Layer 2 blockchains for videogames

Lapo Falcone
lapofalcone@gmail.com

July 24, 2023

Contents

Part I

Blockchain concepts and their limitations

In the last 3 decades internet has become a service used by virtually everyone. The framework which allowed the spread of the internet has been the *client-server* paradigm. Its success must be attributed to some important properties it guarantees.

First and foremost the relative ease of implementation and maintainability of such solutions compared to others, such as peer-to-peer architectures.

Moreover, client-server solutions can achieve high performance and efficiency, by handling a lot of requests thanks to the power of ad-hoc hardware and replication.

Another key characteristic of client-server is the fact that the owner of the service can fully control it, and change it based on his needs, as long as he follows the laws of the region he operates in and possible commitments he made with the users.

Despite all the positive properties of the client-server paradigm, there are some situations for which it is not suitable. What if we cannot trust the service provider? What if we want a service that is fully transparent for the users? What if we want a strong guarantee that data won't be modified?

In all these cases, and more, blockchain technologies are more suitable.

In the next sections we will briefly describe how a blockchain works, why it can be useful in videogames, and what are the main limitations that have prevented its use in the videogaming industry so far.

1 What is a blockchain

A blockchain, as the name suggests, is a chain of blocks. Each block contains transactions, which, in the simplest form, are exchanges of tokens between 2 users.

What actually creates the bond between 2 contiguous blocks of the chain, is the fact that each block also contains the hash of the previous one in the chain.

Blockchains are decentralized and distributed because the record of all the existing blocks (and therefore their relative *position* in the chain) are not stored in a single, central server owned by some company. Indeed they are stored by the participants of the blockchain itself.

Since all the existing blocks are stored somewhere[1], it is possible to reconstruct the tokens

[1]Not every node stores the full history of blocks, only full nodes do [Zohar, 2015].

owned by each user.

In order for such a distributed ledger to work in a useful way, there is the need for a method to ensure 3 basic properties:

- The transactions in a block must be valid.
- There must be an agreement among all the participants to the blockchain about the content (and therefore the order) of the blocks in the chain.
- Once a block has been added to the chain, it cannot be modified.

1.1 Transactions

Each user of the blockchain is identified only by a pair of private and public keys $< p_k, P_k >$.

A transaction is a *packet* of data like this[2]:

Sender P_k	Receiver P_k	Amount	Sender signature

For example, if the user with public key `0x1234` and private key `0x5678` wants to send 3 tokens to the user with public key `0xabcd`, the transaction will look like this:

0x1234	0xabcd	3	sign(0x1234, 0xabcd, 3, 0x5678)

Where `sign` is a function to sign the transaction with the private key of the sender.

The signature is crucial since it guarantees authentication, integrity and non-repudiation of the transaction.

Once a user creates a transaction, it sends it to the miners network.

1.2 Miners

The role of the miners is to guarantee the validity of the transactions, create blocks and agree on the content of each block in the chain.

When a miner receives a transaction, he propagates it to the other miners. Each miner keeps the transaction it receives in a pool, then he chooses[3] a number of transactions to include in the next block.

All the transactions in a block must be valid, therefore the miner must check that:

[2] In Bitcoins things are more complex than this. A user can spend an old transaction if he has a script to *redeem* it. To spend a transaction he must provide the script to *redeem* it and a script to re-lock it. Moreover, a user can merge many old transactions into a single one and can split the output by locking portions of it with different scripts. [Atzei and Bartoletti, 2013]

[3] Fees [Easley et al., 2018] play a key role in the way transactions get chosen.

- The signature of the transaction is valid.
- The transaction is not already present in a previous block of the blockchain.
- The sender actually owns enough tokens to complete the transaction.

When a block of transactions is created, the miner adds the hash of the previous block of the chain to it.

Now the miner must solve a computationally complex puzzle and then send the block to the other miners.

If the solution to the puzzle is correct, and all the transactions in the block are valid, all the other miners will accept the block as the last block of the chain.

1.3 Proof of work

If miners could add any block they create to the blockchain, it would be impossible to obtain a consensus between the miners on what blocks are actually part of the chain.

For example, it would be impossible to have a consistent state of the blockchain among all the participants, due to the CAP theorem [Gilbert and Lynch, 2012].

The blockchain wouldn't even be immutable, since it would be easy to replace an old block with another one.

For these and many other reasons, there must be a consensus mechanism among miners, in order to agree on what blocks are part of the chain.

The chronologically first consensus mechanism that has been used in a blockchain is proof of work (PoW) [Nakamoto, 2009]: after a miner creates a block of transactions, it has to add a *number* to it, such that the hash of the whole block starts with a predefined amount of zeroes. Since the hash is a non-invertible function, the only way to find a number with the property described above is brute force.

This simple mechanism solves the problems we discussed before. Indeed, now, it requires a lot of resources to modify an old block of the chain: not only a malicious miner should re-compute the hash of the block he wants to change, but also the hashes of all the blocks after it (since each block contains the hash of the previous one, which has changed).

Moreover, PoW reduces the number of blocks that can be created in a given interval of time, making it possible to reach a consistent state among the majority of miners[4].

In order to make PoW work with faster hardware, it is possible to collectively choose the amount of required leading zeroes in the hash of the new proposed block.

[4]It is still possible that 2 miners solve different blocks at roughly the same time. In this case, the blockchain will have a branch. After some time one of the 2 branches will be longer than the other, because it is unlikely that the 2 branches evolve at the same speed. At this point, the longer branch will be the one chosen by the miners, and the other one will be discarded.

Moreover, miners can add their address in a field of the block, so that they get rewarded for their work if they are able to create a new block.

1.4 Journey of a transaction

Here we make a summary of the steps that a transaction must follow to be included in the blockchain:

1. User `A` creates a transaction `T`, signs it and sends it to the miners network.
2. A miner `M` receives `T` and forwards it to the other neighboring miners.
3. `M` receives many other transactions, then creates a block `B` with only valid transactions, among which `T`.
4. `M` adds the hash of the last block of the chain and his address for the reward to `B`.
5. `M` starts trying different numbers until it finds a number that satisfies the PoW requirements.
6. `M` forwards the new block `B` to the other miners.
7. The other miners check the validity of `B`, if it is valid they will consider it the last block of the chain, else they will simply ignore it.

