A decentralized approach to award game achievements

Francesco Bruschi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
francesco.bruschi@polimi.it

Donatella Sciuto
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
donatella.sciuto@polimi.it

Tommaso Paulon
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
tommaso.paulon@polimi.it

Andrea Marchesi
Milan, Italy

***Abstract*—Blockchain technology allows players to own in-game assets and to be rewarded with NFTs or tokens for their game achievements, thus can be a game changer for all the gaming industry. One central issue is how to check that conditions for achievements are met (e.g., that the player completed level 10). Current approaches open cheating backdoors (e.g. if the client checks the conditions) or introduce centralization points (if a backend checks the condition). Ideally, we would like to "run" games on chain, but so far that has not been possible due to the high computational cost, especially on Ethereum; however, the development of technologies like proofs of computation can solve this problem. Being able to run games on the blockchain, new decentralized rewarding systems can be built to ensure the fair and transparent rewarding of game achievements.**

***Index Terms*—blockchain, zero knowledge proofs, verifiable computing, rollups, Starknet, web3, gaming, nft**

I. Introduction

In the last few years the increasing adoption of blockchain technologies has created a new trend pushing both existing and new applications towards decentralization. This phenomenon is often referred to as *web3* and is considered the next evolution of the internet; here the power is shifted from big centralized entities to individual users, who have gained awareness on themes like privacy and personal data sharing. This particular context has the potential to evolve gaming from a non-rewarding activity (except for competitive players) to a rewarding one, leveraging the advantages brought by the blockchain such as transparency, immutability and guaranteed execution. At the time of writing many web3 games have already been developed, however all of them are either not fully decentralized, because only a part of the application runs on-chain, or extremely simple and consisting of very basic interactions. Complex and fully-decentralized games have been an unexplored field so far, but now the technology is mature enough to build this kind of games, thus maximing decentralization on one hand and rewarding players transparently and fairly on the other. The main obstacle to their development has mainly been the high computational cost on networks like Ethereum, which is the most successful smart contracts platform so far; several experiments have been tried with minor or application-specific networks, however this choice inevitably cuts off the game from the biggest web3 ecosystem and has a negative impact on the monetization of the rewards. The development of Layer 2 scaling solutions like rollups has the potential to solve this problem: they offer greater transaction speed and lower fees while maintaining the same security level as the Layer 1, as well as the interoperability with it. In this paper we first analyze the existing architectures for game rewards and the state of the technology, then we propose a method for distributing rewards based on verifiable game results and finally we apply it to Snake and Flappy Bird, two single player games respectively of discrete-time and continuous-time.

II. State of the Art

A. Existing architectures for game rewards

In the rewarding of in-game achievements two main components must be considered. First, the reward needs to be stored in some place, which we will refer to as *storage*, and it must allow players to retrieve the rewards they own. Second, every game needs an environment to be executed, which we will refer to as *execution*; rewards are distributed according to its results. We will now analyze the four possible configurations of these parameters, with a particular focus on the pros and cons of each solution.

1) client execution, server storage: The easiest way to distribute a reward is running the game on the client side and then send the results to a centralized server, which stores it. The obvious downside in this approach is that the client can lie about its game execution, to receive a reward he isn't entitled for. This configuration is used not only by Google Play Games [6] and Apple's Apple Store [2], but also by Steam [21], the popular videogames distribution platform. Here game achievements are online badges that can be shown off to other players, and they are issued upon client request when some game conditions are met; however, the client is able to cheat and get the rewards from Steams API without even running the game, in fact a project called Steam Achievements Manager [20] has been developed specifically for that.

2) server execution, server storage: To prevent clients from cheating about their achievements a possible solution can be asking them to submit only the input moves of the game in order to rerun it on a trusted server. In this way the game is executed twice (client side and server side), however the player has to know the correct moves to receive the reward. Heartstone [7] is a well known example of this approach; it is a card game in which players can earn in-games assets by winning matches against other players or the AI. In this case achievements have an in-game value as they can be used during matches, therefore the possibility of cheating must be averted; because of this, the game is executed on the server side with client's inputs to ensure that the player has really achieved the rewards. The main downside of this approach is that the reward is not truly owned by players and cannot be monetized as there is no underlying infrastructure for trading this asset.

