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

Johan Bratt

February 21, 2025

Abstract

The rapid evolution of competitive online gaming and the emergence of token-based reward systems have exposed a critical challenge: the inability to reliably access real-time, tamper-evident event data from popular game servers. This paper addresses the issue by proposing a server-agnostic framework that captures essential in-game events directly on the client side using advanced hooking frameworks. Our approach aims to overcome the limitations imposed by restricted or concealed official server logs, thereby enabling third-party systems to validate game events without relying on privileged server access. The end goal is to create a robust, transparent mechanism that not only facilitates fair play and accurate match reconstruction but also underpins the secure distribution of token rewards to players. To achieve this, we present two consensus models—a streamlined supermajority aggregator for low-cost, scalable deployments, and a Byzantine-fault-tolerant solution for high-stakes environments—both designed to support partial data coverage and detect dishonest submissions. This work lays the foundation for integrating secure, real-time event verification with innovative monetization strategies in modern gaming ecosystems.

1 Introduction and Motivation

Online gaming has rapidly expanded beyond casual entertainment to include competitive events, influencer-driven tournaments, betting markets, and even token-based reward programs. A growing number of platforms wish to convert in-game achievements or milestones into cryptocurrency rewards[4], whether by staking mechanisms or broader 'play-to-earn"[10] models. The challenge is that most popular titles either keep their official server logs private or require developers to own or manage the underlying server in order to retrieve reliable event data. This restriction forces many third-party projects to operate custom servers if they want to confirm kills, round outcomes, or other actions in real time. While effective in principle, the need for a dedicated server infrastructure becomes a strong barrier to adoption if players prefer the official or best-populated servers. Once the community reverts to those official environments, a custom server's data—and any associated play-to-earn features—can become irrelevant.

These issues illustrate the broader need for a system that is both server-agnostic and tamper-evident. The core premise here is that we can circumvent private or unavailable server logs by capturing essential game events on the client side. Such an approach allows matches played on any server to feed into an external aggregator that verifies each event's authenticity and consolidates the data into a globally consistent record. Whether the ultimate goal is distributing token rewards, offering betting markets, or providing community-level analytics, the ability to monitor official servers without relying on official APIs or server logs is what makes this design broadly scalable and appealing. As long as the client can intercept the necessary gameplay callbacks, the system

can build a reliable timeline for calculating rewards or detecting anomalies, regardless of the server in use.

1.1 Integration with Existing Gaming Ecosystems: A Business Perspective

Our framework is designed not only to deliver technical robustness but also to seamlessly integrate into the modern gaming ecosystem. It operates unobtrusively in the background, eliminating the need for invasive screen overlays—though optional overlays can be provided if desired. This ensures that players enjoy a native experience without disruption, while event data is captured transparently for real-time analytics and reward distribution.

In today's gaming landscape, streamers and content creators play a crucial role in driving community engagement and brand awareness. By integrating our solution into games, organizers and publishers can tap into the free marketing provided by these influential figures, as the system's real-time, tamper-evident logs enhance the credibility of competitive events and token-based reward schemes. This organic reach helps elevate the game's profile without additional marketing expenditure.

Additionally, our architecture supports flexible monetization strategies. For example, players may opt to run in-game advertisements while playing, with the promise of increased token rewards in exchange for their participation. This creates a mutually beneficial ecosystem where advertisers gain access to a highly engaged audience, and players receive augmented incentives—all without compromising the core gameplay experience.

Overall, the system is tailored to align with the evolving dynamics of the gaming industry. By merging non-intrusive, background data capture with opportunities for innovative monetization and leveraging the power of streamers, our solution positions itself as a strategic asset for publishers, tournament organizers, and platform partners looking to drive engagement and build trust in competitive gaming.

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]. By subscribing to these hooks, a lightweight application on the player's computer collects cryptographically signable[9] event records as they occur. Unlike server-specific solutions, such a client-level capture does not require privileged server access; instead, it translates engine-confirmed actions into verifiable data streams. Each event is signed locally and passed to an external aggregator, which can reconcile input from multiple participants to detect discrepancies or validate consensus before finalizing the match log.