1.5 Blockchain programmability

Until now we have seen a blockchain where simple users can exchange some currency with plain transactions. Bitcoin is an example of a blockchain with these characteristics.

More recently a new kind of programmable blockchains has arisen, whose main exponent is Ethereum.

The core concepts of a programmable blockchain like Ethereum are similar to the ones of a *plain* one, with 2 main additions[5]:

Smart contracts Blockchain wallets controlled by an immutable script.

Account-based model Nodes keep the blockchain state in a *table* like the one in Table 1. A transaction can now be defined as an argument in a state transition function:

$$\mathcal{N}(state_{old}, transaction) = state_{new}$$

A smart contract is deployed via a transaction, where its source code is specified.

This means that the smart contract private key is never generated. A smart contract is

[5]Another evident difference of Ethereum is that it now uses proof of stake, but it is not important in this context.

Address	Balance	Nonce	Code	Storage
0x1234	3	149		
0xabcd	101	74	`if(bal > 0) then sendMoney(...)`	bal = 23

Table 1: Simplified example of the account-based state

indeed controlled by the source code, and not a private key. For example a smart contract cannot sign transactions[6].

Other users of the blockchain can interact with a smart contract by sending transactions where they specify the interface function to call and the necessary arguments.

When such a transaction is included in a block, the validator must also execute the code specified by the smart contract, whose execution, being deterministic, can be checked by the other validators. All validators should reach the same state.

Each smart contract has associated storage, which is part of the state of the blockchain and can be read and written during execution.

Smart contracts are immutable, which means it is not possible to change, or even delete, the code of the smart contract after it is deployed. This property is particularly relevant because it means that a user can be sure of what will happen if he interacts with a smart contract, by simply reading the smart contract code.

Another important aspect of smart contracts are execution fees. A smart contract will be executed by a validator, and the more complex the smart contract is, the more resources and time the validator will have to use.

For this reason, when a transaction is created, it is necessary to specify a maximum fee that the validator will earn by including the transaction in the block. Each instruction in a smart contract has an associated fee. If during the execution the total fee exceeds the maximum fee specified in the transaction, the execution will be aborted.

There are 3 main types of resources that require fees during the execution of a smart contract:

Computation The instructions executed.

Calldata The arguments passed in the transaction.

Storage The cost of writing to the smart contract storage.

[6]It is theoretically possible that someone generates a public-private key pair such that the hash of the public key corresponds to the smart contract address. In this case, a smart contract would also work as an EOA. This, although formally possible, is considered practically impossible.

1.6 Tokens

Smart contracts can be used to implement tokens, which are virtual assets that can be owned by the accounts.

Tokens differ from the currency of the blockchain (e.g. Bitcoin or Ether) because they are generally used in a closed environment and serve a specific purpose (for example a certain token could signify membership to a certain organization).

There exist 2 kinds of tokens:

Fungible tokens A token is indistinguishable from another one, and it can be fractioned.

Non-fungible tokens A token has a unique identifier and value, it cannot be fractioned.

2 Why blockchain can be used in videogames

In the previous sections, we have briefly seen how a blockchain works. Now we will see why it can be used for videogames.

Ownership of digital assets A videogame could reward the player with NFTs. This would be different than rewarding a player with a game-specific item, because the player would own the NFT independently from the game, and would even be free to sell it for fiat money.

Cross compatibility If players are rewarded with tokens, it would be easier for other games to integrate them. In this way, players could easily import and use items earned in one game in another one.

Transparency One of the key characteristics of blockchains is transparency. All transactions are public, therefore it can potentially be easier to detect cheaters in a competitive environment. Blockchain would also make it impossible for software houses to censor players.

Fairness Smart contracts could be used to create a fairground between the software house and the player base, for example by using smart contracts to implement loot boxes.

3 Why blockchain cannot be used in videogames

Blockchain can be used in videogames at different stages.

Most blockchain-powered videogames today are using blockchain as a storage for in-game rewards. This means that the execution of the game happens off-chain (generally on a server), but the server can decide to reward a player with on-chain tokens.

This paradigm is useful to implement the ownership of digital assets, and, to some extent, cross-compatibility, but it doesn't help with transparency and fairness.

To achieve transparency and fairness the game itself should be executed inside a smart contract on the blockchain.

There are 2 main issues with doing so:

Costs Usually the main loop of a videogame is quite complex. This means that the fees for executing the smart contract will be high.

Scalability and performance Blockchains have a much lower throughput than client-server solutions. This is an inherent characteristic of blockchains as we know it nowadays, and, as we have seen, a core principle of their functioning.

In recent years a new framework that can help with both issues has emerged: blockchain layer 2 [corwintines, 2023].

Part II

Blockchain layer 2

As we have seen in Why blockchain cannot be used in videogames layer 1 blockchains suffer from high fees and poor scalability.

This is a direct consequence of the blockchain trilemma [Musharraf, 2021], which states that we can have only 2 out of distribution, security and scalability. To try to tackle these issues layer 2 blockchains have been invented.

The core idea behind layer 2 is to transfer as much load as possible outside of the blockchain, and use the layer 1 properties only to finalize the operations that happened off-chain.

They try to do so by starting from a predefined state, usually stored in a smart contract, bundling up many transactions off-chain, and eventually reporting to the smart contract what happened off-chain [EthereumCommunity, 2022]. The smart contract can verify that some predetermined rules have been followed during the off-chain operation.

The off-chain transactions can be faster because they don't have to involve all the nodes of the blockchain to be verified, and can resort to simpler verification protocols that scale better and are more suited for certain applications.

In the next sections we will explore some of the most used layer 2 approaches, and we will find out why ZK rollups are arguably the best solution for videogames.

4 Off-chain state channels

The core idea behind state channels is to let channel participants perform transactions off-chain, and only submit 2 on-chain transactions to open and close the channel.

The lifecycle of a channel is made up of 3 phases [minimalsm, 2023]:

Opening To open a channel all the participants must add funds to a smart contract and agree to the initial state by signing it.