3) server execution, blockchain storage: An evolution of the previous approach is a configuration in which the game is played on the server side but the rewards are stored on the blockchain, in the form of fungible or non-fungible tokens. The first advantage of this model is that it increases interoperability: tokens are controlled by a smart contract which is public and freely callable, therefore anybody can build an application that uses them. Secondly, every blockchain already offers marketplaces, exchanges and all the trading infrastructure that was missing in the previous case, so in-game assets are automatically monetizable on the chosen blockchain ecosystem. Splinterlands [18] is a card game which resembles Heartstone and uses this approach; it is executed on centralized servers but the achievements are stored on the Hive blockchain [8] in the form of NFTs and fungible tokens and can be traded on third-party marketplaces like NFTHive [11]. This configuration is becoming more and more popular; however, it still has one downside which is the centralized execution, meaning that theoretically the server for example could cheat to prevent the user from getting the rewards or could create infinite rewards thus destroying the value of existing ones.

4) blockchain execution, blockchain storage: The centralized server can indeed be removed from the architecture and be replaced by a set of smart contracts, with whom the player can communicate directly. This eliminates the costs related to servers maintenance and ensures that the game will be reachable as long as the blockchain itself is running, even if the developers leave the project. The only expense incurred by the creator is related to smart contracts' deployment; game computation instead is payed by players in the form of gas fees. Moreover, in this case no centralized entity can rig the rewarding process, because the rewards are controlled and assigned by the code contained in the smart contracts. This model is implemented for example by Alien Worlds [1], a play-to-earn game which is fully on-chain; however, existing on-chain games are closer to DeFi protocols rather than games, as they only allow very basic interactions with the blockchain like collecting and spending resources.

B. Ethereum

Ethereum [4] is the most popular blockchain for smart contracts and one of the most secure and decentralized networks. It offers a Turing complete language and allows transparent, trustless and immutable execution and storage. However, the high execution cost on this platform makes running even the most simple game infeasible; the network has indeed an inherent scalability issue because of its limited throughput and therefore transactions require high fees to be executed, especially when the network is congested. After the release of Ethereum many solutions have been proposed to solve this problem, which can be grouped in two main approaches: the first trend tries to scale the blockchain itself, which is also referred to as *layer1*, whereas the second one focuses on building scaling solutions (referred to as *layer2*) on top of the layer1 blockchain.

C. Other L1 networks

Many blockchains have been built to overcome the limitations of Ethereum, however the scalability increment comes along with several downsides. Solana [23], for example, is commonly considered one of its main competitors, as it can reach a block time of 400 milliseconds and theoretically over 60k TPS (Transactions Per Second) in opposition to the 10-15 TPS achieved by Ethereum. This performance boost however has been obtained at the expense of decentralization; the block size has been enlarged to reduce the competition for including a transaction in the next block and thus lowering the gas fees, but this approach has the downside of increasing nodes' hardware requirements and therefore centralizing the network. At the time of writing, Solana claims to have 1818 active validators while Ethereum's ones are more than 4500. Even if decentralization was not a concern, using a layer 1 blockchain more scalable than Ethereum would still not be practically feasible for the computation of games; game execution is in fact based on loops, which can have many interactions and take a long time to finish, so a transaction containing a long game execution would halt the network until it is completed. In order to prevent this scenario some execution fees must be set just like in the Ethereum protocol, thus making this kind of operations still expensive.

D. L2 scaling solutions

Layer 2 solutions are so called because they are built on top of the layer 1 blockchain, to increase its performances. These solutions move smart contracts' execution and storage off-chain, by using a smart contract on layer 1 with the following tasks:

- processing deposits and withdrawals
- verifying proofs demonstrating that everything happening off-chain is following the rules

The proofs can have different shapes, however the on-chain verification process is always much cheaper than performing the original computation on-chain. The layer 2 scaling solutions can be classified in four main groups, each one with its pros and cons:

- **State channels** [10] allow a set of users to perform unlimited private transactions off-chain

- **Plasma chains** [14] are separate blockchains anchored to Ethereum, and they are sometimes called child chains because they act as smaller copies of the mainnet
- **Sidechains** [17] are chains with their own consensus protocol and network parameters; they are connected to the mainchain via *bridges*, a mechanism to move assets between the mainchain and the sidechain
- **Rollups** [16][**zkrollups**] perform transaction execution off-chain and post the transactions and results on the mainnet

State channels are application-specific whereas plasma chains can only support basic transactions like token transfers or swaps, therefore they are not suitable for general purpose game execution. On the other hand, sidechains are basically clones of the original blockchain so they can scale the layer 1 only linearly, meaning that, for example, adding one sidechain would double the performance and adding two sidechains would triple it. Linear scaling is not sufficient in this scenario, as running a game is a computationally intensive task; rollups instead can scale computation exponentially, thus they are able to support game execution.