Despite its advantages, this model must address the reality of partial coverage. Certain event categories may be unavailable if the hooking API is incomplete or if a patch temporarily disables key callbacks. Additionally, players might enable the overlay late or lose hooking functionality mid-match, causing ephemeral events to be missed. An aggregator that compares data from multiple participants can still achieve a broad view of the session. When several players consistently confirm a kill or objective, that event is marked authentic; when only a single reporter claims an occurrence, the aggregator may flag it for additional scrutiny or discard it if no corroboration emerges.

To finalize a match record, the aggregator monitors these streams until all participants have either ended the session or gone silent for a specified interval. At that point, it logs the data as complete or partial, ensuring that honest players are credited for their legitimate events while those with incomplete data do not receive full benefits. If the overarching system uses cryptocurrency

rewards or staking incentives, this aggregator-level record provides the anchor for payouts or penalty enforcement. In a play-to-earn context, tokens can be minted or unlocked based on the confirmed game events, allowing players on popular or official servers to earn rewards just as effectively as they would on a custom server. The difference is that they retain their preferred gameplay environment while still benefiting from transparent, tamper-evident match logs.

Because the hooking approach only needs client authorization, it stays resilient against closed-source server architectures. By freeing tournament organizers, play-to-earn platforms, and community projects from the requirement of hosting their own servers, this system avoids splintering the player base. Instead, anyone who installs and runs the hooking overlay can contribute verifiable data to the aggregator, and the aggregator's cross-verification ensures that no single perspective or partial coverage gap undermines confidence. This architecture thereby unifies a server-agnostic methodology with a robust foundation for fair cryptocurrency distribution and match analytics.

3 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.

3.1 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.

3.2 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.

3.3 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.

4 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.

4.1 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.

4.2 Cryptographic Checks

To combat tampering during transmission, each client signs its reported events with a private key. The aggregator verifies these signatures upon receipt, discarding any submissions with invalid signatures or altered payloads. This mechanism ensures the final match record remains verifiably authentic, as each event can be rechecked or anchored to a public ledger. If you wish to include more details on live data transmission or batching, you may integrate them here.

4.3 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.

4.4 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.

4.5 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.

4.6 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. To switch from local IDs to aggregator-assigned IDs, one need only remove the incrementing step in the client's `LocalEventCapture` logic and replace the aggregator's local ID checks in `AggregatorThresholdIDSync` with a routine that either issues or retrieves the global event identifier. Once assigned, that global ID forms the basis of any finalization or threshold checks, allowing the aggregator to unify partial perspectives under a single, consistent reference for each event.

5 Detailed Algorithmic Workflow

This section showcases the core algorithms of the proposed event capture and aggregation framework. Each algorithm is accompanied by line-by-line explanations in paragraph form, clarifying the reasoning behind each instruction. The aim is to illustrate how the system maintains consistency, validates authenticity, and synchronizes event data in a realistic gaming environment.

5.1 Client-Side Hook Capture

This algorithm runs on each player's machine to intercept, label, and transmit real-time events via a hooking framework such as Overwolf. First, the client checks hook availability (for event types like kills or rounds) and registers for those callbacks. Optionally, it captures a snapshot of persistent data (health, rank, etc.) if that feature is supported. Next, upon detecting a `matchStart`, the client looks up a configuration for its `game_id` to decide which event types increment which local counters and whether local ID assignment is done on the client or deferred to the aggregator.

During the main capture loop, every new raw event from the hooking API is processed according to its `type`. If the aggregator is *not* responsible for assigning IDs, the client increments the relevant local counter (for example, a `killCounter` or `roundCounter`) and attaches the resulting local ID to the event. If the aggregator is in charge, the client sets `localID` to zero (or `null`), indicating that ID assignment will occur upstream. The client then signs the event with its private key, timestamping and packaging it for the aggregator. If needed, a "skip-forward" signal from the aggregator prompts the client to adjust or discard a range of local IDs that are deemed invalid or out-of-sync. Once the match ends or hooking fails, the client ceases capturing and returns whatever data it has collected. By separating different event types into multiple counters, this procedure avoids conflating high-frequency minor updates (like health changes) with more important events (like kills), preventing large jumps and partial-coverage confusion.

