Blockchain-based Real-time Cheat Prevention and Robustness for Multi-player Online Games

Sukrit Kalra*
UC Berkeley

Rishabh Sanghi†
IBM

Mohan Dhawan
IBM Research

ABSTRACT

The gaming industry is affected by two key issues—cheating and DDoS attacks against game servers. In this paper, we aim to present a novel yet concrete application of the blockchain technology to address the seemingly disparate problems. Our approach uses blockchain to manage definitive game state and exploits peer consensus on every player action to track modifications to tangible player assets. While a key impediment to adopting blockchain for real-time systems is its high per-operation latency, our approach leverages several optimizations to enable real-time prevention of a large class of cheats where the reported client state is inconsistent with the observed state at the server. Further, blockchain-based games leverage the robust peer-to-peer architecture to successfully defend against DDoS attacks.

Our strategy enables flexibility to customize games with minimum modifications to game clients by porting server-side logic to smart contracts that execute atop peers. We evaluate our approach on a recent port of the multi-player game Doom. Our prototype can scale to client tickrates matched by modern games, and prevent cheats in <150ms for 32 peers deployed across the Internet, which is well within the latency requirements for online gaming.

CCS CONCEPTS

• **Security and privacy → Denial-of-service attacks**; *Distributed systems security*; • **Computer systems organization → Client-server architectures**; **Peer-to-peer architectures**; • **Software and its engineering → Consistency**;

KEYWORDS

Blockchain, multiplayer online games, peer-to-peer, client-server, cheat prevention, distributed denial-of-service.

ACM Reference Format:
Sukrit Kalra, Rishabh Sanghi, and Mohan Dhawan. 2018. Blockchain-based Real-time Cheat Prevention and Robustness for Multi-player Online Games. In *The 14th International Conference on emerging Networking EXperimentsand Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281438

*Work done when the author was at IBM Research.
†Work done when the author was an intern at IBM Research.

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281438

1 INTRODUCTION

The cheating industry for multi-player online games (MOGs) has become a multi-million dollar business [10]. Games such as Age of Empires have seen a huge exodus of players due to cheating [74], and for subscription-based games, player loss translates directly to a reduction in revenue. Further since game performance is susceptible to communication latencies, and because the game servers employ a client-server (C/S) setting (which is a central point of failure), game networks have become an easy target for DDoS attacks. As a result, Microsoft, Sony, Nintendo, Steam, etc., have all suffered debilitating DDoS attacks in the recent past [1, 27, 33, 39] enduring losses worth millions of dollars, in spite of investing heavily in maintaining and protecting their servers.

Much prior art in the area of cheat detection/prevention involves monitoring game state, either at the client's end or server or both. Server-side solutions require correct modeling of client behavior to validate client inputs [42, 44, 55, 56, 58, 66]. Such solutions mandate some trusted reference implementations against which client observations are matched. However, gathering reference implementations may not be practical in all scenarios. Client-side techniques often require support from sophisticated software [29, 35, 50, 70] or hardware [51] for intrusive end-host monitoring, and have met with strong resistance from the users [30].

While prior work has proposed several solutions as discussed earlier, a comprehensive defense for MOGs mandates two key properties. First, the solution must ensure that participating entities must abide by the rules of the game at all times. Second, the non-participating entities such as network bots must not be able to sabotage game play. In other words, the game must be resilient to the actions of malicious entities both within and outside the game.

Tracking malicious behavior in distributed systems has traditionally leveraged audit systems that rely on tamper-evident logging [44, 55, 56, 66]. A recent secure-by-design logging scheme is blockchain which fuses known cryptographic techniques with peer consensus to provide a tamper-resistant, distributed, append-only ledger. While traditionally developed for virtual currencies, blockchain has been adopted in domains such as physical asset management [16], healthcare and real-estate [15]. However, many academics and practitioners have argued that the use of blockchain, especially in a private setting, is driven by hype [8, 28, 38] since most of these applications do not necessarily require peer consensus amongst all participating entities. In contrast, peer consensus is a natural fit for the detection of cheating in multi-player games, given the subjective nature of cheating due to ambiguity in interpretation of the implicit rules of a game.

In this paper, we present a novel yet concrete application of the blockchain technology where it can be used efficiently. Specifically, we demonstrate an approach that exploits peer consensus in blockchain to address the problem of cheating in online games. Our technique leverages blockchain to manage the definitive game state, and uses peer consensus to validate all player actions that affect it in real-time. Our approach is novel because it obviates the need for a centralized, trusted server-based validation while retaining the prevention of a large class of cheats, equivalent to that available in C/S architecture, where the reported client state is inconsistent with the observed state at the server. Furthermore, since blockchain mandates a P2P architecture, we get added benefits of scalability and robustness against crippling DDoS attacks by design [4–7, 9]. Note that C/S architecture has always held precedence over peer-to-peer (P2P) deployment, in spite of P2P's robustness to withstand attacks and obvious scalability benefits. This bias is due to the fact that P2P clients are deemed untrustworthy and cheat detection is hard in the absence of a trusted intermediary. Lastly, our approach affords unprecedented flexibility in game customization without making major changes to the client itself.

While using blockchain facilitates easy migration to P2P architecture, it makes game state publicly visible to all peers. This problem gets exacerbated in the case of incomplete information games where it can lead to a loss of player privacy and introduce bias in gameplay. For example, players may willfully target other players with lower resources. We address this challenge by leveraging prior art [43] on privacy-preserving consensus for blockchains using secure enclaves. Note that while privacy can be enforced in a C/S setting through lack of visibility across user actions, absolute cheat prevention along with defense against DDoS attacks is typically harder to achieve due to architectural constraints.

In order to leverage blockchain, our strategy requires developers to port code running previously on the server to a smart contract, which programmatically encodes the rules of engagement and executes atop all participating peers in the blockchain network. There are two key benefits of this mechanism. First, it enables players to modify smart contract code according to their needs. Current games allow extremely limited or no customization, forcing individuals to modify game binaries, thereby violating the publisher's intellectual property and potentially defeating security features built into the game. Thus, the legitimate customization offered by our approach allows playing the game under custom rules and settings agreed upon by a group of players, which further reduces incentives for cheating. Second, our approach does not disrupt the existing client communication ensuring that games can work with minimum client modifications.

In our approach, the game performance is contingent on the blockchain platform's per-operation latency. While recent platforms [3, 31] allow throughput of thousands of operations per second, their per-operation latencies are still high. We present several optimization strategies to improve the per-operation latency and enable real-time cheat prevention while keeping the game scalable. We further argue the efficacy of our approach to support modern games though a study of scalability and performance requirements for fast-paced games available on Steam.

While our strategy is most suitable for games developed from scratch, we demonstrate its effectiveness by applying it to a recent port of Doom [11], a hugely popular, multi-player First Person Shooter (FPS) game, and use Hyperledger Fabric (or Fabric) [22] as our blockchain platform. Our choice of Doom was limited by the lack of modern open-source FPS games available. Note that our current implementation atop Fabric does not leverage secure enclaves for privacy-preserving computation. Our evaluation shows that the approach is scalable and reports a cheat prevention latency of <150ms with 32 peers deployed across the Internet, which suggests that the performance with secure enclaves in place will still be well within the latency requirements for online gaming. Further, our port of Doom can scale to client tickrates—the event sampling rate at the client to update the server—supported by modern games without affecting cheat prevention latencies.

This paper makes the following contributions:

- We demonstrate a novel yet concrete application where the blockchain technology can be used efficiently. Specifically, we present the first practical approach (§ 4 and § 5) to prevent cheating in MOGs using blockchain, which provides scalability and robustness by design, and also enables legitimate game customization.
- We present an implementation of our approach (§ 6) complete with several optimizations that make it suitable even for games with low-latency requirements, such as FPS.
- Our evaluation (§ 7) shows that our technique is accurate, scalable, and applicable to modern games. We also demonstrate case studies to highlight its extensibility and flexibility.

2 MOTIVATION

2.1 Peer Consensus as Anti-cheat

Game modifications (or mods [25]) are usually deemed as cheating by the game developers and publishers alike [20] for two reasons. First, modification of game binaries violates the publisher's intellectual property and may defeat security features built into the game. Second, it may pit players of unequal strength against each other. Publishers, thus, exercise strict control over the game ecosystem, right from the distribution of the game to its patches and addons.

Players, on the other hand, perceive games as fair when they achieve rewards commensurate with their actions. In online multi-player games, cheating provides an unfair advantage and detracts from the experience for other players. However, cheating is often dependent on the gaming community's social customs and implicit rules. Prior art [45, 46, 59–64, 68, 71, 77–79] cites cheating as the gulf between the designer's intent and the social aspects of games, the implicit rules, and the ambiguity of interpretation and use, i.e., the range of ways that individuals play. Given the subjective nature of cheating, peer consensus provides a level ground for all players and is an attractive and inexpensive anti-cheat mechanism. It also offers transparency and flexibility in designating what constitutes as cheating.