Usage Once a state channel is open, participants can start exchanging transactions off-chain. A transaction looks like this:

Old state	New state	Transaction to go from old state to new state	Nonce

A transaction to be considered final must be valid (i.e. following the rules stated in the smart contract, such as not spending more than what you own) and signed by all parties in the channel. If a transaction is valid, it cannot be reverted.

Closing Any participant can decide to send a transaction to the smart contract on layer 1. At this point, a challenge period starts, where the other parties can send a transaction to challenge the one proposed by the first participant.

For example, if user `A` wants to finalize the state channel with an old transaction, and user `B` has a newer transaction signed by all the parties, he can use the newer transaction to prove that user `A` is malicious. In this case user `A` would lose all the funds he added to the smart contract of the state channel.

The presence of the challenge period enhances the security of honest users: they can challenge malicious parties and they can withdraw their funds if other parties stop responding.

In any case, at the end of the challenge period, funds in the state channel are distributed based on the last valid transaction sent to the smart contract.

It is important to note that the state of a channel doesn't only hold currencies, but also any other possible variable. For this reason, state channels can be used to run computation off-chain.

4.1 State channels and videogames

As we have seen state channels rely on layer 1 blockchain to guarantee:

Liveness The smart contract for the channel is always available.

Security Honest users can always prevail, thanks to the layer 1 smart contract channel-closing policies.

Finality As soon as a transaction is signed by everyone, it is not reversible.

Moreover, since they allow off-chain computation, state channels don't suffer from the scalability and performance issues typical of blockchains.

On the other hand, since state channels computation runs off-chain, they do not guarantee the transparency and fairness properties we are looking for.

For example, let's imagine we have a state channel to play a game of chess against a computer, where you get a reward if you win. The state channel would be initialized and signed by the player and the server where the chess engine runs.

Then the player and the server could start exchanging transactions to move pieces. Let's now say that the player is one move away from checkmate, and the server refuses to acknowledge the transaction that delivered checkmate.

The player cannot send the checkmating transaction to the layer 1 smart contract, because the server could challenge it with the last co-signed transaction. And in this framework it would be impossible to prove that the server did it intentionally, maybe it simply didn't receive the transaction.

Therefore this solution wouldn't be much different than a simpler client-server solution with on-chain storage.

5 Sidechains

Sidechains are independent blockchains that can interoperate with the layer 1 blockchain.

Being separate, sidechains can use different consensus mechanisms than the main blockchain. On the one hand, this can help with performance, scalability and fees cost, but, on the other hand, the sidechain may lose some important guarantees of the layer 1 blockchain. For example, a sidechain could trade distribution for scalability to satisfy the blockchain trilemma and be scalable.

Sidechains can communicate with the blockchain via bridges.

Bridges are mechanisms that allow to connect different blockchains, allowing users to transfer assets between them [alemanaa, 2022].

There are many types of bridges, one of the most used kind locks or burns items in one blockchain and mints the same item in the other one, in order to simulate the transfer.

6 Rollups

Rollups are mechanisms where multiple transactions are performed off-chain and aggregated in a single *batch transaction* which is submitted to a smart contract on the layer 1 blockchain.

Off-chain transactions are executed in a layer 2 blockchain, where the validators (also called sequencers) also have the task to send batches of layer 2 transactions to the layer 1 smart contract.

Based on the mechanism that is used to determine the validity of the transactions in the batch on layer 1, it is possible to categorize rollups into optimistic and zero-knowledge.

6.1 Optimistic rollups

The main assumption of optimistic rollups is that the majority of the transactions inside a batch are valid, therefore sequencers can send the new state of the layer 2 blockchain to the layer 1 smart contract, without any validity proof.

The caveat is that the layer 1 smart contract will wait a challenge period (usually 7 days) before accepting the new state. During this period other participants to the layer 2 blockchain can send fraud proofs to challenge the new state sent by the sequencer.

This mechanism enables better scalability and cost efficiency than in layer 1 blockchains, since smart contracts can be executed on layer 2, and the layer 1 fees are split among all the transactions in a batch.

6.1.1 Entering an optimistic rollup

To enter a rollup a user must deposit some assets (generally tokens) in the layer 1 rollup smart contract. These assets are then transferred to the layer 2 blockchain via a bridge to a specified address.

At this point, the user must wait until the transaction made by the bridge to the specified address gets processed by the sequencer and sent in a batch to the layer 1 smart contract.

6.1.2 Exiting an optimistic rollup

To exit a rollup a user must send an exit transaction to the layer 2 sequencer. The sequencer will execute the transaction by burning all the assets of the user (that should be included in the exit transaction), and then include this transaction in a batch.

At this point, the user must generate a cryptographic proof (generally a Merkle proof) to show that the exit transaction was inside the last batch, and that the layer 2 account that submitted such transaction belongs to the user.

Now it is possible to send the proof to the layer 1 smart contract, along with an address where to send the withdrawn assets.

6.1.3 L2 - L1 interaction

The sequencers store the state of the rollup as a Merkle tree (see section Merkle trees). The root of such a tree is also stored in the layer 1 smart contract. [EthereumCommunity, 2023].

When a sequencer wants to send a batch to the layer 1 smart contract, it must send a transaction containing:

- The Merkle root of the old state.
- The Merkle root of the new state.
- An highly compressed summary of all the transactions in the batch (containing also the input data of the transactions).
- The Merkle root of the transactions in the batch.

The layer 1 smart contract receives the transaction and compares the Merkle root of the old state with the one in its storage. If they match, the Merkle root in the storage is updated to the Merkle root of the new state.

The highly compressed list of all the transactions and their inputs won't be part of the layer 1 state, but it will simply be stored as calldata, for cost reasons.

The Merkle root of the transactions in the batch is necessary in order to let other participants prove that their transaction is part of the batch in an efficient way (for example when a user wants to withdraw assets from the rollup).

At this point, the challenging period begins, and other users can send fraud proofs. 2 main proving schemes can be adopted:

Single round In this scheme all the transactions' inputs being part of a batch must be stored (as calldata) in the layer 1 smart contract. Since the smart contract has all the transactions and their inputs, it can replay them and verify whether the new state Merkle root matches with the one sent by the sequencer.

