Elympics Whitepaper

Michał Dabrowski

July 5, 2023
v0.2.1

Contents

1 Abstract

Smart contract are able to revolutionize gaming as we know it. Digital economies have been the foundation of free to play games for over a decade. These genres of games were the home for digital assets and unprecedented kinds of monetization. With global, enormous audiences, games are perfectly equipped to broaden the adoption of blockchain in everyday use, as it's a very natural technology to represent and maintain digital ownership, as well as provide unprecedented security and transparency to digital economies.

However, players are used to certain standards when it comes to user experience, especially in realtime gameplay. There are limits to practical applications of blockchain technology, as well as there are compromises one must take in order to achieve balance between security and experience.

In this paper we propose a decentralized game oracle network, which will serve as the missing link between the blockchain world of trustless digital economies with realtime gameplay experience. This technology will provide security and verifiability to gameplay, which will be the basis for any on-chain events that will happen after a certain gameplay is finished.

2 Principles

This document explains how security, verifiability and transparency can be achieved in the world of gaming.

2.1 Trust nobody

We realize that every entity in Elympics ecosystem can be compromised. Therefore we're treating both players, game hosts, and game developers in a decentralized system as potentially compromised entities. This makes us focus our efforts on potential exploits and on how to prevent them.

2.2 Transparency by design

It is impossible to use data one doesn't possess. It is impossible to leak data that is already public. Therefore we design our ecosystem as public, making it easy for everyone to access games and gameplays played on Elympics.

2.3 Trust the Math

Mathematics is more trustworthy than humans or code. All systems in Elympics should be fool-proof and their correctness should be enforced by math instead of specific implementation or company policy.

2.4 There are limits to practicality

As in any security system, we must first understand the practical limitations in a specific domain, so that the security measures applied do not limit practicality. We have to focus on specific game genres and gameplay types so that we can support designs that are popular and fun to play, instead of focusing on cool-but-non-practical gameplay. For example, we have to support Physics Engines as they're an inherit part of most games. Their limits in determinism have to be overcome and it's Elympic's job to make them easy to use in a fully distributed, trustless ecosystem.

3 Gameplay Anatomy

In order to achieve trustless, secure gameplay, we must first consider different practical applications of security in online games. Let's start by defining the common language that we'll be using for describing gameplay elements in the language of mathematics. We'll focus on mathematical abstractions of gameplay, without focusing much $fornow$ on the implementation details. However, later in this document, we'll explain how each of those mathematical goals can be achieved practically, and what limitations practicality enforces in a real-life, user friendly game.

We'll focus on time-framed world simulation, that is run in realtime without lockstep. However, as we'll see in later chapters, very similar definitions apply to different types of games.

light-bulb

In order to define replay-able, provable gameplays, we have to define specific time quants that are the basis of the simulation. From now on, we'll call those time quants **ticks**. Let's define, that each gameplay simulation starts at tick 0, and runs until the game finish condition is met.

Let's begin with some variable definitions:

- T - total number of ticks in a game. Note, that $T \in [0, \infty)$
- n - total number of players in a game. Note, that $n \in [1, \infty)$
- s_t - the state of the game simulation at tick t. This covers the entire world of the game.
- $i_{k,t}$ - input for player k at tick t. This defines the intent of a specific player at a specific point in time.

With those definitions in place, we can define the **Gameplay Function** f:

$$s_{t+1} = f\left(s_t, i_{1..n,t}\right) \tag{1}$$

This is the **gameplay function**. It takes the current state of the simulation, as well as the inputs from all players for a specific tick, and produces the state for the next tick. We can represent this as the game loop:

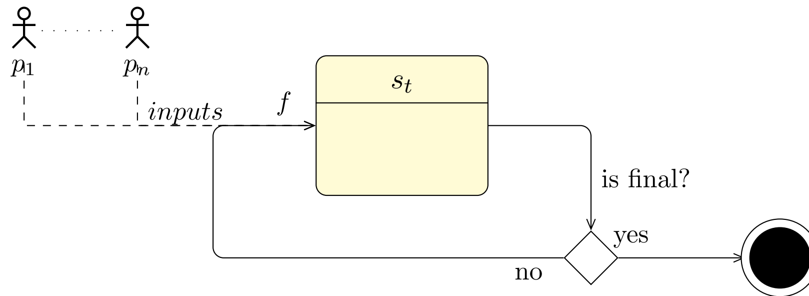

Figure 1: Gameplay function

We need a few more definitions to be able to start and finish such simulation:

- s_0 - initial game state. Note that this has to cover all data for the game when it is started.

- $o(s_t)$ - function that defines the outcome R of the match (result / winners / scores / etc) based on the final state of the match. This defines if t is the final tick of the game, based on current state of the simulation.

light-bulb

Note that such definitions enforce that a specific game is played by its rules, as players only communicate their intent in forms of inputs, instead of some part of the simulation. This means that there's no client authority in the gameplay.