2.2 Online Games Most Vulnerable to DoS

There are three key factors that make a game network more vulnerable than traditional Internet-based services.

(1) Gaming platforms rely on unique, custom network protocols built for performance. There is very little information available

about how legitimate users interact with such services, thus making it virtually impossible for third-party solution providers to distinguish between a DDoS bot and a legitimate gamer, and causing mitigation for gaming servers to be much more challenging and resource-intensive.

(**2**) Online multi-player action games primarily depend on the absence of latency, and every millisecond between user input and its feedback can severely disrupt gaming experience. Thus, attackers need not shut down a server to cripple game networks; they merely need to introduce latency, which is much easier than bringing down an Internet-based service. For example, while an attack on an e-commerce website resulting in a half-second latency might go unnoticed, an assault on a game server causing the same delay would completely stop all activity—not because the server is unavailable, but because the game becomes unplayable.

Further, attackers may not even target the game network directly. Instead, they may focus on the upper-tier ISPs required to connect to the game server/data center. For example, attackers targeted the Final Fantasy XIV game servers by focusing on upper-tier ISPs causing instant disconnects [17].

(**3**) High traffic periods in the gaming world are predictable. Providers announce product releases in advance, and these dates are often followed by peak traffic. Even in the absence of DDoS attacks, gamers complain about latency due to high traffic. With servers already operating at capacity, the DDoS threshold is significantly lowered. Thus, it is the perfect time to launch assaults like the Christmas day attack [13].

3 BACKGROUND

3.1 Blockchain and Smart contracts

Blockchain is a distributed ledger that records transactions between multiple, often mutually distrusting parties in a verifiable and permanent way. These transactions are maintained in a continuously growing list of ordered "blocks", which are tamper-proof and support non-repudiation. Apart from the transactions, each block also includes state metadata, including the creation timestamp, the hash of the previous block in the chain, and smart contract code and data. A smart contract is an autonomous application that digitally enforces the rules of multi-party interaction through a blockchain.

The distributed ledger is managed atop a P2P network that adheres to a common protocol for inter-node communication and validation of new blocks. This validation involves a distributed, computational review on each block, and enables consensus even in a mutually distrusting environment. Once a block is validated, its data cannot be altered retroactively without collusion of the network majority.

3.2 Attack Model

There are four classes of cheats that we consider in our attack model.

(**1**) **Game** cheats occur completely within the game program and cheaters exploit design/implementation to gain advantage without additional programs or modifications. For example, cheaters can gain gaming advantage by purchasing a game item or virtual currency using a real-world currency. A cheater may also pay another person to play their character for them in a game session. Such cheats are often detected using statistical analysis of the game log files, and are relatively easier to detect in C/S architectures than the traditional P2P.

(**2**) **Application** cheats either modify the game executable or data files, or run programs that read from/write to the game's memory while it is executing. An example is information exposure where cheaters to gain access to information that they are not entitled to, such as their opponent's health, resources, etc. Other examples in this category include the use of bots/reflex enhancers. Such cheats usually require modification to the game client or running an external program to generate player input. Bots use computer AI to automate repetitive tasks, while reflex enhancers augment user input to achieve better results. FPS games suffer from reflex enhancers that automatically aim at opponents. Both C/S and P2P architectures are vulnerable to this form of cheating.

(**3**) **Protocol** cheats involve interfering with the game traffic. An attacker may insert, modify, drop, or duplicate game packets either sent or received. These cheats, however, are often dependent on the game architecture, i.e., C/S or P2P. Several of such cheats can be detected/prevented with use of synchronized time clocks and cryptographic protocols. However, collusion, which involves two or more players communicating using an out-of-band channel to gain an unfair advantage, is still extremely difficult to detect/prevent.

(**4**) **Infrastructure** cheats involve tampering game software (e.g., display drivers) or network hardware. For example, information exposure is enabled by modifying either the client's network or display drivers. Further, infrastructure-level reflex enhancers involve a proxy between the client and the server that modifies client's packets to improve the cheater's actions.

We address prevention of cheats where the reported game state is inconsistent with the observed state at the server, such as invalid modifications to tangible player assets. Thus, we aim to prevent majority of the cheats across the classes as discussed earlier, except for collusion and reflex enhancers. While collusion cannot be detected/prevented by any known anti-cheat mechanism, detecting reflex enhancers requires additional client-side heuristics [29, 35] even for state-of-the-art C/S games. Further, we also do not consider prevention of cheats related to client-only state such as map hacks since they are not tracked at the server. In other words, we aim to address all cheats that can currently be detected/prevented in the C/S game model.

We also assume a Byzantine majority of honest players who do not share their private keys to collude. Each player sends state updates and receives feedback over secure channels, i.e., messages cannot be tampered with, and all peers are equipped with secure enclaves to enable privacy-preserving computation. We place no other restriction on the players.

3.3 Game Model

Consider an MOG $\mathbb{G}$ with a set of k players ($\mathbb{P}$) and n available assets ($\mathbb{A}$). We model $\mathbb{G}$ as a sequence of states with transitions being player actions that take the game from state S_i to S_{i+1}. An action by player p is an update to a game asset a. Thus,

$$\mathbb{G} = [S_i] \mid \{S_i \xrightarrow{\tau} S_{i+1}\} \tag{1}$$

where $\tau = \mathcal{F}(p, a)$ represents game update due to player p ($p \in \mathbb{P}$) on an asset a ($a \in \mathbb{A}$), and $\mathcal{F}$ is the update function. Overall game state S_i is computed as an aggregation of individual client states. However, not all assets contribute to the client's game state; some assets may only be required to render relevant client state and do not impact the game logic. Thus, we define a game state S_i at time t as an aggregation of all relevant asset valuations across all players.

$$S_i(t) = \bigcup_{j=1}^{|\mathbb{P}|} \bigcup_{k=1}^{|\mathbb{A}_R|} [\![\phi(p_j, a_k, t)]\!]_v \tag{2}$$

where v is the actual valuation of ϕ (the asset valuation function) at the current timestamp t, with $p_j \in \mathbb{P}$ and $a_k \in \mathbb{A}_R$ (which is the set of all relevant assets), and $|\mathbb{A}_R| \leq |A|$.

Communication. Client to server messages involve sending individual client valuations $[\![\phi(p_j, a_k, t)]\!]_v$ over secure channels. Some of these messages may be batched together at the client to improve throughput. Server to client feedback is communicated corresponding to each individual valuation sent by the client. In case of batched client valuations, the server may batch the feedback as well.

Cheating. Since all messages are transmitted over secure channels, cheating may occur iff (i) client valuations sent to the server are themselves incorrect, or (ii) the feedback from the server is faulty leading to incorrect subsequent client valuations. Formally, we define cheating as an illegal state transition, i.e., a transition not allowed by the game due to an incorrect client valuation $[\![\phi(p_j, a_k, t)]\!]_v$. Thus, cheat prevention in games crystallizes to preventing clients from communicating incorrect valuations. A fair game $\mathbb{G}_{fair}$, thus, mandates that each game state is also individually fair.

$$\mathbb{G}_{fair} = [\, S_i \mid \mathcal{P}(S_i) \,] \; \forall S_i \in \mathbb{S} \tag{3}$$

where the function $\mathcal{P}$ determines the veracity of a game state, and $\mathbb{S}$ is the set of all game states. Since S_i is an aggregation over individual client valuations (per Eqn. 2), it is fair iff all individual client valuations are also correct. Thus,

$$\mathcal{P}(S_i(t)) = \bigcup_{j=1}^{|\mathbb{P}|} \bigcup_{k=1}^{|\mathbb{A}_R|} \mathcal{P}([\![\phi(p_j, a_k, t)]\!]_v) \tag{4}$$

4 OUR APPROACH

Overview. Our approach leverages blockchain's peer consensus to update all game states, enabling prevention of a large class of cheats. Blockchain offers a convenient way to detect cheating since the individual game states S_i map naturally to transactions in the blockchain with the function $\mathcal{P}$ being the consensus protocol. Thus, any state S_i that does not reach consensus on client valuations may potentially represent an attempt to cheat, and is not stored on the blockchain. In other words, a blockchain with client valuations as committed transactions represents the fair game $\mathbb{G}_{fair}$. As will be shown later in § 7.2.2, our approach does no worse cheat detection than the standard C/S architecture.

Our approach also enables decoupling asset management from other game logic into a smart contract. A publisher may choose to distribute their version of the contract that supports just visual enhancements. In contrast, publishers willing to embrace community efforts, can provide a game client that supports a more powerful contract managing both asset management logic and visual customization. Unlike game binary modification, the player community can leverage such game clients to develop their own smart contracts and customize the game as desired, which further reduces any incentive for cheating.