Multi round This proving mechanism is more complex but also more efficient. In this case, only the Merkle root of the summary of transactions is required to be stored as calldata.

There are 2 ways in which the sequencer may lie: including the wrong transactions and computing a wrong final state.

In the first case, it is simple to prove the sequencer is malicious, since it suffices to provide the right transactions, which are public on Starknet.

In the second case, instead, the process to prove fraud consists of a dialogue between the sequencer (asserter) and the challenger, supervised by a layer 1 smart contract. At each step the asserter must split the computation performed in a batch into 2 equal halves and state the output[7] of each one of the 2 halves. The challenger can decide which half he wants to challenge. This process goes on until only a single instruction remains in the asserter computation. At this point, both the asserter and the challenger must send their prediction for the outcome of that instruction, which is executed by the layer 1 smart contract, which can easily find out who is lying. This process can be visualized in Figure 1.

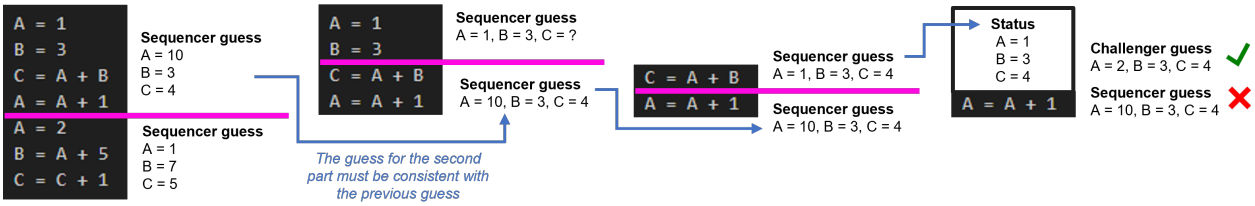

Figure 1: The multi round fraud proof mechanism visualized.

It is interesting to note that even a single honest participant to an optimistic rollup has the power to stop malicious activities by any number of other nodes, thanks to the security guarantees provided by the layer 1 smart contract.

6.1.4 Optimistic rollups and videogames

Optimistic rollups are a step in the right direction towards the use of the blockchain in videogames.

[7] With output here it is intended the hash of the full state of the computation till this point.

Indeed they do not suffer from the transparency and fairness problems we discussed in section State channels and videogames, and, at the same time, they maintain more security and decentralization guarantees than sidechains, since they maintain a strict bond with the layer 1 blockchain.

The main problem of the majority of the implementations of optimistic rollups are:

High fees Even if fees are lower than using a layer 1 blockchain, the need to send all the transactions and their inputs in a batch (although compressed) as calldata, isn't enough to lower the fees enough for adoption in the videogaming world.

Withdrawal time As we have seen in section Exiting an optimistic rollup, it takes a very long time (usually 7 days) to be able to withdraw the assets from an optimistic rollup. This doesn't pair well with the cross-compatibility property, since different games will most likely run on different optimistic rollups implementations.

6.2 Zero-knowledge rollups

Zero-knowledge rollups are a layer 2 scaling solution that operates similarly to optimistic rollups. Therefore, as in optimistic rollups, the layer 2 sequencers receive various transactions, execute them and only update the layer 1 smart contract when a batch of transactions has been executed.

As in optimistic rollups, this mechanism improves scalability and performance of the blockchain, and splits the layer 1 fees among all the transactions in a batch.

The core difference between optimistic rollups and ZK rollups is in the way the transactions batch update gets validated in the main blockchain.

Sequencers of ZK rollups must send a zero-knowledge proof along with the changes to the old state. The zero-knowledge proof is a cryptographic demonstration that the proposed changes to the state are actually the result of executing all the transactions in the batch.

This change makes it so that it isn't required to store a compressed summary of the transactions on the layer 1 chain as calldata, greatly lowering the fees needed. Moreover, the layer 1 smart contract can immediately verify whether the changes are valid, so the challenge period can be completely removed.

6.2.1 Entering a ZK rollup

The process to enter a ZK rollup is very similar to the one described in Entering an optimistic rollup.

The user must deposit some assets in the layer 1 smart contract, a bridge will transfer the assets to the rollup. After the sequencer adds the transfer transaction to a batch, the user will be able to use his assets in the ZK rollup.

6.2.2 Exiting a ZK rollup

If the process to enter a ZK rollup is basically the same as the one to enter an optimistic rollup, the exiting process shows some differences caused by the different way in which transaction batches are verified in layer 1.

To exit a ZK rollup a user must initiate an exit transaction by sending all his assets to a specific burn address.

As soon as this transaction is added to a batch by the sequencer, the user must create a Merkle proof to prove the presence of the exit transaction in the batch.

Then he can send a withdrawal request to the layer 1 smart contract with the following information:

- The Merkle proof.
- The exit transaction.
- The Merkle root of the transaction batch containing the exit transaction.
- A layer 1 address where to deposit the withdrawn assets.

The layer 1 smart contract can hash the transaction data and use the Merkle proof to verify that such transaction is indeed part of the Merkle root of the batch. If this is the case the smart contract will deposit as many assets as those burned in the exit transaction to the specified layer 1 address.

This process has a way lower latency than the one used for optimistic rollups, because the challenging period doesn't exist in ZK rollups.

6.2.3 L2 - L1 interaction

Similarly to optimistic rollups, the sequencers must compute the Merkle tree of the state of the rollup, and send its root along with:

- The Merkle root of the old state.
- The validity proof.
- The Merkle root of the transactions in the batch.

to the layer 1 smart contract.

When the smart contract receives the batch, it will execute the validity proof, check the old state and, if everything matches, update the rollup Merkle tree root in its storage [d1onys1us, 2023].

6.2.4 Intuition on the validity proofs

Validity proofs are complex cryptographic objects that allow to demonstrate the truthfulness of a computation-based statement in way less time than executing the full computation.

In the case of ZK rollups, the statement to prove is that, given the computation deriving from the list of transactions in the batch and the old state, we get to the proposed new state.

