P File System. IPFS Technical Whitepaper, 2014. Available at: `https://ipfs.tech/`

[7] Herranz, J. Attribute-based encryption implies identity-based encryption. IET Information Security, 2017. Available at: `https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-ifs.2016.0490`

[8] Directed Acyclic Graph-based Distributed Ledgers – An Evolutionary Perspective. International Journal of Engineering and Advanced Technology, 9(1), May 2020. DOI:10.35940/ijeat.A1970.109119.

[9] Josefsson, S., and Ladd, I. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, IETF, 2017. Available at: `https://datatracker.ietf.org/doc/html/rfc8032`

[10] Krawczyk, H., and Eronen, P. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, IETF, 2010. Available at: `https://datatracker.ietf.org/doc/html/rfc5869`

[11] Şahin, F. Play to Earn Web 3.0: The Future of Gaming and Marketing. Proceedings of TIEM 2023. Available at: `https://www.researchgate.net/publication/376985610_Play_to_Earn_Web_30_The_Future_of_Gaming_and_Marketing`

[12] Almeida, P. S. A Framework for Consistency Models in Distributed Systems. arXiv preprint arXiv:2411.16355, 2024. Available at: `https://arxiv.org/abs/2411.16355`