Workflow. Our approach relies on two components—(i) a smart contract that encapsulates server logic and transparently executes within a secure enclave, and (ii) a shim that interfaces between the game client and the contract, to preserve the original C/S-style communication. The shim also sets up peer network generation—a one time activity to setup blockchain peers and interfaces between the game client and the blockchain platform.

The game client sends events to the shim, either raw keystroke or mouse events or game events. The shim encapsulates these events and relevant asset information within a query object along with a nonce (to defend against replay attacks). It leverages the blockchain APIs to create a transaction for this query, which contains an invocation of the publicly visible smart contract API responsible for handling the event. Subsequently, the blockchain platform (a) leverages an ordering service [1] to determine the order of transactions received from different peers, (b) generates a block containing the ordered transactions, and (c) sends it to all peers for validation. The peers then execute these transactions in order locally (via the smart contract API invocation), and vote for consensus on each event following which they update their copy of the ledger. Finally, the shim polls the smart contract to determine the status of the transaction, i.e., whether peers achieve consensus, and communicates it back to the game client, thereby completing the feedback loop.

Note that such decentralized consensus entails that each peer has access to every peer's asset information stored on the ledger, albeit encrypted. This proliferation of data on the blockchain contradicts the goal to keep each player game state confidential and maintain privacy for the players. Although cryptographic protocols such as secure multiparty computation offer attractive solutions for privacy, they may not offer scalable, real-time performance. A promising alternative is the use of trusted execution environments such as secure enclaves. Thus, our approach mitigates privacy concerns by executing the game smart contract within secure enclaves.

4.1 Smart contract

The smart contract is akin to a server and encodes the logic to manage player assets. However, manually writing contracts from scratch is error-prone. Thus, we provide a template for developers to easily specify constraints on game events and assets, and a code generator that reads the specification to generate boilerplate contract code that separates management of game assets and its rendering logic. Additionally, this boilerplate code extracts the nonce from the per-transaction query object and determines whether or not the transaction has been observed earlier to defend against replay attacks.

Note that smart contract execution within the secure enclave is transparent to the client and the shim. Thus, we omit discussion on

[1] The ordering service is a high availability cluster of nodes that leverage protocols such as Kafka to reach consensus over the order of the transactions submitted to the blockchain. The orderers use the transaction's timestamp to order it within a block, before sending the block out for validation.

Tag	Description
Power	(pwId, change, factor), where pwId $\in \mathbb{N}$, change $\in \{+, \times\}$, and factor $\in \mathbb{Z}$
Asset	(aId, value, name, {power}), where aId $\in \mathbb{N}$, value $\in \mathbb{R}_{\geq 0}$
Player	{pId}, where $\{\texttt{pId} \in \mathbb{N} \mid 1 \leq \texttt{pID} \leq \text{Max}_p\}$
Affects	(pId, aId, pwId), where pId $\in (\mathbb{N} \cup \{\texttt{self}, \star\})$
Event	(eId, name, {affects}), where $\{\texttt{eID} \in \mathbb{N} \mid 1 \leq \texttt{eId} \leq \text{Max}_e\}$

Table 1: Constraint specification language.

smart contract registration with the secure enclave and subsequent privacy-preserving secure computation on the blockchain state. We refer interested readers to Brandenburger *et al.* [43] for the necessary details.

4.1.1 Constraint specification. Table 1 lists the language for specifying constraints and dependencies between game events and assets. While we do not claim completeness of the language, we believe it is expressive enough for most games. The specification has three parts—asset, event and player definitions. `Asset` specifies a game asset with several attributes—`value` specifies a default value, `power` defines the different modes of operation of the asset, where `change` indicates whether increase or decrease is additive/multiplicative, and `factor` lists the scale by which to increase or decrease the asset value. The `power` definitions model power modes of assets in the game where asset increment/decrement differs from normal game mode. `Event` lists the game event and its dependencies upon the already specified assets. The `Player` tag specifies a player.

4.1.2 Code generator. The code generator has three main functionalities. First, it creates APIs to add players after peer discovery, and start the game whenever players are ready. Second, it generates code to instantiate key-value stores (KVS) corresponding to each player's assets. This fine grained per-player per-asset partitioning helps minimize any potential read/write conflicts within the blockchain platform. Third, the code generator creates publicly visible smart contract APIs to instantiate a game session with peer players and manage the KVS on receipt of specific game events. Any additional logic must be added by the developer himself.

4.2 Shim

The shim provides an interfacing between game clients and smart contract running atop the blockchain. It provides mechanisms for (a) peer discovery, (b) peer network generation, (c) game instantiation, (d) mapping game events to smart contract code, and (e) ensuring client communication model remains unaltered. For the rest of the paper, *player* refers to the network entity visible to the game client, while *peer* is the network entity visible to the blockchain.

4.2.1 Peer discovery. Our approach assumes that there is one starting peer, akin to the player starting a game room. The shim for this peer is responsible for identifying available peers corresponding to the players that participate in a game. While our approach does not mandate any specific peer discovery protocol, a truly decentralized approach such as distributed hash tables [75] will work. To do so, the shim advertises the smart contract for the game and its associated consensus policy. Specifically, it listens for incoming connections from other peers for a designated time duration. Interested peers communicate their intent to play the game by sending their credentials, i.e., PKI certificates and IP address, to the initiator shim. Note that these credentials are required for instantiating a game session. The consensus policy is a boolean formula over asset update validation results communicated by each peer. In the absence of any user specified consensus criteria, we fallback on the blockchain platform's default consensus policy.

4.2.2 Network generation. Post peer discovery, the initiator shim creates and distributes a genesis block to all peers signifying the start of the common distributed ledger. With all the peers setup, the initiator shim starts a protocol to generate random numbers at each peer's shim using secure multi-party computation [83], and maps each peer's certificate with its generated random number (representing unique player identities). This mapping is required to maintain player anonymity. Note that this sensitive communication happens out-of-band and is not stored on the public ledger. Each peer's shim maintains this mapping for all peers. Whenever any asset validation response needs to be forwarded to the game client, the shim uses this mapping to replace the peer certificate with the correct player identity, so that the corresponding game client can render correct information for the affected player.

The initiator shim finally deploys the game smart contract on every peer, following which each peer shim independently executes the contract. The underlying blockchain platform ensures that the same contract is deployed on every peer. Any discrepancy would result in the specific peer not being able to join the game. Further, the contract at each peer has no knowledge of other peer's certificate to player identity mapping. This mechanism helps suitably anonymize players in the contract without affecting the game code.

4.2.3 Game instantiation. With the game smart contract deployed, every peer invokes a specific smart contract API to instantiate its relevant initial asset state on the ledger. Subsequently, the initiator shim invokes another smart contract API to launch the game UI and start the blockchain-based validation of game events.

4.2.4 Game events to smart contract mapping. In a typical game, raw user inputs such as keystrokes or mouse movements are sent to the server or other peers to simulate corresponding player actions. In our approach, these inputs are mapped to the relevant smart contract APIs responsible for handling the event (in a query object). However, to correctly decode the inputs and handle the events, our approach requires a client to register its (a) game session encryption keys, (b) network packet format for valid asset update events with the shim, and (iii) network packet format for event acknowledgements. This interaction enables the shim to (a) intercept and decrypt the network packets from the game, (b) subsequently, extract the asset updates from the network packets (given the registered packet format) and send them to the smart contract along with a per-event generated nonce, and (c) forward an acknowledgement to the client for every event it receives.

The above mechanism ensures our approach works with an unmodified client. However, clients may not share their session encryption keys for security reasons. Thus, we envision the shim, which is a minimal, standalone component by itself, to be integrated

as a module within the client. This inclusion makes the shim a part of the client's trusted computing base, and ensures that it is protected by the same security mechanisms as the client itself, i.e., a cheater may need to inspect process memory to extract the player identity to peer certificate mapping and deanonymize the players.

4.2.5 Client communication. The shim acts as a proxy for the game client, and ensures an unchanged client communication model by transparently transforming the messages between the client and the smart contract, which encodes the game's server-side logic. We now highlight features of our approach that enable this unaltered client communication.

(**1**) **Feedback loop**: Clients perform prediction along with entity interpolation to keep the game responsive. However, they must reconcile with the global game state when the server pushes the updates back to the clients. This server reconciliation step constitutes the feedback loop for the client. Our approach remains faithful to this model. If a `SHOOT` event occurs, the shim synchronizes with the client informing it about the reduction in ammunition. To do so, the shim maps each query object to a unique transaction identifier generated by the blockchain platform. It then polls the blockchain at every tick for the transaction's commit status, and sends an update to the game client, which is akin to the server acknowledging the receipt of events per tick in a C/S setting.