For costs reasons, it is not possible to run the full computation of the batch on layer 1 (it would defeat the purpose of ZK rollups), therefore we want to translate this problem into an easier-to-compute mathematical problem, where the transactions become values to plug in a set of equations, so that these mathematical equations are satisfied.

One way to do so is to use the PLONK scheme [Buterin, 2019], or ZK-STARK [Berentsen et al., 2022].

6.2.5 ZK rollups and videogames

ZK rollups share a lot of features with optimistic rollups:

Improved scalability ZK rollups help with blockchain scalability, by operating on a smaller blockchain and only communicating with the layer 1 blockchain in batches.

Reduced layer 1 fees ZK rollups share the fee of the layer 1 transactions among all the transactions in a batch.

Transparency and fairness In a ZK rollup context, operations still happen on a public blockchain with a consensus algorithm.

Decentralization and security These 2 are directly inherited from the layer 1 blockchain, thanks to validation proofs.

Moreover, ZK rollups do not suffer from the 2 main problems identified in Optimistic rollups and videogames:

~~**High fees**~~ ZK rollups further reduce the fees compared to optimistic rollups, since layer 2 transactions inputs must not be stored as calldata.

~~**Withdrawal time**~~ ZK rollups, thanks to validity proofs, do not have a challenge period. Exiting a rollup takes as much time as waiting for the sequencer to update the layer 1 smart contract, and the smart contract transaction to be included in a block on layer 1.

For these reasons, ZK rollups are, in the majority of cases, the most suited layer 2 blockchain type to implement videogames.

Part III

Starknet

Starknet is one of the most successful implementations of ZK rollups. It uses Ethereum as layer 1.

In the next sections the core concepts of Starknet will be explained. Then it will be explained each step to set up Starknet on a Linux machine. Last, an overview of Cairo, the programming language to write smart contracts, will be presented.

7 Account abstraction

As we have partially seen in section Blockchain programmability, Ethereum has 2 different kinds of accounts:

EOA Externally owned accounts are controlled by users via their private key. These accounts can initiate transactions by signing them with the private key.

Contract accounts Contract accounts are instead controlled by code. The private key for these accounts is not generated (and it is practically impossible to compute), therefore they cannot sign transactions, but can only respond to transactions with the specified source code.

This means that there isn't much flexibility left for the users: they must have the private key and use it to generate transactions.

As the blockchain gets more and more diffused, the need for more flexibility and user-friendliness has arisen. For example:

Access recovery Nowadays if you lose the private key, you lose access to your account. It is not possible to recover it.

Multi approval transactions In some cases it is important that a transaction is approved by more than one person. This is not *natively* possible with only a single private key.

Transaction limits It could be handy if it was possible to limit the amount of transactions in a given time interval, or the maximum amount transferable.

At the moment, in Ethereum, all of these features are achieved via the use of smart contracts.

For example, in order to be able to have multi-approval transactions, it is possible to set up a smart contract with a method to move currency only if it is invoked by a certain number of predefined addresses.

However, this way of solving the described problems is cumbersome, as it requires deploying

a smart contract and then using a different account to interact with it via transactions.

It would be handy if we had only one kind of account, controlled by code but at the same time able to generate transactions.

This is a simplification of the Starknet account model: everything is a smart contract.

In other words, your account balance is now controlled by code, and not by your private key. Your private key is only used to initiate transactions to call other smart contract functions, or to deploy a new smart contract. But it cannot be used to directly modify Starknet state (for example by sending an amount of ETH to another account).

The account balance holds STARK tokens (the native currency), ETH and other Ethereum common tokens, such as ERC-20.

The account balance, along with the account contract code and the account contract storage, form the Starknet state.

In Starknet, if a smart contract has the following interface [StarknetCommunity, 2023], it is then considered an account contract (which, technically, is no different than any other smart contract):

constructor Used to initialize the account contract.

validate_deploy Used when deploying this account.

validate_declare Used when declaring a new contract.

validate A method to check the validity of a requested transaction. For example, it can be used to compare the signature on the sender's transaction with a public key stored in contract storage. The key point is that this method can check any criterion. In this way it is possible to have multisig accounts, or ways to change the *password* to make transactions with this account.

execute Actually executes the requested transaction, after having checked with `validate`.

When a new user wants to create a new account on Starknet, it will use counterfactual deployment. The user will generate the canonical key pair, then will fund his account, and, eventually, will perform a transaction to deploy a smart contract (in particular an account contract) on the address[8] relative to his public key. The funding must be done prior to the deployment of the account contract to be able to pay the fee of the deployment transaction.

8 Starknet actors

Starknet organizes its participants into various roles [StarknetCommunity, 2022]. Each role carries out one of the tasks needed to make ZK rollups work.

[8] With address we refer to an hexadecimal number generated starting from the private key. For example the hash of the private key.

At the moment some of the roles present in Starknet are centralized for development reasons, but the core concepts of each role do not make any centralization assumption, and in the future will be decentralized.

8.1 Sequencers

Sequencers can be compared to miners. Their tasks are:

Aggregation They have to collect and store in a pool transactions from users.

Batching They have to group transactions together into batches, and decide when to close a batch. This decision can be taken based on the number of transactions received, or on the time elapsed since the last batch.

Execution Sequencers have to validate and execute the transactions to update the state.

At this point, sequencers have to pass the block they've created to provers.

8.2 Provers

Provers must supervise the work of sequencers and communicate with the layer 1 smart contract:

Execution check Provers execute a second time some sampled transactions they received in a block, to check sequencers work.

Proof generation Provers have the task to generate the zero-knowledge proof to demonstrate a correct handling of the execution of the transactions.

Communicating with layer 1 Finally they have to send the batch of transactions and the proof to the layer 1 smart contract.

8.3 Nodes

Nodes are the auditors of Starknet. Their tasks are:

Intermediaries A node is the intermediate actor between a client and the sequencer.

Replaying Nodes can replay old transactions to ensure their correctness.

Storage Nodes can store the blocks produced by the sequencers and provers, to guarantee availability in the layer 2 blockchain.

Checking proofs Nodes can also check the proofs provided by the provers. This is a cheaper way compared to replaying transactions to guarantee the honesty of the provers.