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

```
1: procedure LocalEventCapture((pk, sk), aggregatorEndpoint = A, gameID)
2:     /* Step 1: Hook Version Support */
3:     for hook in {match, round, kill, ...} do
4:         if hook is unavailable then
5:             report "HOOK VERSION ERROR"; return
6:         end if
7:     end for
8:     report "HOOK VERSION OK"
9:     /* Step 2: Register Hooks */
10:    if registration fails then
11:        report "HOOK REGISTRATION ERROR"; return
12:    else
13:        report "HOOK REGISTRATION OK"
14:    end if
15:    /* Step 3: Optional Snapshot */
16:    if getInfo is supported then
17:        retrieve persistent fields; sign(snapshot, sk); report "SNAPSHOT OK"
18:    else
19:        report "SNAPSHOT ERROR"                                          ▷ non-fatal
20:    end if
21:    /* Step 4: Match Start Check */
22:    if matchStart not confirmed then
23:        return "MATCH START ERROR"
24:    else                                                                 ▷ New or updated lines:
25:        localCounters ← {counterName ↦ 0}                                ▷ e.g., killCounter=0, roundCounter=0, etc.
26:        aggAssignedMode ← fetchConfig(gameID).assignIDs                  ▷ if true, aggregator assigns event IDs
27:        report "MATCH START OK"
28:    end if
29:    /* Continue in Part 2 */
30: end procedure
```

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

procedure LocalEventCapture((pk, sk), aggregatorEndpoint = A, gameID)
  /* Step 5: Main Capture Loop */
  $\mathcal{Q} \leftarrow \varnothing$
  while match is ongoing **do**
    $rawEvent \leftarrow$ next engine hook event
    $type \leftarrow rawEvent.type$ ▷ e.g. "kill", "health", etc.
    if not aggAssignedMode **then**
      $cName \leftarrow$ `mapToCounter`$(type, gameID)$
      localCounters$[cName]$ += 1
      $E.localID \leftarrow$ localCounters$[cName]$
    else
      $E.localID \leftarrow 0$ ▷ Aggregator will assign an ID
    end if
    $E.eventType \leftarrow type$
    $E.gameID \leftarrow gameID$
    $E.payload \leftarrow rawEvent.payload$
    $E.timestamp \leftarrow$ localTime()
    $E.signature \leftarrow \text{Sign}(E, sk)$
    $\mathcal{Q} \cup \{E\}$
    send(E, A)
    if aggregator skip-forward **then**
      adjust or reset localCounters per aggregator's instruction
    end if
    if `matchEnd` confirmed **then**
      report "MATCH END OK"; cleanup; **break**
    end if
  end while
  /* Step 6: Cleanup */
  if `matchEnd` not confirmed **then**
    report "MATCH END ERROR"
  end if
  return $\mathcal{Q}$
end procedure

5.2 Aggregator Threshold Logic

On the aggregator side, each event is examined to determine whether it is valid, which `game_id` and event type it belongs to, and what policy applies (for instance, a required threshold of signers or a strict local ID sequence). If the configuration says that the aggregator assigns IDs, the incoming `localID` from the client is replaced with an aggregator-issued number (e.g., an incrementing global counter). Otherwise, the aggregator respects the client's counter-based local ID.

The aggregator then checks whether partial or strict ordering rules are in effect for the relevant event type. In strict mode, an event at local ID $n + 1$ can only finalize after local ID n has been confirmed or explicitly skipped. In partial mode, local ID $n+1$ can complete even if n remains stuck in a pending state. Once ordering constraints are satisfied, the aggregator logs the event in a shared trackerMap, incrementing the set of signers for that $\{client, counterName, localID\}$ key. When the number of unique signers meets or exceeds the policy-defined threshold, the aggregator deems the event final and updates MaxConfirmedID$[client][counterName]$ accordingly. This per-event finalization approach supports concurrency among different event categories (like kills versus round transitions) and allows partial coverage if the game rules permit it. Finally, the aggregator merges confirmed events into a DAG, which may be topologically sorted later to produce a total order if desired. If hooking data ceases or the match terminates, any unfinalized events are simply left incomplete, and the final aggregator record captures whatever endorsements have been validated so far.