3.1 Gameplay function

```
// given s0 is the initial state
// given f  calculates the next state by inputs of all players
// given o  returns the result for a state that is the final state
// given i  returns the inputs for players for all players for a tick

tick   = 0;
states = [s0];

while (o(states[tick]) == null):
  states[tick+1] = f(states[tick], i(tick));
  tick = tick + 1;
return o(s[t]);
```

With those definitions in place, we have to assess whether those definitions match practical use cases in the gaming space.

3.2 Turn-based Games

Turn-based games are considered to be the easiest in terms of multiplayer implementation. Since timing doesn't really matter, the gameplay can be defined as variable tickrate (the next tick only happens when a specific player provides their input). Also, in a classical turn-based game like chess, there is only one input present per tick (from one specific player only).

$$\forall_{t \in [0,T]} \quad \exists!_{x \in [1,n]} \quad i_{x,t} \neq null$$

We can define inputs for all other players as 0, extending the gameplay function, and have it perfectly match the definitions above.

3.3 Realtime Games

Realtime means that players can interact with the state of the game at the same time. In terms of the definitions above, we can say that:

$$\forall_{t\in[0,T]} \quad \forall_{x\in[1,n]} \quad i_{x,t} \neq null$$

This means that the game can be simulated with from the initial state knowing inputs for each player for each tick of the gameplay.

3.4 Practical limitations of universal gameplay

Particular game genres, as well as network situations, impose different limitations on the applications of above definitions. In the real-world scenario of a game, player experience should be the priority, and there are certain compromises and standards that have been established y the gaming industry over the years, that should be taken into considerations when designing Universal Gameplay Systems.

It's beyond the scope of this paper to go over every single feature. However, we believe they're worth mentioning, so that we are aware of the potential limitations and complexity of running parallel gameplay simulation.

- Constant tickrate vs lockstep (handling input timing and dropping late inputs)
- Lag compensation
- Client-side state prediction
- Client-side state reconciliation
- RPCs ($player \rightarrow host$ and $host \rightarrow player$)

4 Proof of Game

4.1 Universal Replays

In order to provide trustless, secure environment for gameplay execution, we propose a Proof of Game system, that will be based on universal, verifiable replays. Given definitions above, a full replay that is possible to be re-simulated in order to be validated, consists of:

- gameplay function $f(x)$
- list of all inputs from all players I
- initial state s_0

Given such information, we can compute and review the entire gameplay, as well as compute the final game outcome R in linear time (given that f is $O(1)$).

$$R_{s_0,I} = o\left(s_T\right) \tag{2}$$

$$s_t = \begin{cases} s_0, & t = 0 \\ f\left(s_{t-1}, i_{1\ldots n,t-1}\right) & t > 0 \end{cases} \tag{3}$$

Proof of Game is a gameplay system in Elympics, that is the foundation for Elympics Security. It removes the trust component in a true web3 fashion, making the entire ecosystem fully trustless. PoG makes Elympics and games hosted on Elympics trustless for:

- Players don't have to trust other players (gameplay security)
- Players don't have to trust hosts (verifiable, signed inputs, deterministic gameplay)
- Developers don't have to trust hosts (host reputation system / verifiable, deterministic gameplay outcome based on initial state and gameplay logic)
- Hosts don't have to trust other hosts (verification and reputation system)

4.2 Malicious actors

Now we have to think of each party trying to exploit the system, as well as the way the network can protect itself from such exploits.

4.2.1 Malicious players

The topic of cheating by players in traditional games is vast, and far beyond the scope of this whitepaper. However, many cheating techniques are eliminated by limiting player's influence on the game world (external authority). The responsibility of a player is to provide their input to the game. This crucial assumption renders all kinds of cheats (i.e. world state manipulation) impossible, because it means that the game has to be played by the rules given by the developer. This gives no guarantee on players generating those inputs by themselves, or with external aid. Therefore all feasible cheating techniques for players fall into one of the following categories:

Providing late input

This is mitigated by running the simulation in constant tickrate (without lockstep). Players have to provide their input to the authority on time in order for their input to be played. This however opens an opportunity for authorities to cheat with intentionally missing player's input (which is covered in the next section).

Reading theoretically inaccessible data

This are cheating techniques like wallhacks, which let players view data in a game simulation, that they shouldn't be able to access. It's done using techniques of game engine manipulation, extracting data from a synchronized world state. This class of methods can be mitigated differently for different genres of games, from limiting synchronized state to only the data that should be visible for a specific player (practical to games like RTS), client-side anti-cheat scripts that can be embedded into the game client, to ML-based gameplay input analysis (marking suspicious behaviors like looking at a wall for a long time).