(**2**) **Event batching**: Events, like location updates or `SHOOT`, may be generated at a high frequency and can cause a large number of them to queue up at the shim leading to jitter and lag. The shim addresses this challenge by batching up similar but consecutive events with continuous acknowledgement numbers before creating the query object. This optimization ensures that dependent events are always processed in order. For example, if a client sends five successive `SHOOT` events within a single game tick, the shim batches them and creates one query object to decrease the ammunition by five. This batching reduces the invocations to the blockchain (and subsequently to the smart contract) to just one instead of five. However, if a player gets hit after the second `SHOOT` event, his associated health must decrease before he can shoot any further. Thus, the shim creates two separate batches of two and three `SHOOT` events respectively, with the health reduction event in between, thereby preserving the order of events.

4.2.6 Blockchain teardown. Since a game session is ephemeral and state does not persist across sessions, the shim tears down the blockchain at the end of the game session.

5 DISCUSSION

Security. The shim uses PKI certificates to bind peer identities with the blockchain, and ensure that an adversary cannot masquerade unless it has the victim peer's private key. Since the game's communication model remains unchanged, our approach does not violate any safety and liveness property that held true in the C/S version of the game, assuming that the private keys are safely managed. An adversary can, however, still misrepresent the amount of batching to his advantage. For example, in an FPS game, using a modified shim, a player can mask ammunition wastage by sending incorrect batch updates to the smart contract. If a player sends five `SHOOT` events in quick succession, three of which miss the target, a tampered shim can send a batched `SHOOT` event decreasing the ammunition by two. This is akin to using reflex enhancers that auto-generate user events and do not impact the client's state-managed at the server. Determining the veracity of these events requires additional client-side heuristics [29, 35] that fall outside the purview of our approach. Note that use of such heuristics to detect cheating is a requirement even for state-of-the-art C/S games.

Privacy. Although a game session is ephemeral and game state does not persist across sessions, Blockchain's open design exacerbates the privacy concerns during a game session by making it hard for blockchain-based games to keep sensitive asset information private in adversarial P2P settings. Consider two scenarios that demonstrate why we need safeguards before placing asset information atop the blockchain. For the scope of this paper, we limit ourselves to FPS, RTS and full information games. Prior work [65] has already explored blockchain for card games.

(**1**) In an RTS game, a player's strategy is influenced by the opponent's resources, such as available wealth. Full knowledge of a player's resources can influence game strategy and introduce bias against the player. In a C/S model, such critical information is kept secret with the server and not revealed.

(**2**) In full information games, like dice-based board games, merely putting the value of a dice roll on the blockchain raises concerns about non-repudiation. All players may not trust the value generated by the player under consideration.

Our approach leverages prior work [43], which has studied privacy-preserving smart contract computation in the context of blockchain using secure enclaves, to achieve consensus on the encrypted player asset values. Use of secure enclaves allow peers to operate with entire encrypted asset stores as opposed to individual asset entires, and are much more robust to dictionary attacks due to the significant randomness built into the data structures. For full information games requiring robust distributed random number generation in adversarial presence, our technique leverages prior art [40, 67]. Our approach can also be applied to other full information games, like Chess or Monopoly, that need only non-repudiation and have no privacy concerns.

Note that while in principle it may appear that secure enclaves obviate the need for trusted data sharing leveraging blockchain, there are several operational limitations that make use of enclaves alone prohibitive. First, enclaves offer limited physical resources, including memory, which may not be sufficient to keep the entire game state in memory. Second, use of secure enclaves is susceptible to rollback and forking attacks on stateful applications that make use of persistent storage [69, 76]. While enclaves provide mechanisms against main-memory replay attacks, persistent storage is not under the direct control of the enclaves and therefore harder to secure. Our approach based on prior work [43] mitigates such threats.

Extensibility. Our approach enables enhancements (albeit limited) to both visual and logic aspects of the game. This extensibility requires the developer to support enhancements using sprites via well-defined smart contract APIs. For example, addition of a new weapon in an FPS game would require (a) installing the weapon sprite on the client, and (b) regeneration of the smart contract with appropriate asset definitions. Since asset management

```
<Assets>
  <Asset aId="1" value="100" name="Health">
    <power pwId="0" change="+" factor="-1" />
    <power pwId="2" change="+" factor="1" />
  </Asset>
  <Asset aId="2" value="0" name="Ammunition">
    ...
  </Asset>
  ...
<Players>
  <player pId="1"> Player 1 </player>
  ...
<Events>
  <Event eId="1" name="Shoot">
    <affects pId="*" aId="1" pwId="0" />
    <affects pId="self" aId="2" pwId="0" />
  </Event>
  ...
```

Figure 1: Snippet of constraint specification for Doom [11].

is decoupled from the game, such customization does not require modifications to the game binary, and thus it neither tampers with any security features nor violate any intellectual property. Also, since the smart contract is advertised a priori, all players start on the same footing.

DDoS mitigation. Our approach, which inherits the blockchain's P2P architecture, has the desired protections to mitigate DDoS attacks built into it by design. In order to DDoS all game rooms in a C/S setting, the adversary may exploit the locality of the rooms on the central game server, and only needs to attack either the server or the network route leading to it. However, to cause equivalent damage in our approach, which has no central point of failure, the adversary must bring down at least one-third of the participants in each game room (when considering Byzantine failures), which is orders of magnitude harder than the C/S setting since the participants have no locality and can be spread anywhere in the world. Thus, unlike in C/S setting, our approach enables game interactions to continue even if several nodes go offline.

6 IMPLEMENTATION

We implement a prototype of our approach and demonstrate its effectiveness on a recent port of Doom [11]—a fast-paced FPS game that represents a worst case scenario for our approach due to the high-frequency of events and real-time constraints. We used Fabric v1.0 as our blockchain platform, and our shim required ~2K LOC of Java with ~500 LOC of Bash scripts as glue code. We parse the Doom constraint specification (snippet in Fig. 1) and implement the smart contract generator in Jennifer [24], resulting in a generated Go code of ~500 LOC. Since Fabric v1.0 supports only crash fault tolerance (instead of Byzantine failures), our default consensus policy involves a simple majority. At the time of writing, Brandenburger *et al.*'s Intel SGX-based [23] secure enclave implementation [43] for privacy preserving computation was unavailable for Fabric v1.0. Hence we compute consensus on unencrypted asset values and leave the SGX-enabled implementation for future.

(i) Doom client. We integrated the shim with the client and registered packet formats for 9 assets, i.e., ammunition, weapon, health, armor, keys, player position, invisibility pack, radiation suit and berserk pack. Our client did not use any encryption so we did not register any game session key with the shim. Overall, we modified ~70 LOC across 4 files to include the shim into the client.

(ii) Smart contract generator. Our Doom specification (snippet in Fig. 1) includes 9 assets and 11 events corresponding to shoot, weapon change, damage to sprites, gaining power ups (weapons, clips, medical kits, radiation suit, invulnerability, invisibility and berserk) and location updates. The code generator uses the output from the parser to generate a boilerplate smart contract consisting of (i) a verifier against transaction re-playability by checking the submitted nonce against previous nonces, (ii) per-player per-asset KVS definitions, (iii) publicly visible APIs for handling every event defined in the specification, (iv) `addPlayer` API to initialize player assets on the ledger, and (v) `startGame` API to begin asset validation and run the GUI. Our prototype includes support for both player and weapon sprites.

(iii) Shim. The shim implements three key functionalities.

- It provides a REST-ful peer discovery mechanism for ease of implementation. Interested peers communicate their intent to play the game by issuing a request to the listener with their PKI certificate and IP address as the payload.
- It automatically generates the blockchain network for the advertised peers. Fabric mandates blockchain network configuration to be specified in the `configtx.yaml` file, which contains entries for each advertised peer and is used to create the genesis block. The initiator shim distributes the generated genesis block and the smart contract along with the consensus policy to be asserted for each asset validation. Subsequently, each peer starts execution of the distributed contract by invoking the `peer chaincode` command in Fabric CLI and initializes its assets by calling the `addPlayer` API.
- Each peer shim generates the player identity to peer identity mapping using prior art [40, 67] and the initiator shim invokes the `startGame` API. On game start, the shim listens for client events, generates a per-event nonce, maps events to contract APIs and polls the blockchain for consensus at every tick, and relays it back as an acknowledgement.

(iv) Optimizations. Event validation is the complete processing of a single asset update. It has two stages—(i) peer consensus, i.e., agreement amongst the peers regarding the correct execution of the transaction in accordance with the smart contract, and (ii) synchronization of the ledger state amongst the peers. We discuss below several optimizations aimed at reducing event validation latencies.

- Blocks in Fabric can contain multiple transactions. When processing a block of transactions, Fabric acquires a block-level read/write lock on the KVS corresponding to the assets under consideration. For example, if a player shoots two successive bullets and the two events spawn two transactions within the same block, Fabric will reject the latter transaction. This problem is exacerbated when the smart contract maps the player (as key) with *all* his assets (as value) leading to sequential event validation. Since the scenario of different transactions operating on different assets belonging to the same player is frequent (e.g., ammunition and health reduction),

the inability to pipeline the stages in the event validation process restricts any improvements to the average validation latency. Our prototype overcomes this challenge by (i) generating smart contract code that splits each asset per player into separate KVS, and (ii) limiting a block to transactions affecting mutually exclusive KVS.