Algorithm 3 Aggregator Threshold Logic

```
1: procedure AggregatorThresholdIDSync(C, policyDB)
2:     for all c ∈ C do
3:         MaxConfirmedID[c] ← empty map
4:     end for
5:     trackerMap ← ∅
6:     while aggregator is running do
7:         e ← getNextEventRecord()
8:         if e.clientID ∉ C  or  not VerifySig(e) then
9:             report "INVALID EVENT"; discard(e); continue
10:        end if
11:        c ← e.clientID
12:        policy ← policyDB[e.gameID][e.eventType]
13:        if policy.aggregatorAssignedIDs then
14:            assignedID ← assignGlobalID()
15:            e.localID ← assignedID
16:        end if
17:        counterName ← policy.counterName
18:        if policy.seqMode = "strict" then
19:            if e.localID ≠ MaxConfirmedID[c][counterName] + 1 then
20:                report "STRICT ORDER VIOLATION for client" c
21:                discard(e); continue
22:            end if
23:        else
24:            if e.localID ≤ MaxConfirmedID[c][counterName] then
25:                report "REPLAY/OLD EVENT (partial mode)"; discard(e); continue
26:            end if
27:        end if
28:        if trackerMap[(c, counterName, e.localID)] is undefined then
29:            trackerMap[(c, counterName, e.localID)] ← {signers : ∅, payload : e.payload, timestamp : e.timestamp}
30:        end if
31:        trackerMap[(c, counterName, e.localID)].signers += c
32:        threshold ← policy.threshold
33:        if |trackerMap[(c, counterName, e.localID)].signers| ≥ threshold then
34:            report "EVENT FINALIZED (counter="counterName")"
35:            MaxConfirmedID[c][counterName] ← max(MaxConfirmedID[c][counterName], e.localID)
36:            call BuildAndOrderDAG(...)
37:        end if
38:        call skipForwardIfNeeded(c, counterName, trackerMap, MIN_WAIT_TIME, SKIP_GAP)
39:    end while
40: end procedure
```

5.3 Hybrid Skip-Forward Mechanism

To further enhance concurrency while ensuring the integrity of the event sequence, we introduce a hybrid skip-forward mechanism. This mechanism combines a minimum waiting period with a check on how far ahead future events have advanced. The timer used for the waiting period can be implemented as a wall clock timer or as a block count. In our blockchain version, if a new block is produced every 1 second, the timer effectively reflects a block count.

The routine `skipForwardIfNeeded` (Algorithm 4) is invoked from the main aggregator logic to decide whether to finalize an unconfirmed event as "skipped" (or marked as incomplete) when it is holding up the sequence.

Algorithm 4 Hybrid Skip-Forward Mechanism

procedure SKIPFORWARDIFNEEDED(client c, counterName, trackerMap, `MIN_WAIT_TIME`, `SKIP_GAP`)
  $currentID \leftarrow$ MaxConfirmedID[c][counterName] + 1
  while trackerMap contains an event for (c, counterName, $currentID$) **do**
    $e \leftarrow$ event with localID = $currentID$ from trackerMap
    if thresholdReached(e) **then**
      finalizeEvent(e)
      MaxConfirmedID[c][counterName] $\leftarrow currentID$
      $currentID \leftarrow currentID + 1$
    else if ($currentTime() - e$.timestamp $\geq$ `MIN_WAIT_TIME`) **and** (getHighestLocalID(c, counterName) $\geq currentID +$ `SKIP_GAP`) **then**
      markEventAsSkipped(e)
      log("Skipped event for client " c " at localID=" $currentID$)
      $currentID \leftarrow currentID + 1$
    else
      break ▷ Wait for more endorsements or future events.
    end if
  end while
end procedure

5.4 DAG Construction for Confirmed Events

Once the aggregator deems certain events "confirmed" (for example, because they meet a threshold requirement or follow strict ordering rules), those events are woven into a coherent Directed Acyclic Graph (DAG)[8]. Each finalized (c, `localID`) pair becomes a node in the DAG, along with metadata such as client identifiers and payload details. Edges within the DAG reflect both the per-client sequential constraints—where a localID of 1 for a particular client precedes that same client's localID of 2—and any broader global dependencies, like "`matchStart` must occur before any kills."