Generating valid winning input with external aid

This is for example using a chess engine in an online game of chess. Or using aimbots in an FPS shooter. Or using specific hardware for an unfair advantage. Those techniques fall into the gray area and are a constant struggle of esports communities (even on tournament level). In a general sense, these techniques can be detected at a lower skill level using statistics by modeling learning curve of a typical player and looking for anomalies. Although worth pointing out, detecting those kind of cheats on the highest, tournament level is a constant struggle for esports community, and beyond the scope of this paper.

4.2.2 Malicious hosts

In a decentralized environment, one has to take rogue nodes (hosts) into consideration, especially when they have full authority of the gameplay (which is necessary for practicality). The host's main incentive is to submit a game result to the blockchain in order to get financial hosting reward (and in turn allow players to get their game rewards). We also have to consider a possibility of players becoming hosts, as this is not forbidden.

Censoring games with undesirable outcome

This is solved with a sequencer that assigns nonces to each game requested by the system. Each assigned nonce is expected to be delivered on-chain until a specific timeout. If it isn't, the node's stake is slashed.

Replacing inputs from specific players

A node could try to pretend they received valid, yet suboptimal inputs (moves) from a specific player. This is mitigated by highly optimized, tick-specific input signatures.

Censoring inputs from specific players

A malicious node could also *pretend* not to have received a specific inputs. This is mitigated by input presence in server-to-client messages (which are also necessary for gameplay experience reasons). However, this cannot be fully eliminated and has te rely on **node reputation system**.

4.2.3 Malicious developers

Malicious developers mean developers that include **hidden variables** inside the gameplay function f that give **unfair advantage** to some players (i.e. themselves) in an otherwise fair competition.

4.3 Requirements and assumptions

> **light-bulb**
>
> It's not practical for developers to define f as a function in a programming sense, as there are many internal variables and functions of a gameplay system. Instead, f should be expressed indirectly by the game code, running in a game engine (such as Unity) that developers are familiar with.

Elympics **Proof of Game** system with its **Universal Replays**, provides transparency and security for online games without compromising on performance. There are a few requirements that game developers using this system have to follow in order to play into the system and take full advantage of its possibilities. These can be guided in the documentation, as well as enforced with static and dynamic checks after the game function is uploaded into the system:

- The game needs to have no internal randomness (i.e. non-replayable). All pseudo-randomness has to be seeded deterministically from s_0, so that it can be replayed.
- The game server has to have no access to the internet, nor the disk space (isolated runtime)
- Physics Engines are chaotic systems, so the floating point implementations on different architectures have to be taken into considerations. This means that a full replay will only be able to run on the same architecture as the one it was originally hosted on.

5 Game Nodes Network

Given a secure gameplay execution environment provided with Proof of Game technology, the trust from a game Host is removed, and therefore can be decentralized. Everybody can join Elympics node network and start earning **$ELY** by hosting games in the network. This would require putting in a stake and starting a node program. There are several requirements for being able to start a gaming node in such a network:

- CPU capacity
- Available RAM
- Network bandiwth
- Running stability

Each node will be scored based on the parameters above, and added as a potential host (Oracle) to the network. The expanded matchmaking process will match players with hosts for optimal ping between the host and all players. The typical match process would consist of the following steps:

6 Matchmaking and Sequencing

In a fully decentralized network, where games are stored by the node network, gameplays are hosted by nodes, players can join (and incentivise) nodes to host certain gameplays, we still need some semi-centralised solutions. The matchmaking ticketing system has to favour practicality, as well as provide game-level knowledge and security based on skill and reputation of players. Regardless of a specific matchmaking implementation (rank-based, skill-based, history-based, etc.) the system needs to be aware of specific games and their requirements (i.e. required CPU and RAM in order to host a specific game), as well as location on players and nodes in order to match them in a group with relatively low latencies.

6.1 Player Matching

In order to provide an enjoyable gaming experience for all players, the matchmaking system must consider various factors when finding the best possible matches. Developers can configure queues with customizable settings, such as maximum wait time and bot matching, allowing them to choose between fast or optimal matchmaking, depending on their game's specific requirements. The system supports multiple queue configurations, such as duels, group, and team matchmaking.