E. Rollups

Rollups can be divided in two main categories: optimistic rollups and Zero-Knowledge (ZK) rollups. Optimistic rollups store the *Merkle root* of the L2 state on a smart contract on the L1, whereas the complete state is kept off-chain. A Merkle root is the root of a tree in which each leaf node is labeled with the hash of a data block, and each non-leaf node is labeled with the hash of its child nodes' labels; it can be used to generate a cryptographic footprint from a set of data in order to reveal any subsequent modification on it. The state can be updated by publishing a *batch*, i.e. a collection of compressed transaction with also the previous and the new state root, along with some funds that will be slashed if the new state is proven to be invalid. These rollups are called *optimistic* because they assume by default that a batch is correct, and if it is the case no further action is required. However, the collateral supplied serves as an economic incentive to find and report incorrect transactions, in fact the receiving of the batch opens a time window in which anyone can publish a *fraud proof* on-chain, showing that the new state was not computed correctly from the transactions, in order to take the stake. There are various techniques to prove the new state inconsistencies to the rollup smart contract; in case of fraud, the contract reverts the batch and all the batches after it and pays the reporter using the collateral. Several successful implementations of optimistic rollups have been proposed, like Optimism [12] or Arbitrum [3], however they require all input data of a transaction to be published on-chain as calldata because it is needed to calculate the next root when the contract re-runs all the transactions of a fraud proof. This means that using these solutions all the input moves of a game would cost calldata fees; considering that a game can have even thousands of input moves and that calldata is expensive, optimistic rollups don't seem to be suitable for the purpose.

ZK rollups solve this problem because they don't need to save all the input moves on-chain, therefore they seem the solution which best fits the requirements. The main difference with optimistic rollups is that ZK rollups rely on cryptographic proofs instead of external actors to verify computation; these proofs are *Zero-Knowledge proofs* (ZKP), i.e. proofs that allow one party to prove to another one that a given statement is true without giving away any additional information about it. In this case it's not really important to keep the input moves secret, however this technology removes the need to store them on-chain thus avoiding the calldata costs of optimistic rollups. These proofs can be quickly verified directly on-chain; each batch contains one of them, proving that the post-state root is the correct result of the execution of the batch. Thanks to this mechanism ZK rollups have the additional advantage of reaching immediate transaction finality, since the proof is verified at the receipt of the batch; there is no need to wait as in optimistic rollups.

Currently, there are three projects offering a ZK rollup for general purpose computation: *Starknet* [19], which uses ZK-STARKs, *zkSync V2* [24], based on ZK-SNARKs and *Polygon zkEVM* [13], combining both types. Considering that Polygon zkEVM has not been released on mainnet yet and that ZK-STARKs offer better scalability avoiding the need of a trusted setup, at the moment the best choice for running a game on-chain seems to be Starknet.

F. Starknet overview

Starknet is a permissionless ZK rollup operating as a L2 network over Ethereum to increase scalability, by using ZK-STARKs. It offers an interface similar to a smart contract blockchain, as the complexity is handled in background; the idea is that a smart contract containing both the execution of the game and the reward distribution can be published on the network, and the moves of the game can be simply sent to it. Rewards can be implemented as fungible tokens and NFTs. However, smart contracts in Starknet need to be written in *Cairo* [5], a low-level language that converts program logic into ZK-STARK proofs. Due to the early stage of development, Starknet at the moment is not completely decentralized as the state root can only be changed by a centralized sequencer run by Starkware, the parent organization of Starknet. However, this situation should only be temporary and, according to Starkware roadmap, anybody will be able to run his own sequencer. Regarding the fees, presently the sequencer only takes into account L1 costs involving proof submission; the main factors affecting the L1 footprint of a transaction are:

- **computational complexity**: the heavier the transaction, the greater its incidence in proof verification cost
- **on-chain data**: L1 calldata cost deriving from data availability and L2 $\rightarrow$ L1 communication

III. Proposed Solution

A. Suitable games

The execution of a game on a blockchain in general, and on Starknet as well, is significantly limited by execution latency, as a transaction can take a few minutes to reach finality. In games with a constant back-and-forth communication, like action multiplayer games for example, this is a serious problem that can worsen the user experience; for this reason, the proposed solution will only focus on single-player games where game execution is deterministic and input moves can be verified in a single transaction.