- We maximize the impact of the above optimization by ensuring an optimal block size that corresponds to the number of most frequently updated events operating on mutually exclusive KVS. This optimization amortizes the cost of ledger synchronization across the transactions in a block, since the synchronization happens once on block creation. A higher number of mutually exclusive assets would reduce the average event validation latency even further.
- Finally, we implement multi-threading at the shim to completely parallelize peer consensus for all the transactions in a single block. Note that each thread must handle only one type of asset; handling different types of assets at different execution points may lead to race conditions. Preventing such races would require appropriate thread synchronization. However, to ensure correctness and prevent delays due to synchronization, we do not consider this approach.

7 EVALUATION

Experimental Setup. We evaluate our approach with a recent port of Doom and leverage Fabric v1.0 as our permissioned blockchain platform. Unless specified, we conduct all our experiments in an Internet-wide deployment with Fabric peers as Docker containers running atop SoftLayer [32] servers located at Dallas, San Jose and Toronto. Each peer was provisioned with 4 cores, 16GB RAM and ran Ubuntu 16.04. We use Docker Swarm to orchestrate the setup so that the peers and Fabric services are deployed randomly across the overlay network of the servers.

7.1 Study of Modern Games

While our blockchain-based approach works for all classes of games, FPS games represent a worst case scenario due to their requirement of real-time consensus on every event update. To understand the performance and scalability requirements of modern games, we select 10 hugely popular multi-player FPS games on Steam [33] (of the total 15 available for Linux/SteamOS [18]), and determine the (i) average and maximum number of players per game session, (ii) average latency for successful gameplay, and (iii) client tickrate. Lastly, we also determine the distribution of low latency game servers for our setup, so that we can benchmark the desirable overheads for our prototype. Note that we restrict ourselves to Linux games for operational reasons alone.

Methodology. For each game, we compute the average and maximum player participation per session across top 500 game rooms using data from online game trackers [19].

We extract the average latency for a successful game session and the default client tickrate directly from the Steam console itself. Specifically, we list Steam servers in decreasing order of latency, and attempt a connection with each of them. If the game loads successfully, we play the game for 10 mins to determine actual playability. We record the average latency and default client tickrate, and stop if we do not perceive any jitter or lag. Otherwise, we attempt connection to the next server in the list.

Game	# Players		Average	Client
	Avg.	Max	Latency (ms)	Tick Rate
Counter-Strike 1.6	25.49	32	241	30
Counter-Strike: GO	18.93	63	240	64
Counter Strike: Source	14.84	64	234	66
Day of Defeat	4.59	30	245	30
Double Action: Boogaloo	0.42	17	288	30
Half-Life	1.75	31	258	60
Half-Life 2: Deathmatch	0.99	64	244	30
Left 4 Dead 2	2.38	24	272	30
Team Fortress Classic	0.41	15	253	30
Team Fortress 2	5.63	32	270	30

Table 2: Study of latency, tickrate and player participation in FPS games.

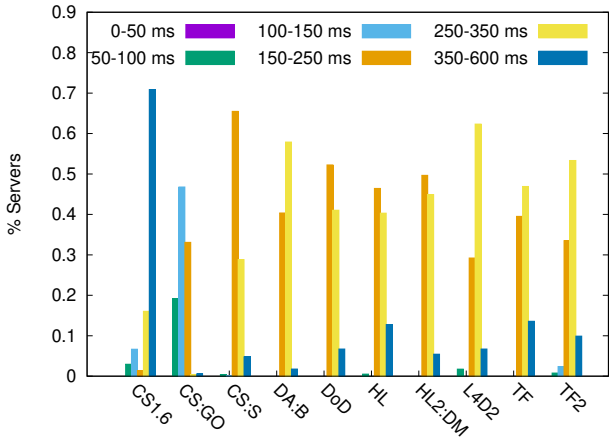


Figure 2: Distribution of servers based on observed latencies.

Since high latencies can hamper performance in multi-player games [21, 36, 37], we determine the latency distribution of the available game servers. To do so, we categorize the observed latencies for the corresponding Steam servers into 6 latency bins.

Results. We plot the findings in Table 2 and Fig. 2, and list the key take aways below.

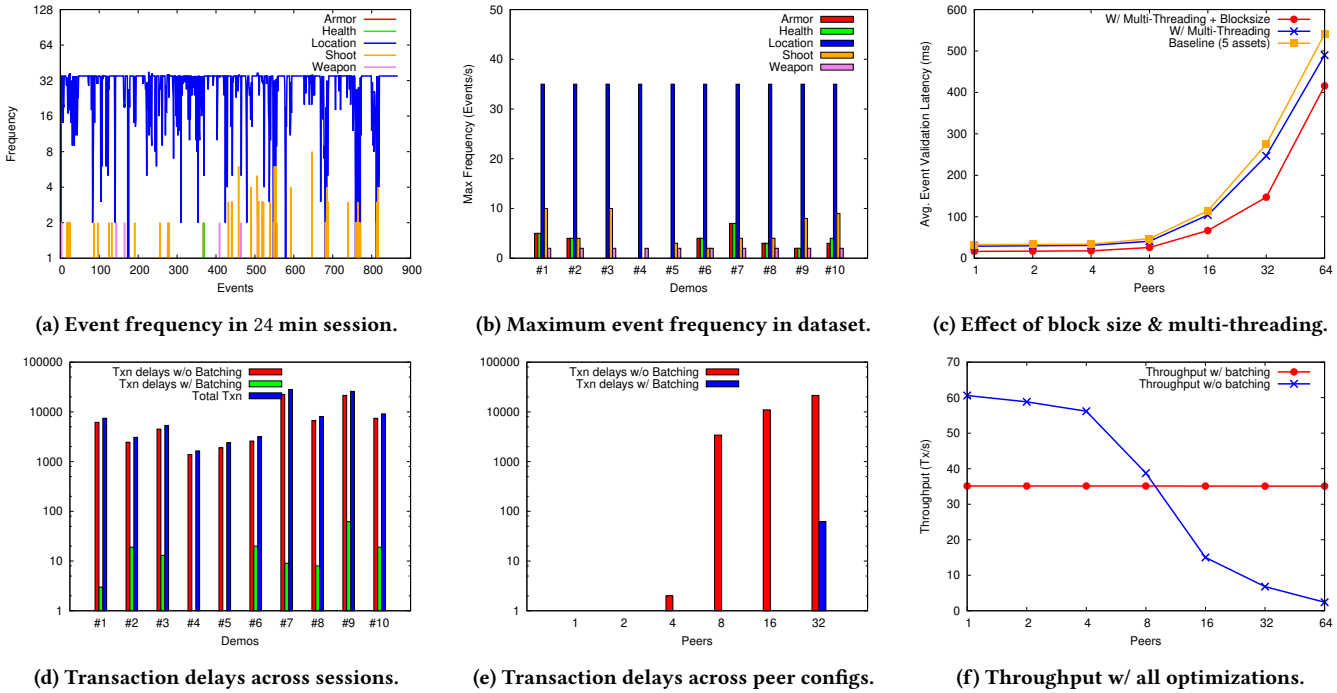
(1) The average latency for perceived successful play across all games in our setup ranged upwards of 230ms.

(2) Most modern FPS game clients operate at a tickrate of 30, and only 3 out of 10 games operate at a higher tickrate.

(3) The average player participation across all games was ~8, and only 3 games have player participation >32.

(4) Across all games, the majority of the servers available lie within the 100-350ms latency buckets. Even though lower latency is desirable, there are not enough servers available with <100ms latency.

The above study lists empirical observations based on the prevailing conditions at that moment, some of which may change over time. However, it provides an overview of the popular games, and other non-Linux FPS games would also exhibit similar statistics.



(a) Event frequency in 24 min session.

(b) Maximum event frequency in dataset.

(c) Effect of block size & multi-threading.

(d) Transaction delays across sessions.

(e) Transaction delays across peer configs.

(f) Throughput w/ all optimizations.

Figure 3: Evaluation of our approach with Doom.

7.2 Our Approach

7.2.1 Dataset. We analyzed 25 real-world Doom game sessions provided by the community [14, 34] and use our prototype to characterize the generated events at the shim. We log each event name, the asset it depends on and the event timestamp. Overall, the 25 Doom sessions clocked over 6 hours of gameplay and logged ~350K events.