After creating nodes for all confirmed events, the aggregator adds edges for per-client ordering. If client c has confirmed local IDs $\{1, 2, 3, \dots\}$, each consecutive pair forms a directed edge ($1 \rightarrow 2 \rightarrow 3$). This enforces a strict progression for that single client's events. It can then apply any game-specific or global constraints, ensuring, for instance, that a round transition node precedes kill nodes if that is part of the game's logic. Where multiple clients report the same real-world occurrence from different perspectives, nodes representing this occurrence can be merged or annotated with multiple signers, preventing redundancies in the final DAG. Finally, if the system demands a linear ordering (for example, for replays or tie-breaking), a topological sort on this DAG yields a single sequence respecting all directed edges. If only a partial order is needed, the DAG by itself suffices, clearly showing concurrency and logical constraints without artificially imposing a total order on unrelated events. This structure supports reward computations by clarifying precisely how events relate and which must come before or after one another.

Algorithm 5 DAG Construction for Confirmed Events

```
procedure BuildAndOrderDAG(trackerMap)
    DAG ← empty data structure.
    /* STEP 1: Construct DAG Nodes from confirmed events */
    for each key (c, lid) in trackerMap where the event is confirmed do
        v ← createNode(client: c, localID: lid, payload: trackerMap[(c, lid)].payload).
        Add v to DAG.nodes.
    end for
    /* STEP 2: Add Edges for Per-Client LocalID Ordering */
    for each client c in C do
        let L_c ← sorted list of confirmed localIDs for c.
        for i = 1 to |L_c| − 1 do
            addEdge( node(c, L_c[i]) → node(c, L_c[i + 1]) ).
        end for
    end for
    /* STEP 3: Add Edges for Global Logical Dependencies */
    for each known logical rule (e.g., "matchStart must precede kill") do
        identify affected nodes and add corresponding edges.
    end for
    /* STEP 4: (Optional) Merge Nodes for Identical Events */
    for each set of nodes representing the same logical event (e.g., same kill) do
        merge nodes or annotate them with multiple signers.
        adjust edges accordingly.
    end for
    /* STEP 5: Obtain a Total Ordering via Topological Sort */
    linearOrder ← KahnsSortWithPriority(DAG).
    return linearOrder.
end procedure
```

6 Storing Game Events in Decentralized Storage Using ABE (Example: ZK-DABE)

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. 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.

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.

7 Lightweight Live Event Aggregation - Main Proposal

A lightweight aggregator orders incoming streams of game events using a comparatively simple verification mechanism, typically based on a supermajority of participants. This design refrains from employing a fully fledged Byzantine fault-tolerant protocol among multiple aggregator nodes, relying instead on fewer assumptions about the network or a small cluster of relatively trusted nodes. It caters to situations in which the cost or operational overhead of a blockchain solution is not justified. By consolidating the signed data from game clients in a single place, the aggregator can swiftly finalize most events, anchor them in a tamper-evident repository, and allow external parties to verify the completeness and correctness of the resulting record.

A common use case for this design arises in gaming communities that wish to offer token rewards for achievements but do not have the resources—or see no need—to manage a more complex decentralized network. As long as the aggregator provides a transparent record-keeping process, interested parties can review the final timeline, confirm that sufficient endorsements were given to each event, and detect any suspicious alterations. This simpler approach focuses on providing a robust "first line of defense" against cheating or manipulation without incurring the considerable technical and logistical demands of a fully decentralized consensus system.

7.1 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.

7.2 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.

7.3 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.

8 Direct Blockchain-Based Ordering of Game Events

A more decentralized variant of this system circumvents the "lightweight aggregator" by inserting each event directly into a blockchain, relying on its consensus, such as BFT [5], for both ordering and finalization. Rather than sending signed events to an off-chain aggregator that sorts and batches them into blocks afterward, clients transmit their signed events to on-chain logic (a smart contract or native blockchain module) in near real time. The chain itself then ensures global ordering and detects whether sufficient endorsements exist for each event.

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.

Partial vs. Total Ordering[11]. 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.

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.

9 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.

9.1 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.

9.2 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 $(1-p)\,r \le p\,S_i(t)$. 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.

9.3 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 $r \le 6.67$. 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.