The idea is to make the client record all the player moves in a log, and then, using a Cairo representation of the game's engine, to produce a proof of the state reached by the player. The proof will then be checked on-chain, to convince a smart contract of the achievement reached.

A game can be formalized as a function $\mathcal{F}$ which takes in input an initial state S_i and a sequence of moves m and returns a final state S_f:

$$\mathcal{F} : (S_\text{i}, m) \rightarrow S_\text{f}$$

$\mathcal{F}$ can be further structured in two functions: `transitionFunction`, which takes a state and a move and returns a new state, and `isFinalState`, to check whether a state is final or not.

Algorithm 1 Game execution

```
1:  procedure RUN_GAME
2:      $isFinal \leftarrow false$
3:      while $isFinal \neq true$ do
4:          $S_{n+1} \leftarrow$ transitionFunction($S_n$, $m_n$)
5:          if isFinalState($S_{n+1}$) $== true$ then
6:              $S_\text{f} \leftarrow S_{n+1}$
7:              $isFinal \leftarrow true$
8:          else
9:              $S_\text{n} \leftarrow S_{n+1}$
10:         end if
11:     end while
12: end procedure
```

The case in which no more moves are available but current state is not final can be handled arbitrarily, for example reverting the execution or returning the non-final state.

B. Strategies to hide winning sequences

Since game execution in this scenario is deterministic, and considering that Starknet transactions are public and transparent, it follows that players could just copy other users winning transaction to get the same reward. A first possible solution to this problem could be implementing some privacy features to hide players moves in the transaction; the problem with this approach is that winning sequences could be disclosed online by using other channels thus making it ineffective. A better strategy could be to reward only the first player who finds a specific solution; in this way, players copying a solution that has already been used would get no prize. However, also this approach has some downsides: first of all, each already found solution should be stored somewhere so that it can be compared against new ones. The most naive way could be storing them directly on Starknet, but it would be very costly as the moves increase, as seen with optimistic rollups; a better approach could be to ”summarize” each found solution into a single hash. However, it's impossible to tell which solutions have been discovered just by looking at their hash, so discovered solutions should be stored off-chain to let players know which solutions are not usable anymore. In addition to the issues concerning the implementation, this approach has also other problems. First, the number of claimable rewards is constantly decreasing making harder and harder to get them, and at some point there might even be none left to claim; this is a disadvantage for players joining the game later on. Another potential problem is related to the *miner extractable value*, i.e. the ability of miners (or, in the case of Starknet, of the sequencer) to include, exclude, and change the order of transactions in a block in order to extract some value [9]; for example, a malicious sequencer could discard a transaction with a solution and send it from its address to steal the reward. To solve these issues another approach can be adopted, which is to make the initial state variable and dependent on the player. In this way a player cannot copy the solution of another user, because being the initial state different he would obtain a different outcome; it also solves the limited reward problem of the previous solution, as there is a reward for every possible address. However, this strategy introduces a new problem: nothing is preventing a player who has already mastered the game from creating lots of addresses to claim potentially unlimited rewards. This could be solved in the future with solutions like *Proof of Humanity* [15], which is a system to assign a human identity to an address. The last two approaches are both valid in their own way; the third one better fits situations in which the prizes are in the form of non-tradable rewards (for example memberships), whereas the second one is best used for distributing tradable rewards like tokens.