Event frequency. We classify the logged events into five categories—armor, health, location, shoot and weapon. We omit the other low frequency events (per § 6) for brevity. Fig. 3a presents a time series plot for different events in the longest session (of 24 mins) that contained ~25K events. We observe that location update is the most frequently received event at the shim. Since Doom operates at 35 game ticks per second, we see a stable location update frequency of 35. We further plot the maximum event frequency distribution for each event per session in Fig. 3b and observe that shoot, besides location, is frequently updated, as expected in an FPS game. Events for other assets have a sparse distribution. Since the maximum update frequency is ~35, our approach must be able to handle at least 35 events per second per player to support multi-player version of Doom, which allows a maximum of four players.

7.2.2 Cheat Prevention. Our approach addresses prevention of cheats where the reported game state is inconsistent with the observed state at the server, such as invalid modifications to tangible player assets. We now discuss how our approach handles such cheats and list a comprehensive classification for the same in Table 3.

Built-in game cheats. Doom supports a total of 15 cheats [12] built into the game, of which only 10 are relevant in our context. The remaining 5 do not affect the relevant game state at the server and are thus not prevented by our approach. In fact, these cheats cannot be detected even in C/S setting as they only impact client-side rendering and the other players remain unaware of the changes. In the interest of space, we describe two such cheats and their mitigation.

Cheat	Our Approach	C/S	PB [29]/ VAC [35]	AS [41]	NEO [52]/ SEA [47]	RACS [81]	P2P RC [57]
Game							
Bug	✓	✓	✗	✓	✓	✓	✓
RMT/Power Leveling	✓	✓	✗	✗	✗	✓	✓
Application							
Information Exposure Invalid Commands	✓	✓	✗	✗	✗	✓	✓
Bots/reflex enhancers	✗	✗	✓	✗	✗	✗	✗
Protocol							
Suppressed update, Timestamp Fixed delay, Inconsistency	N/A	✓	✗	✓	✓	✓	✓
Collusion	✗	✗	✗	✗	✗	✗	✗
Spoofing, Replay	✓	✓	✗	✗	✓	✓	✓
Undo	✓	N/A	✗	✓	✗	N/A	N/A
Blind opponent	✓	N/A	✗	N/A	N/A	✓	N/A
Infrastructure							
Information Exposure	✓	✓	✓	✗	✗	✓	✓
Proxy/Reflex Enhancers	✗	✗	✗	✗	✗	✗	✗

Table 3: Our approach v/s other anti-cheat mechanisms (adapted from [80]).

• The IDDQD cheat makes the player immune to damage from other players by preventing updates to the player's health. It is impossible to use this cheat in our approach since the other players who send a SHOOT event damaging the malicious player will (i) reduce his health according to the rules defined in the smart contract code, and (ii) reach consensus on the asset's reduced value. When invulnerability is gained legally through a power up, peer consensus validates it, and the smart contract ensures temporary immunity to the player.
• The IDCHOPPERS cheat provides the player with a chainsaw. A player cannot use this cheat, since other players will not reach consensus on his state that has a new weapon without traversing the location on the map where the chainsaw is available for collection.

Application / Protocol / Infrastructure cheats. Our approach splits the game logic into a smart contract and a game client. The underlying blockchain platform ensures that the peers run the exact same smart contract and any discrepancy would result in the specific peer not being able to join the game. However, it is possible that the peers may resort to cheating by patching the game client itself. Even then, peer consensus on each relevant game event can detect several classes of such client modifications, and does no worse than the C/S architecture (per Table 3). It can even prevent collusion when limited to minority set of players. Detecting bots/reflex enhancers requires additional client-side heuristics [29, 35], which is a requirement even for state-of-the-art C/S games.

Cheat Prevention Latency. Note that the cheat itself may have been enabled much earlier than when the cheat event affecting the asset is dispatched. For example, the unlimited ammunition cheat may have been applied earlier, however its prevention can happen only when the available amount of ammunition has been fired. Thus, we define cheat prevention latency as the duration between the offending cheat event reaching the shim and the failure notification received for the corresponding event.

To determine the minimum absolute latency of cheat prevention, i.e., without considering the effects of Internet latencies and secure enclaves, we select 3 sessions [14] and replay them on Docker containers running on servers connected over a 1 Gbps LAN setup, while we manually played a game as the fourth malicious participant. We performed the experiment 10 times per cheat and noted that our prototype was able to prevent cheating in under 34 ms across all scenarios for a four peer setup.

7.2.3 Effectiveness of Optimizations. We perform controlled experiments to evaluate the effectiveness of the optimizations on average event validation latency (per § 6). To simulate realistic FPS gameplay, we constrain Swarm to place peers and Fabric services on servers located within the continental US only. Inter-continental online FPS gameplay is rare due to increased latencies. Due to the unavailability of Brandenburger *et al.*'s implementation [43], we do not deploy contracts in secure enclaves to preserve privacy.

We generate synthetic events (extracted from the sessions) and drive the shim at the highest successful event input rate possible, i.e., the shim sends events to the contract immediately after receiving validation notification for the previous event. This event rate is determined empirically by the event generator described next.

Event generator. The time for ledger synchronization (per § 6) is a configurable parameter and can be tweaked to maximize network throughput. We write a script that uses binary search to converge on the minimum state synchronization time, such that there are zero commit failures when sending events operating on the same KVS due to the acquired lock. The script then generates events operating on the same KVS at this maximum throughput.

(**1**) **Multi-threading**: We evaluate the impact of a multi-threaded shim on average event validation latency per peer for 5 threads, with each thread sending events corresponding to one asset type and operating at the highest possible input rate. We repeat the experiment 1000 times with the peers ranging from 1 to 64 and incremented exponentially, and block size set to one. Fig. 3c plots the results. We observe per-asset event validation latencies of ~104ms and ~247ms for 16 and 32 peer setups respectively. However, with a 64 peer setup, the average latency goes up to ~490ms.
Further, across the peer setups, use of a multi-threaded shim only moderately shaves off the per-asset event validation latency over the base case. While the decrease is attributed to the parallelization of peer consensus across asset updates, the benefits are reduced as ledger synchronization takes much more time than peer consensus and is not amortized with a block size of one. Further, since smaller peer configurations have much lower event validation latencies, the relative benefits of multi-threading are more prominent.

(**2**) **Transaction block size**: We determine the impact of varying block size by measuring the average event validation latency across 1000 runs for 5 different asset types with a multi-threaded shim forwarding events to the smart contract at the highest possible input rate. We vary the block size from 1 to 5, since events for only 5 asset updates are frequent. Note that the number of threads in the multi-threaded shim must always be equal to the number of assets, with each thread handling events for just one asset.
Fig. 3c plots the results. We observe that this optimization decreases the average latency by 100ms for the 32 peer configuration and by 48ms for the 16 peer configuration, over the multi-threaded setup. Overall, we note per-asset event validation latencies of ~66ms and ~147ms for 16 and 32 peer setups respectively, while with a 64 peer setup, the average latency stands at ~415ms. We attribute the improvement in average latency to amortization of the ledger synchronization time of the event validation pipeline across the 5 transactions corresponding to each block. Since modern games track more player assets, average latencies may decrease further with increase in asset types and block size.

Discussion. With latencies of ~147ms and below for an intra-continental setup of 32 peers or less, our prototype can achieve cheat prevention in real-time for most online games, including those reported in our study in § 7.1. With 64 peers (or more), the observed average event validation latency shoots up primarily due to increased synchronization between the peers, which is magnified by the intra-continental latencies. Further, use of secure enclaves for privacy-preserving computation may increase the average latencies by 10%-20% (per [43]).

However, recent advancements in areas, including blockchain sharding [54, 67], consensus algorithms [72], and blockchain technology [3, 31], can help reduce the average validation latencies to match the needs of online gaming (see Table 2) for 64 or larger peer setups.

Validity of results. There are two aspects to the evaluation of our blockchain-based cheat detection approach that can potentially

affect the validity of our results. First, the game traffic from the Doom clients is non-encrypted, which eliminates the encryption / decryption overheads. Second, our implementation and subsequent evaluation does not leverage computation within secure enclaves. A conservative encryption / decryption throughput per core when using AES in modern processors ranges about ~250MB/s [2]. With four cores per CPU, the encryption and decryption overheads for 1KB data would range <1ms. Our approach requires one decryption of the Doom messages to extract the asset values and one encryption of these asset values on to the blockchain. Since majority of the computation happens on smaller data sizes as Doom messages are typically much less than 1KB, we can safely assume the overall overhead due to the two decryption and encryption procedures to be bounded at ~1ms. A further overhead of 10%-20% due to the secure enclave processing (per [43]) still keeps the average event validation latencies to be well within the requirements for online gaming.

7.2.4 Scalability and Robustness. We evaluate the scalability and robustness of our approach in a 32 peer setup with event traffic from the 10 longest (of 25) Doom sessions. The shim operates at the event rate observed in the session. Further, we enabled all optimizations and set the number of threads per peer and the block size to 5.