Using skill level as a primary factor for finding matches is crucial for creating balanced and engaging gameplay. However, the matchmaking system can also take into account other

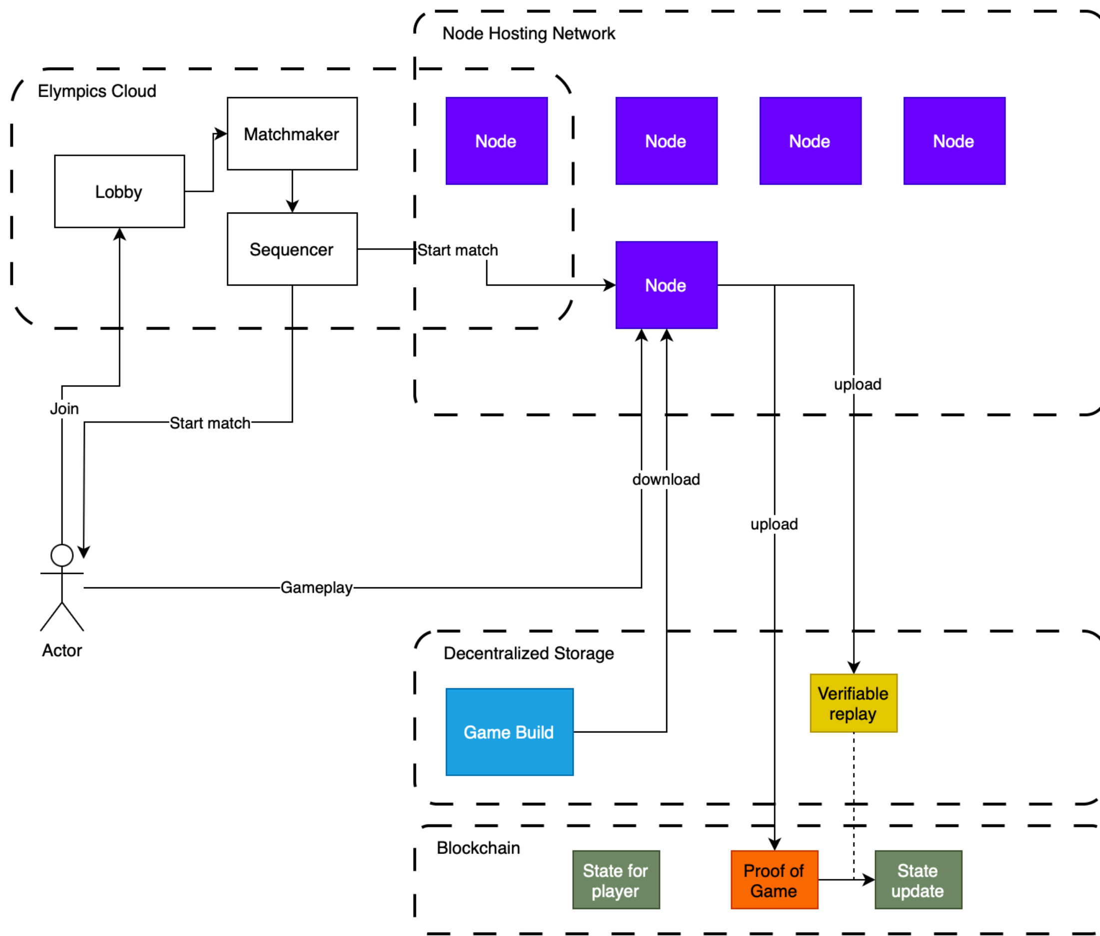

Figure 2: Gameplay scheduling

factors that the developer finds useful, such as the player's chosen role, their proficiency in that role, or their deck of cards. By considering these additional variables, the system can create more nuanced and fair matches, ensuring that all players have an enjoyable and competitive gaming experience.

6.2 Host Matching

specific gameplay function f with a match host. From the perspective of the system (and players):

- The CPU and RAM of the potential host required to run the gameplay function in a required tickrate

- The stability of internet connection in order to receive inputs and send out snapshots at a required bandiwth
- Latencies between a proposed host and all players of a potential match
- The reputation of the specific potential host

There are also considerations from the perspective of the potential host:

- economic viability of running a particular game (rewards vs complexity of f)
- impact on Node Reputation for running a given gameplay (the ability to host it successfully)

Such system for connecting potential matches with potential hosts can be implemented either by push (the system presents a potential match to the host) or pull (the node selects a match to host from all potential matches) logic, or a mixture of the above. Specifics of such implementation will either prioritise players, or hosts incentives.

6.3 Sequencing

Similarly to rollups in the blockchain space, hosting network has to work with sequencers that will provide unique nonces to all matches. Such sequencers will ensure that the network is resistant to replay attacks.

7 Summary

Elympics Proof of Game system with its Universal Replays, provides transparency and security for online games without compromising on performance. With the SDKs for different gaming engines, game developers can build their games in a very similar way to what they're used to, taking advantage of built-in best practices of parallel world simulation directly from esports. Building on top of these practices, as well as full gameplay transparency, we've defined a decentralized, trustless system to which games can be deployed and, in theory, outlive their creators. With a Proof of Game signed, verified and stored on-chain, developers can start building on-chain rewards for actual in-game accomplishments without relying on client-side security.