C. Flappy Bird PoC

In this section the approach will be applied to a continuous time game, namely Flappy Bird. Flappy Bird is a side-scroller game where the player controls a bird, attempting to fly between columns of green pipes without hitting them; it has been chosen as an example because it requires 2d physics simulation while being very simple. The state of the game is composed by the following data structure:

- the position **pos** of the bird, defined by its x and y coordinates
- the velocity **yVelocity** of the bird on the y axis
- the offset **pipesOffset** of each pipe's gap from the ground (these offsets are what makes the initial state player dependent)

The state can be defined in Cairo as follows:

```
struct Position:
   member x : felt
   member y : felt
end
struct State:
   member pos : Position
   member yVelocity : felt
   member pipesOffset: felt*
end
```

Having defined the state of the game, the next step is to design a general and reusable template to handle game execution. The main function called by the players to get rewarded is `validateGame`; in its body two functions are executed, namely `getInitState` to initialize the state and `getFinalState` to get the final state. Once obtained the final state, the reward can be distributed according to it.

```
@external
func validateGame{
      syscall_ptr : felt*, pedersen_ptr :
         HashBuiltin*,
      range_check_ptr
      }(moves_len : felt, moves : felt*,
         tokenAddress : felt) -> (pos:
      Position):
   alloc_locals
   let (local address) =
      get_caller_address()
   let (initState : State) =
      getInitState(address)
   let (local finalState: State) =
      getFinalState(moves_len, moves,
      initState)
   # reward distribution
   return(finalState.pos)
end
```

The `getFinalState` function is recursive and it is called for each state, checking if it is final or not; a state is final either if the sequence of moves has ended or the game is over due to game rules (for a collision for example). If the state is not final the recursive call goes on, otherwise the final state is returned.

As already said, the idea here is to make the initial state variable player dependent; this can be achieved using `getValuesFromSeed` function, that takes as input a seed (in this case, the address of the player), how many values to generate and their max possible value. This function is executed recursively and for each value generates a new seed by squaring the previous one, as this is cheaper than hashing.

In this case the transition function which derives the next state from the current only needs to update the `y` velocity and the bird's position checking if the character is jumping or not. As already told, a reward which best fits this approach can be a membership; in this example, this can be achieved by writing player's address on-chain, so that it can be used as proof of membership. For example, the membership can be granted to the players who have reached an x position of at least 200.

D. *Snake PoC*

The second PoC is Snake,a discrete-time game where the player maneuvers a growing line (the "snake") that becomes a primary obstacle to itself; the player loses when the snake runs into the screen border, or itself. In this case the system will implement a different strategy, rewarding only the first player who submits a specific solution. The state here consists of three pieces of data:

- **body_pos**, the list of snake's body positions
- **food_pos**, the list of food pieces' positions
- **food_index**, the index of the food piece currently available on the board

Unfortunately, Cairo doesn't have something like `list.length()` to retrieve the head of the snake from **body_pos**, therefore this information (called **head_index**) must be saved in the state too. This structure can work, however this state forces to rewrite the body position on each state transition. This can be avoided by keeping track of the tail of the snake, by using a field called **tail_index**; in this way, after a state transition only a single felt needs to be rewritten. Moreover, this time also move's hash must be stored, in order to reward the first player uploading a specific solution.

Since the initial state is not variable. it can be simply initialized in the storage. A more evolved way to do it though is to generate the state from a seed in the storage; doing this, a single value must be updated to change the initial state (for example, when most of solutions have already been claimed). The transition function first has to compute the next position of the snake, then calculates the hash of the moves and finally creates the next state.

In this case, the `IsFinalState` function must check three conditions in order to determine if a state is final or not:

- if the snake has eaten all food
- if the snake has exited the borders of the game (resulting in a loss)
- if the snake has collapsed on itself

If one of these conditions is true, the state is final. Concerning rewards distribution, a registry is needed to store the solutions which have already been discovered; a solution here is modeled as the hash of the moves, calculated in the transition function. When a solution is found, the address who discovered it is written on the storage

IV. Implementation Assessment

A. *Costs*

We experimented with both games, playing matches of various lengths. The total cost for running Flappy Bird can be divided into two components: fixed and variable costs. Fixed costs do not grow as the input moves grow, and include all the operations performed in the main function; they are fixed because this function is only run once. Variable costs instead grow with the input moves, and includes all the actions performed in `getFinalState` recursive call (the function is called once for each input move). Moreover, the

TABLE I: Flappy Bird costs

N. of moves	Game time	ETH cost	USD cost
1	0.6	0.00001	0.016317
48	3.2	0.000011	0.017949
175	11.6	0.000012	0.019580
317	21.1	0.000014	0.022816

TABLE II: Snake Costs

N. of moves	Game time	ETH cost	USD cost
1	1	0.000019	0.031075
13	13	0.00002	0.032711
35	35	0.00002	0.032700
69	69	0.000021	0.034346

amount of input moves is determined by the number keyboard samples per seconds (in this case, 15 samples per second). An interesting feature of the proof type used in Starknet is that proof cost grows sublinearly with input size.