(**1**) **Event batching**: Only continuous events for the same asset type can be batched to preserve event order (per § 4.2). Also, only one batch of events corresponding to an asset type can be processed at a time. In other words, if the shim creates two batches of the location update event interleaved by a shoot event, then the second batch of events will be dispatched to Fabric in the next time window corresponding to the setup's average event validation latency.

• We determine the effectiveness of batching by counting the number of event delays (because they could not be batched in the current time window and thus experienced a delay) and comparing it with the count of event delays when no batching was applied. Fig. 3d plots the results in log scale across all 10 sessions for a 32 peer setup. The maximum count of validation delays after batching was just 62 for session #9, which was the longest running session at 24 mins with over 25K events. Overall, we observe 10× to 1000× reduction in event delays due to batching across all scenarios.

• We plot the variation in such delays across different setups for session #9 in Fig. 3e. None of the setups, except the 32 peer case, reported any delays in event validation due to batching. In contrast, there were huge delays for 8, 16 and 32 peers without batching. Note that the delays are dependent on the nature of events arriving at the shim and the time window (corresponding to the average validation latency for the setup) considered for computing the delays.

• Fig. 3f shows the cumulative effect of all optimizations on throughput across different peer setups for session #9. We observe that even for the 32 peer case, which has an average throughput of ~7 transactions per second (per Fig. 3c), the cumulative effect of our optimizations allow the game to proceed at the client tickrate of 35. For the 1, 2, 4 and 8 peer setups, batching was not needed since the default throughput was higher than required. In contrast, for the remaining setups, the effects of batching outweigh all other optimizations and ensure that the game proceeds normally.

• We also measure the frequency of batching and the average batch size for the same experiment. We note that the highest average batch size was ~14 for the 32 peer case, which means a large number of events were batched. This observation is corroborated by the fact that location updates accounted for ~99.3% of the total events. The rest of the events can be handled with minimal batching.

Client Tickrate	# Transaction Delays					
	p=1	p=2	p=4	p=8	p=16	p=32
30	0	0	0	0	0	62
60	0	0	0	0	33	85
90	0	0	0	38	56	99
120	0	0	3	56	65	112
150	0	5	15	66	73	121

Table 4: Transaction delay with varying client tickrate and peer size (p).

(**2**) **Application to modern games**: Doom operates at a client tickrate of 35. Modern games may, however, operate at client tickrates of over 60. A good measure of the applicability of our approach to modern games is the extent of delays introduced due to batching. A small number of events experiencing delay is better. We replay Doom traffic from session #9 at higher tickrates and determine count of event delays for various peer setups. Table 4 lists the results. We observe that batching delays increase with increase in peer count and tickrates. However, even with 32 peers and at tickrate of 90, we observe just 99 potential delays. Also, scaling tickrate does not affect cheat prevention latency; it just increases the validation throughput due to event batching.

(**3**) **Defense against DDoS**: Our approach provides provable defense against DDoS by design (recall § 2.2). For empirical evidence, we observe the effects on event validation throughput for 8 and 16 peers with number of faulty nodes at 12.5%, 25% and 37.5%. We replay an event trace from Doom session #9 across all peers and note that the throughput remains the same even in the presence of malicious peers. We attribute this behavior to majority consensus that is still achievable in blockchain-based peer networks.

7.3 Case Studies

(i) Extensibility. Doom stores all the external data of a game (sounds, sprites, textures etc.) in the WAD file format. The community augments the game by developing patches to the original WAD file and distributing the mods in the PWAD file format. We use the `-merge` command in the recent port of Doom to introduce new weapon sprites into the game in the PWAD file format. This allows us to change the visual look of the weapons in the game. To change the logical behavior of each weapon, we make appropriate changes in the smart contract. For example, we made a weapon that never ran out of ammunition by disabling the reduction in ammunition in the smart contract, and a weapon with maximum damage by increasing the damage quantifier. A similar approach can be used to introduce new monsters that can withstand higher damage, making the game harder than intended.

(ii) Non-repudiation. We apply our approach to C/S-based Monopoly [26], a full information multi-player game where all claims can be verified through the blockchain's event log. Besides the smart contract, we added a robust, off-chain distributed random number generator (using [40]) to simulate an unbiased dice roll in a decentralized setting. Smart contract generation was trivial

as player assets are limited to currency and property. Property is defined on color basis, and has an owner and price attribute. Each player has 3 attributes: `location`, `currency` and `assets[]`.

8 LIMITATIONS & FUTURE WORK

(**1**) The adoption of our approach cannot be incremental and requires a fundamental change to the online gaming ecosystem, involving publishers, developers and players.

(**2**) A server in a C/S model may allow custom event order to enable better client experience. For example, servers may prioritize `SHOOT` events over location updates, allowing clients to update their state. Our approach relies on the blockchain's ordering service to finalize an order for client events.

(**3**) Our approach uses an aggressive batching strategy that, in the worst case, may only parallelize two transactions per tick with other transactions creating a conflict. This behavior may interfere with smooth entity interpolation at the client introducing jitters, and is comparable to a slow server scenario. However, it is improbable and can be mitigated by a smart selection of assets to be tracked.

(**4**) While some C/S games allow players to join in the middle of a game, our approach mandates that no player can join in the midst of a game session. This limitation is because it is hard to determine a fair starting state for the new player.

(**5**) Unlike in C/S setting, our prototype reports increasing validation latency with increasing peers, and cannot currently scale to massively multi-player online role-playing games (MMORPGs) involving thousands of users. However, recent advancements [3, 31, 54, 67, 72] can help mitigate the issue and blockchain-based MMORPGs may be feasible in future.

(**6**) While the optimizations presented earlier help understand the effectiveness of our approach in the context of the game traffic, determining the exact impact on the client requires a user study, which we leave for future work.

9 RELATED WORK

9.1 P2P Cheat Detection and Prevention

P2P games run the exact simulation on each client, passing identical commands and executing them in exactly the same way at clients to have identical effects on the games. This technique is much simpler and robust than sending each client's state to everyone and then validating them. Prior work [41, 47–49, 52, 57, 81] implement this Lockstep technique and its variants. In contrast, our approach does not run a simulation on every client; it instead requires consensus on every player action. Thus, it adopts the C/S strategy for validation but without the need for a trusted intermediary.

9.2 Other Cheat Detection Techniques

Accountability. PeerReview [56] uses tamper-evident logging to detect when a node deviates from the expected algorithm. However, it must be closely integrated with the application, which requires code modifications and a detailed understanding of the application logic. AVMs [55], on the other hand, use logging to record all incoming/outgoing messages and ensure correct execution of remote processes. Both these systems require a reference implementation, which entrusts faith in a single entity, and can only retroactively detect misbehavior/cheating and not prevent it. In contrast, our approach leverages consensus and a smart contract that encodes server-side logic to prevent cheating in real-time.

Client Validation. Bethea *et al.* [42] employ symbolic execution to extract constraints on client-side state, and use them to determine if the sequence of messages can be explained by any possible inputs. Fides [58] is an anomaly-based approach where a server-side controller specifies how and when a client-side auditor measures the game. To validate the measurements, the controller partially emulates the client and collaborates with the server. Mönch *et al.* [70] propose a framework that employs mobile guards, i.e., small pieces of code dynamically downloaded from the game server, to validate and protect the client. In contrast, our approach does not require any client-side emulation/simulation, which is heavy-weight and not scalable. However, it requires a shim and a contract that enables server-side functionality.

Trusted Computing. It requires a third-party (either hardware/software) to monitor all client behavior and prevent modifications to game binaries or run cheating programs, such as compromised shared libraries to implement bot/reflex enhancers. Schluessler *et al.* [73] utilize a tamper-resistant environment to detect bot generated input. However, such client-side solutions, including commercial efforts [29, 35], pose privacy concerns and have met with strong user resistance [30]. A2M [44] and TrInc [66] share the same goal as our decentralized blockchain-based technique, i.e., to remove participants' ability to equivocate. However, they do so using a trusted component but without incurring any communication overheads that are typically involved with decentralized approaches, including our approach and PeerReview. Moreover, our approach uses techniques from [43] developed for Fabric blockchain platform to enable privacy-preserving asset computation instead of proving equivocation.

Human Interaction/Behavior Proofs. Gianvecchio *et al.* [53] propose an approach to differentiate bots from humans by passively monitoring actions that are difficult for bots to perform in a human-like manner. Yampolskiy *et al.* [82] use an interactive mechanism that embeds CAPTCHA tests in card-based games. In contrast, our approach is real-time, yet non-intrusive to the players, and needs no extra client-side effort to handle cheating.

10 CONCLUSION

We develop a novel yet concrete application of blockchain to prevent a class of cheats where the reported client state is inconsistent with the observed state at the server. Use of blockchain provides the robustness of P2P architecture, and the flexibility to customize games via smart contracts. We evaluate our approach on the multi-player game Doom and show that it prevents cheats in <150ms for 32 peers deployed across the Internet, and scales to client tickrates for modern games. We also show the extensibility of our approach by customizing Doom with new weapons and their accompanying logic.