9.4 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 for kills, objectives, or other

achievements, this algorithm calculates each qualifying player's share of newly minted tokens and appends it to their stake upon return. 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

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 from aggregator logs.
Output: Payouts to honest participants, refunds or partial refunds to incomplete players, slashing for confirmed fraud.
Parse T_{final} to identify the set of participants P_i and each participant's status: {*complete*, *incomplete*, *lateJoiner*}. Also retrieve any flagged `suspects` from detection logic.
for each $p \in P_i$ **do**
    if p is in `suspects` and aggregator audit confirms forged events **then**
        Slash entire stake $S_i(t)$ from p. Optionally redistribute or burn these tokens.
        Mark p as *disqualified*; no further rewards granted.
    else if $p.status = incomplete$ **or** *lateJoiner* **then**
        Return stake $S_i(t)$ to p in full (no slash, but no reward either).
    else
        Compute p's share of $R_i(t)$ based on final scoring from T_{final} (kills, objectives, etc.).
        Return stake $S_i(t)$ plus p's allocated reward.
    end if
end for
return distribution outcome for match i, including a list of disqualified or slashed participants.

Discussion and Implications. This unified procedure ensures that no participant is penalized (beyond forfeiting potential rewards) simply because hooking was partially unavailable or the session ended prematurely. Honest players who remain until match completion with consistent logs receive both their stake and a portion of new tokens. Conversely, proven cheaters lose their stake entirely, and partial participants neither gain nor lose tokens. By mapping aggregator-level detection outcomes directly to stake-based economics, the system harnesses the aggregator's fault-tolerant data pipeline to create a self-regulating environment where players have strong financial reasons to report only accurate gameplay actions.

10 Event Validation

10.1 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.

By distinguishing between these two types of game data, developers can design systems that effectively capture both momentary gameplay actions and continuously evolving game states. This differentiation is essential for constructing a comprehensive event log, ensuring that the overall progression of a match can be reconstructed even if some ephemeral events are missed.

10.2 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. Discrepancies—such as a player's aggregated count being higher than the scoreboard's value—are treated as illegal records. When such discrepancies are detected, the player is flagged and marked as ineligible for rewards, ensuring that the reward distribution process remains fair and free from fraudulent data manipulation.

Algorithm 7 Validate Reported Event Counts for Multiple Event Types Against Scoreboard

procedure ValidateEventCounts($\mathcal{P}$, $\mathcal{E}$, *EventCountAgg*, *ScoreboardData*)
  for all player $p \in \mathcal{P}$ **do**
    for all event type $e \in \mathcal{E}$ **do**
      $aggCount \leftarrow EventCountAgg[p][e]$
      $scoreCount \leftarrow ScoreboardData[p][e]$
      if $aggCount > scoreCount$ **then**
        Flag player p for an illegal record in event type e.
        Mark player p as ineligible for rewards.
        Log "Illegal record: player p reported *aggCount* occurrences of event 'e', exceeding the official count of *scoreCount*."
      end if
    end for
  end for
end procedure

Integrity Check Rationale:
This validation process is a critical integrity check within the system. It compares each player's aggregated event counts—across various event types such as kills, deaths, round transitions, etc.—with the corresponding official counts from the persistent scoreboard. If a player's aggregated count exceeds the official value for any event type, it signals potential data manipulation or reporting errors. Flagging these discrepancies prevents fraudulent records from influencing the reward distribution and ensures that only valid, verified event data is used in subsequent processes. This integrity check is fundamental for maintaining fairness and trust in the game environment.

10.3 Detection and Handling of Non-participating Clients

Continuous event reporting from all clients is crucial for the integrity of the aggregated match record. Clients that miss reporting events undermine the completeness of the data and may compromise the accuracy of the final results. To mitigate this risk, our system monitors the number of missed events per client. If a client's missed event count exceeds a predefined threshold, the client is flagged as a non-participant, removed from the active session, and notified to cease sending further events. This ensures that only reliably reporting clients are considered for reward distribution.

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**
      Mark client c as **non-participant**
      Log "Client c removed due to excessive missing events (MissingCount: MissingCount[c])."
      NotifyClient(c, "You have been removed from the game session due to insufficient event reporting. Please cease sending further events.")
    end if
  end for
end procedure

Non-participation Management Rationale:
This algorithm is designed to ensure that the aggregated data reflects active and reliable participation. By monitoring the missed event count for each client, the system can identify clients whose participation is insufficient. Once a client's missed count reaches the threshold, the client is flagged, removed from the session, and notified to stop sending further events. This proactive measure prevents inaccurate data from skewing the final results and ensures that only valid, fully participating clients are eligible for rewards.