To give an idea of the actual costs, in Table I and II execution cost by number of moves is reported for matches of various lengths (assuming a gas cost of 6 gwei, as of time of writing).

The same graph can be plotted also for Snake:

The experiment suggests that the costs are very low, and that they scale well as the games lengthen.

V. Conclusion

The present work shows that it is possible to create decentralized play-to-earn games using a Zero-Knowledge STARK rollup on Ethereum, at acceptable costs. Even if complex and continuous-time games could at the moment be expensive to prove, the Ethereum scaling roadmap projects significant decreases in the cost of rollups. The architecture we considered addresses only single player games. One interesting dimension to explore would be that of multiplayer games, where approaches based on multiparty computation could allow to offer guarantees where they are perceived as even more important.

Other further work could consider the problem of certifying the time it took the player to complete the match. With the current architecture, a player could play a slowed down version of the game, making it easier to reach a given goal (although it would take longer). It would be interesting to explore how it would be possible to certify that the moves were played within some time bounds. The certified results could be used to distribute *soulbound tokens* [22], non-transferable NFTs that can be used as achievements to certify the completion of a gamified test.

References

[1] *Alien worlds homepage*. https://alienworlds.io/.

[2] *Apple Store*. https://www.apple.com/app-store/.

[3] *Arbitrum homepage*. https://developer.offchainlabs.com/docs/Inside_Arbitrum.

[4] *Ethereum homepage*. https://ethereum.org/.

[5] Lior Goldberg, Shahar Papini, and Michael Riabzev. *Cairo – a Turing-complete STARK-friendly CPU architecture*. Cryptology ePrint Archive, Paper 2021/1063. https://eprint.iacr.org/2021/1063. 2021. URL: https://eprint.iacr.org/2021/1063.

[6] *Google play games*. https://play.google.com/googleplaygames.

[7] *Heartstone homepage*. https://hearthstone.blizzard.com/en-gb.

[8] *Hive.io. 2020. Hive: Fast. Scalable. Powerful. The Blockchain for Web 3.0.* https://hive.io/whitepaper.pdf.

[9] Aljosha Judmayer et al. *Estimating (Miner) Extractable Value is Hard, Let's Go Shopping!* Cryptology ePrint Archive, Paper 2021/1231. https://eprint.iacr.org/2021/1231. 2021. URL: https://eprint.iacr.org/2021/1231.

[10] Lydia D Negka and Georgios P Spathoulas. “Blockchain state channels: A state of the art”. In: *IEEE Access* (2021).

[11] *NFTHive marketplace homepage*. https://nfthive.io/.

[12] *Optimism homepage*. https://www.optimism.io/.

[13] *Polygon zkEVM homepage*. https://polygon.technology/solutions/polygon-zkevm.

[14] Joseph Poon and Vitalik Buterin. “Plasma: Scalable autonomous smart contracts”. In: *White paper* (2017), pp. 1–47.

[15] *Proof of Humanity*. https://www.proofofhumanity.id/.

[16] Tobias Schaffner. “Scaling Public Blockchains”. In: *A comprehensive analysis of optimistic and zero-knowledge rollups. University of Basel* (2021).

[17] Amritraj Singh et al. “Sidechain technologies in blockchain networks: An examination and state-of-the-art review”. In: *Journal of Network and Computer Applications* 149 (2020), p. 102471.

[18] *Splinterlands homepage*. https://splinterlands.com/.

[19] *Starknet homepage*. https://starkware.co/starknet/.

[20] *Steam Achievements Manager*. https://github.com/gibbed/SteamAchievementManager.

[21] *Steam homepage*. https://store.steampowered.com/.

[22] V.Buterin. *Soulbound*. https://vitalik.ca/general/2022/01/26/soulbound.html.

[23] Anatoly Yakovenko. “Solana: A new architecture for a high performance blockchain v0. 8.13”. In: *Whitepaper* (2018).

[24] *zkSync V2 homepage*. https://v2.zksync.io/.