11 ACKNOWLEDGEMENTS

We thank our shepherd, Ben Zhao, and the anonymous reviewers for their insightful comments. We are also grateful to Seep Goel and Praveen Jayachandran for their feedback.

REFERENCES

[1] 2011 PlayStation Network outage. https://goo.gl/28eaFb.
[2] AES-NI SSL Performance. https://calomel.org/aesni_ssl_performance.html.
[3] BitShares. https://goo.gl/x3j2f6.
[4] Blockchain can fight cyberattacks. https://goo.gl/gwVPxk.
[5] Blockchain Could Have Prevented DDoS. https://goo.gl/ghmhqX.
[6] Blockchain DDoS attacks. https://goo.gl/Hux2yB.
[7] Blockchain Deters DDoS Attacks. https://goo.gl/HvtmTE.
[8] Blockchain is so hyped right now and many companies will get burned. https://www.cnbc.com/2018/06/06/blockchain-is-so-hyped-right-now-and-many-companies-will-get-burned.html.
[9] Blockchain will protect us. https://goo.gl/tM16FN.
[10] Business of video game cheating. https://goo.gl/5tKP2A.
[11] Chocolate Doom. https://www.chocolate-doom.org/.
[12] Chocolate Doom Cheats. https://goo.gl/svBJUV.
[13] DDoS attack on Xbox and PlayStation. https://goo.gl/VnbxYe.
[14] Doom Demos. https://doomwiki.org/wiki/Demo.
[15] Dubai Blockchain Strategy. https://smartdubai.ae/en/Initiatives/Pages/DubaiBlockchainStrategy.aspx.
[16] Everledger. https://www.everledger.io/.
[17] Final Fantasy XIV DDoS. https://goo.gl/5im94A.
[18] FPS games for Linux/SteamOS. http://store.steampowered.com/tag/en/Action/#tag%5B%5D=1663&category%5B%5D=1&os5B%5D=linux&p=0&tab=PopularNewReleases.
[19] Gametracker. https://www.gametracker.com/.
[20] GTA V Modding Community Explodes. https://goo.gl/haHqNy.
[21] How latency is killing online gaming. https://venturebeat.com/2016/04/17/how-latency-is-killing-online-gaming/.
[22] Hyperledger Fabric. https://goo.gl/RLfhyo.
[23] Intel SGX. https://software.intel.com/en-us/sgx.
[24] Jennifer. https://github.com/dave/jennifer.
[25] Mod (video gaming). https://goo.gl/SYMfk1.
[26] Monopoly. https://github.com/gmichaeljaison/monopoly.
[27] Pokemon GO down. https://goo.gl/i5CuoQ.
[28] 'private blockchain' is just a confusing name for a shared database. https://freedom-to-tinker.com/2015/09/18/private-blockchain-is-just-a-confusing-name-for-a-shared-database/.
[29] Punkbuster. http://evenbalance.com/.
[30] Punkbuster is hard to remove. https://goo.gl/Xr1FM9.
[31] Red Belly Performance. https://goo.gl/9kzjuL.
[32] SoftLayer Cloud Platform. http://www.softlayer.com/.
[33] Steam servers DDoSed. https://goo.gl/vnqRrv.
[34] ULTIMATE DOOM DEMOS. https://goo.gl/rbVTfD.
[35] Valve Anti-Cheat System (VAC). https://goo.gl/Wh6M5h.
[36] What is the maximum playable latency for FPS games ? http://www.mmo-champion.com/threads/751681-What-is-the-maximum-playable-latency-for-FPS-games.
[37] What's the max ping you consider acceptable for FPS? https://forums.anandtech.com/threads/whats-the-max-ping-you-consider-acceptable-for-fps.2176896/.
[38] Why blockchain hype must end. https://channels.theinnovationenterprise.com/articles/why-blockchain-hype-must-end.
[39] Xbox Live DDoS attack. https://goo.gl/nUw9CP.
[40] B. Awerbuch et al. Robust random number generation for peer-to-peer systems. *Theor. Comput. Sci.*
[41] N. E. Baughman et al. Cheat-proof Playout for Centralized and Peer-to-peer Gaming. *IEEE/ACM Trans. Netw.*
[42] D. Bethea et al. Server-side Verification of Client Behavior in Online Games. In *NDSS '10*.
[43] M. Brandenburger et al. Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric, 2018. https://arxiv.org/pdf/1805.08541.pdf.
[44] B.-G. Chun et al. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *SOSP '07*.
[45] M. Consalvo. Cheating: Gaining advantage in Video Games.
[46] M. Consalvo. There is No Magic Circle. *Games and Culture*, 2009.
[47] A. B. Corman et al. A Secure Event Agreement (SEA) Protocol for Peer-to-peer Games. In *ARES '06*.
[48] A. B. Corman et al. A Secure Group Agreement (SGA) Protocol for Peer-to-Peer Applications. In *AINAW '07*.
[49] E. Cronin et al. Cheat-Proofing Dead Reckoned Multiplayer Games. In *ICADCG '03*.
[50] M. DeLap et al. Is Runtime Verification Applicable to Cheat Detection? In *NetGames '04*.

[51] W.-c. Feng et al. Stealth Measurements for Cheat Detection in On-line Games. In *NetGames '08*.
[52] C. GauthierDickey et al. Low Latency and Cheat-proof Event Ordering for Peer-to-peer Games. In *NOSSDAV '04*.
[53] S. Gianvecchio et al. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. In *CCS '09*.
[54] Y. Gilad et al. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP '17*.
[55] A. Haeberlen et al. Accountable Virtual Machines. In *OSDI '10*.
[56] A. Haeberlen et al. PeerReview: Practical Accountability for Distributed Systems. In *SOSP '07*.
[57] P. Kabus et al. Design of a Cheat-resistant P2P Online Gaming System. In *DIMEA '07*.
[58] E. Kaiser et al. Fides: Remote Anomaly-based Cheat Detection Using Client Emulation. In *CCS '09*.
[59] G. King et al. *Tomb Raiders and Space Invaders: Videogame Forms and Contexts*. I. B. Tauris, 2006.
[60] J. Kücklich. Precarious playbour: Modders and the digital games industry. *Fibreculture*, 2005.
[61] J. Kücklich. Homo Deludens: Cheating as a Methodological Tool in Digital Games Research. *Convergence*, 2007.
[62] J. Kücklich. Wallhacks and Aimbots: How Cheating Changes the Perception of Gamespace. *Space Time Play: Computer Games, Architecture and Urbanism*, 2007.
[63] J. Kücklich. Forbidden pleasures: Cheating in computer games. *The pleasures of computer gaming*, 2008.
[64] J. Kücklich. Virtual Worlds and Their Discontents: Precarious Sovereignty, Governmentality, and the Ideology of Play. *Games and Culture*, 2009.
[65] R. Kumaresan et al. How to Use Bitcoin to Play Decentralized Poker. In *CCS '15*.
[66] D. Levin et al. TrInc: Small Trusted Hardware for Large Distributed Systems. In *NSDI '09*.
[67] L. Luu et al. A Secure Sharding Protocol For Open Blockchains. In *CCS '16*.
[68] T. M. Malaby. Beyond Play: A New Approach to Games. *Games and Culture*, 2007.
[69] S. Matetic et al. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security '17*.
[70] C. Mönch et al. Protecting Online Games Against Cheating. In *NetGames '06*.
[71] S. Morris. WADs, Bots and Mods: Multiplayer FPS Games as Co-creative Media. In *DiGRA '03*.
[72] R. Pass et al. Thunderella: Blockchains with optimistic instant confirmation. In *Eurocrypt '18*.
[73] T. Schluessler et al. Is a Bot at the Controls?: Detecting Input Data Attacks. In *NetGames '07*.
[74] D. Spohn. Cheating in Online Games. `https://www.lifewire.com/cheating-in-online-games-1983529`.
[75] I. Stoica et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01*.
[76] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security '16*.
[77] T. Taylor. Power games just want to have fun?: instrumental play in a MMOG. In *DiGRA '03*.
[78] T. L. Taylor. *Play Between Worlds: Exploring Online Game Culture*. The MIT Press, 2006.
[79] T. L. Taylor. Pushing the borders: Player participation and game culture. *Structures of participation in digital culture*, 2007.
[80] S. D. Webb et al. A survey on network game cheats and P2P solutions. In *AJIIPS '08*.
[81] S. D. Webb et al. RACS: A referee anti-cheat scheme for P2P gaming. In *NOSSDAV '07*.
[82] R. V. Yampolskiy et al. Embedded Noninteractive Continuous Bot Detection. *Comput. Entertain.*
[83] A. C. Yao. Protocols for Secure Computations. In *FOCS '82*.