10.4 Summary and Future Extensions

A token-inflation mechanism, coupled with collateral staking, motivates players to submit legitimate events to the aggregator layer. The aggregator's detection logic and final timeline labeling then control who is slashed, who is denied rewards for incomplete data, and who is fully compensated for their documented achievements. Designers can refine the model by adjusting ρ, θ, or p to reflect the game's complexity and the prevalence of cheating. Future research may incorporate advanced anomaly-detection algorithms, zero-knowledge proofs of certain event types, or dynamic staking rates that adapt to player reputation. By integrating robust consensus with a well-calibrated economic model, multiplayer environments can deter dishonest behavior, provide meaningful rewards for fair play, and operate independently of closed-source server APIs.

11 Conclusion and Future Directions

This work introduced a framework for capturing, validating, and distributing in-game events through client-side hooking mechanisms—freeing third-party platforms from relying on private or unavailable server logs. By intercepting both ephemeral events (e.g. kills) and persistent states (e.g. scores or health) on the client, the system compiles cryptographically signed data that is robust to tampering and applicable across different server environments.

Two primary aggregator models anchor this architecture. The *lightweight* approach uses a supermajority threshold to confirm events and optionally anchors final logs in decentralized storage (e.g. IPFS). It suits scenarios where the aggregator is trusted (or at least neutral) and the overhead of a fully byzantine-fault-tolerant process is unnecessary. The *robust* approach, on the other hand, extends confirmation logic into a blockchain or DAG, protecting against malicious aggregator nodes. While it ensures stronger guarantees for high-stakes tournaments and e-sports, it also introduces higher network load and complexity.

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.

11.1 Potential Research Directions

Advanced Privacy Solutions. Incorporating zero-knowledge proofs to confirm event details—e.g., "a kill occurred" without disclosing exact player positions—would further protect sensitive in-game information while maintaining verifiability.

Machine-Learning-Based Anomaly Detection. As hooking datasets expand, real-time ML models could identify suspicious behavior (e.g., improbable performance spikes or contradictory kill logs) on the fly, triggering deeper manual or automated audits.

Efficient Handling of Massive Event Rates. Many multiplayer titles produce thousands of events per second. Designing aggregator pipelines or Layer-2 blockchain solutions that can finalize high-volume streams without creating bottlenecks remains an open scalability challenge.

Refined Economic Models. Dynamic or reputation-aware staking might better align participant incentives. For instance, players with histories of trustworthy reporting could stake less, while newcomers or previously flagged users stake proportionally more.

ross-Game Standardization. Establishing a universal hooking and event-schema standard across popular titles would streamline aggregator integration, reduce maintenance overhead, and foster a broader ecosystem of data-driven game applications.

11.2 Closing Remarks

By combining hooking-based data capture, partial coverage detection, secure event ordering, and an economic incentive layer, this system unlocks rich new possibilities for fair play, community analytics, and token-based ecosystems—even on servers where official logs are off-limits. The approaches outlined here—ranging from lightweight supermajority aggregation to full blockchain-backed finality—highlight a spectrum of solutions adaptable to different trust models and performance needs. As gaming continues to expand into realms of real-world finance, streaming, and e-sports, these design patterns will form a critical foundation for transparent, verifiable, and inclusive gameplay analytics.

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 and Hooking Framework. 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, Massachusetts Institute of Technology. Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (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] Author(s). 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. Available at: https://www.researchgate.net/publication/341654362_Directed_Acyclic_Graph-based_Distributed_Ledgers_-An_Evolutionary_Perspective

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

[10] Şahin, F. Play to Earn Web 3.0: The Future of Gaming and Marketing. Proceedings of the 8th International Scientific Conference "Telecommunications, Informatics, Energy and Management" (TIEM 2023), December 2023, Balıkesir, Turkey. Available at: https://www.researchgate.net/publication/376985610_Play_to_Earn_Web_30_The_Future_of_Gaming_and_Marketing

[11] 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