9 Transaction journey

In this section it will be summarized the various steps a transaction must go through to be accepted on the layer 1 blockchain:

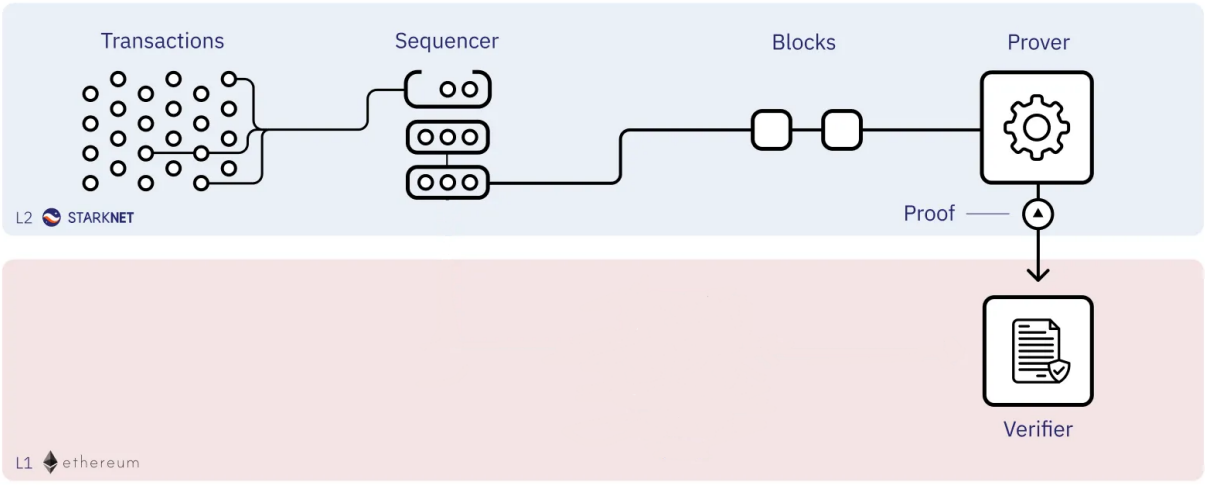

Figure 2: The journey of a transaction.

Creation A Starknet user can create a transaction by adding an account-specific nonce, and signing it. After this the transaction gets sent to a specified node.

Reception The node will receive the transaction, apply some preliminary checks (mostly syntactic) and then forward it to a sequencer.

Acceptance on layer 2 As soon as the sequencer receives the transaction, it validates and executes it. When a transaction is executed, Starknet state changes immediately and the transaction is considered accepted on layer 2. However the block isn't emitted immediately: the sequencer will batch together multiple transactions before emitting a block.

Proof generation The block is received by a prover, which verifies again each transaction and generates the zero-knowledge validity proof and sends it to the Ethereum smart contract.

Acceptance on layer 1 The Ethereum smart contract verifies the zero-knowledge proof. At this point, the transaction is considered accepted on layer 1.

10 Starknet setup

Users can interact with Starknet via a CLI available for Linux and MacOS. We are going to see how to create a smart contract and interact with it in Linux:

10.1 Prerequisites

We are going to work on a clean installation of Ubuntu 22.04.2 LTS, installed on a virtual machine in VirtualBox.

10.2 Install Python3.9

As of July 2023, Starknet CLI can work with Python3.9, but not with Python3.10 yet. Since Python3.10 is the default version on Ubuntu 22.04.2, we must install Python3.9 and create a virtual environment for it.

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa #Python3.9 is not directly available
sudo apt install python3.9

python3.9 --version #should return Python 3.9.1x
```

Then we have to install some Python3.9 modules:

```
sudo apt-get install python3.9-distutils #necessary to install some other modules
sudo apt-get install python3.9-dev #necessary to install some other modules
sudo apt-get install python3.9-venv #to create virtual environments
```

Now we can create the virtual environment and activate it:

```
python3.9 -m venv ~/cairo_venv
source ~/cairo_venv/bin/activate
#next lines in the terminal should start with (cairo_venv)
```

From now on all commands will be run inside the virtual environment, where the default version of Python is 3.9.

10.3 Install Cairo and Starknet

First, we have to install these dependencies:

```
sudo apt install -y libgmp3-dev #for multiple precision arithmetic
sudo apt install git #will be used later on
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh #Rust
pip install wheel #install if the installation of fastecdsa fails
pip install ecdsa #cryptographic algorithm
pip install fastecdsa #cryptographic algorithm
pip install sympy #symbolic math
```

Now we can install the Starknet CLI:

```
pip install cairo-lang
starknet --version #should return 0.12.0
```

And the Cairo compiler[9]:

[9] I found out that there is a bug preventing using mappings (`LegacyMapping`) inside contracts. A workaround is to comment out line 71 at `.cairo/crates/cairo-lang-sierra-gas/src/gas_info.rs`. The underlying problem has something to do with a check on the estimated gas consumed by the contract.

```
git clone https://github.com/starkware-libs/cairo/ .cairo #we download the repo to ~/.
        cairo
cd .cairo #we move to the newly created folder
git checkout tags/v1.1.1 #we move to the release branch, v1.1.1 is the last supported
        version on Starknet
cargo build --all --release #we build the compiler via Cargo
cd .. #we move back to the home folder
```

10.4 Setting up environment variables

To set permanent environment variables we have to append content to `~\.bashrc` file:

```
nano ~/.bashrc #open .bashrc with a text editor
```

Now write these variables at the end of the file:

```
export STARKNET_NETWORK=alpha-goerli #the Starknet network we will use (in this case the
        testnet)
export STARKNET_WALLET=starkware.starknet.wallets.open_zeppelin.OpenZeppelinAccount #the
        service to create account contract
export CAIRO_COMPILER_DIR=~/.cairo/target/release/ #where Cairo compiler is placed
export CAIRO_COMPILER_ARGS=--add-pythonic-hints #Cairo compiler options
```

Now press `Ctrl + X`, then `y` and finally `Enter`.

Now close and re-open the terminal (don't forget to re-activate the virtual environment when you re-open the terminal).

You can test if the environment variables we added exist with the command `printenv`.

10.5 Creating an account

To create a Starknet account:

```
starknet new_account --account myAccountName
```

This will generate the address and the public-private key pair of the account, which will be printed in the terminal and also saved to `~/.starknet_accounts/starknet_open_zeppelin_accounts.json`.

Now we have to add some funds to the account, so that we can use them to deploy the account contract.

To do so we can use any Starknet faucet, for example: https://faucet.goerli.starknet.io/.

Then we can check on https://starkscan.co/ that the transaction with the funds is at least in *Pending* state.

At this point, we can deploy the account:

```
starknet deploy_account --account myAccountName
```

This will return the transaction hash: we must wait until the transaction is accepted on layer 2.

10.6 Deploying and interacting with a smart contract

To deploy a smart contract, first, we must have some Cairo code. There are some examples online, such as https://github.com/starknet-edu/deploy-cairo1-demo.

Let's say we have the code for the contract at `~/src/myContract.cairo`.

First, we have to compile the Cairo code into Sierra bytecode:

```
~/.cairo/target/release/starknet-compile ~/src/myContract.cairo ~/src/myContract.json
```

Then, we have to declare the smart contract. This will create a *Class* object in Starknet, which is a template containing our code:

```
starknet declare --contract ~/src/myContract.json --account myAccountName
```

Now we have to wait until the transaction declaring the contract is accepted on layer 2.

Eventually, we have to deploy the contract. A smart contract will be created starting from the *Class*.

```
starknet deploy --class_hash HASH_OF_THE_PREVIOUSLY_CREATED_CLASS --inputs
        CONSTRUCTOR_ARGS --account myAccountName
```

As soon as the deploying transaction is accepted on layer 2, we can interact with the account:

```
starknet invoke --address SC_ADDRESS --function FUNCTION_NAME --inputs FUNCTION_ARGS --
        account myAccountName
```

And we can also interact with the read-only functions of the contract via:

```
starknet call --address SM_ADDRESS --function FUNCTION_NAME --inputs FUNCTION_ARGS --
        account myAccountName
```

11 Cairo

Cairo is the programming language in which we can write smart contracts in Starknet.

Cairo is a language in continuous development[10]. At the moment, on Starknet, it is possible to write contracts in Cairo 0.x (which is now deprecated) and Cairo 1.x. Soon it will be possible to write contracts in Cairo 2.x.

The implementation part of this research has been compiled with the Cairo 1.1.1 compiler, therefore, in the next sections, we will see some concepts of Cairo 1.

[10]For this reason it is very difficult to find updated information online. Even the official Starknet website uses Cairo 1 in some sections, and Cairo 2 (which isn't supported yet), in other sections.

11.1 General concepts

Cairo is a Turing complete programming language, but in the next sections we will only cover basic concepts, useful to be able to understand the implemented smart contract. For a deeper guide it is possible to look at [CairoCommunity, 2023].

In Cairo it is possible to use the standard control flow constructs, such as `if` and `for` statements:

```
if x > y {
    \\...
}
```

Cairo has a simple native type system. The basic types are: `u8`, `u16`, `usize`, `u64`, `u128`, `u256` which are basic integer types, and `felt252` which can hold any type, but has peculiarities during the arithmetic division operation (is is not suggested to use it).

In Cairo it is possible to use the `let` keyword to declare a variable, and then use it.

It is not possible to perform arithmetic operations on mixed types, but it is possible to use the `into()` function to perform casts.

It is possible to perform casts only to bigger data types. In case of a cast to a smaller data type, and given that the variable to cast fits the smaller type, it is possible to use `try_into()`, which returns an `Option<T>` data type. `unwrap()` is used to return `T` in case the conversion succeeded.

```
let x: u8 = 9_u8;
let y = 1000_u128;
let z = x + 5_u8;
let a: u128 = x.into();
let b: u16 = x.try_into().unwrap();
let c: u8 = x.try_into().unwrap(); //error, 1000 doesn't fit in a 8-bit integer
```

It is possible to import functions and data types via `use`.

Some useful imports are:

```
use starknet::get_execution_info; //to get info about the environment of the smart
        contract execution (e.g. calling user, block hash, ...)
use::starknet::ContractAddress; //type for addresses (a wrapper of felt252)
use traits::Into; //to use into()
use traits::TryInto; //to use try_into()
use option::OptionTrait; //to use unwrap()
```

Traits are comparable to interfaces in other programming languages.

Functions have this syntax:

```
fn foobar(par1: type, par2: type, ...) -> return_type {
    \\...
}
```

11.2 Storage

Starknet smart contracts can hold state. In order to create state variables in Cairo, we have to declare a structure:

```
struct Storage {
    foo: u8,
    bar: u256,
    mapping: LegacyMap<ContractAddress, u8>,
}
```

Here we declared an 8-bit integer (`foo`), a 256-bit integer (`bar`) and a dictionary (`mapping`) holding `ContractAddress` as key and `u8` as value.

Then, we can access storage variables via `read` and `write` calls, like this:

```
let x = mapping::read(0x5e6f);
foo::write(foo::read + x);
mapping::write((0x1a2b, 149_u8));
```

It is important to note that it is possible to use a certain type inside the storage, only if it implements a trait with `read` and `write`.

11.3 Contract sections

When writing a Starknet contract, it is important to understand how to structure a Cairo program so that it can be parsed by Starknet.

The file must start with `#[contract]`, followed by the declaration of the module where the contract functions will live.

At the beginning of the module, usually, there are the imports used in the contract.

Then there is the storage structure declaration.

Eventually there are the functions, that can be decorated with the following labels:

#[**constructor**] The next function will be the constructor of the smart contract. The constructor will be called when the smart contract is being deployed, and can initialize state variables.

#[**external**] Marks functions that modify the storage of the contract. Such functions can be called via transactions.

#[**view**] Marks functions that only read contract state parameters. Such functions, since they don't modify the state, can directly be queried, without a transaction.

#[**event**] Marks functions used to emit an event. An event is a mechanism that can be used to communicate information outside of the contract.

Part IV

Implementation: immutable loot boxes

In the next sections it will be presented a Starknet smart contract that aims to show one possible application of the blockchain to gaming.

In section Why blockchain can be used in videogames we listed 4 videogame-related properties that can be achieved by using a blockchain.

In [Marchesi and Bruschi, 2022] thesis they proposed 2 projects that aimed to show how it is possible to use smart contracts to run simple videogames on Starknet:

Flappy birds This implementation aimed to show how to implement a continuous-time game on-chain, where the initial seed was randomized based on player identity to prevent the user to copy other users' solutions.

Snake This implementation aimed to show how to implement a discrete-time game, where only the first player to send a specific solution could redeem a reward. This approach was subject to the fact that available solutions decreased with time, and that it was a source of miner extractible value.

We propose instead an implementation to show how it is possible to use blockchain in videogames to enhance transparency and fairness.

The idea originated from [Carvalho, 2021], but in this solution they proposed an implementation based on Ethereum layer 1. We propose a simple implementation based on Starknet layer 2.

This will allow us to have all the important fairness and transparency properties guaranteed by layer 1 immutability and publicity, and at the same time have lower latency and fees.

The full project can be found on GitHub:
`https://github.com/Lapo9/Layer-2-Blockchains-for-Videogames`

12 Problem statement

Loot boxes are a widely used reward mechanism used in videogames, especially in recent years.

When loot boxes are part of a videogame, the player is presented with the possibility of purchasing an item (the loot box) which can be opened and give as a reward one (or more) in-game objects that are part of a collection.

At the moment of the purchase of the loot box, the player only knows what are the objects part of the collection, but not the random objects he will get. And of course some objects will be more valuable than others.

For this reason, in recent times, loot boxes started being considered akin to gambling in some countries, for example in Belgium. This is due to the fact that loot boxes share many features of standard gambling, as I discussed in this presentation: [Falcone, 2023].

The main reason why they are often compared to gambling is the fact that they can be classified as intermittent rewards with variable ratio [Skinner, 1950], which is a technique to keep humans engaged in an activity. In short, when opening loot boxes, you know you could get a desired prize, but you don't know if and when you will get it.

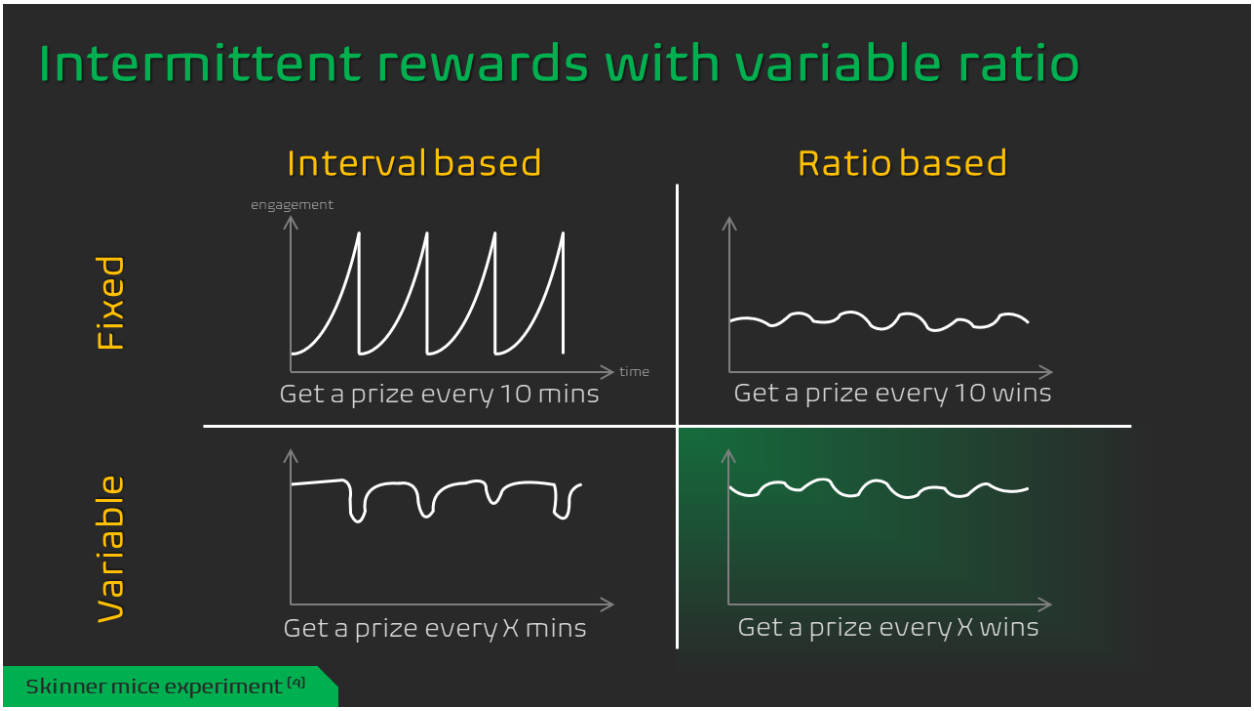

Figure 3: Comparison of different reinforcement methods from [Falcone, 2023].

In order to combat this dangerous phenomenon, in many jurisdictions, software houses are now required to clearly show the probabilities to get each possible item part of a loot box.

The main issue of this measure is the fact that legal bodies lack the tools, both technical and juridical, to enforce the rules. Loot boxes execution happens on private servers, therefore it is difficult to analyze. At the moment the only practical way is to analyze the outcomes, rather than the code execution, but to get an accurate result many samples are required (and they wouldn't even hold well in a trial).

Moreover, software houses tend to keep the engagement high with other means, for example

by letting famous internet content creators have more chances to draw valuable items.

Both these problems would be resolved by moving loot boxes code execution to a public blockchain. Indeed, in this scenario, the player can read and understand the loot box source code, and be sure it will be executed without modifications.

After having opened the loot box, the player could look up the transaction via an external blockchain explorer, and be sure that he got awarded in-game with the reward the loot box contract actually extracted.

13 The smart contract

The smart contract is a very simple implementation of the ideas we presented in section Problem statement.

```
#[contract]
mod LootBox {
use starknet::get_execution_info;
use starknet::ContractAddress;
use traits::Into;
use box::BoxTrait;
use option::OptionTrait;

//the storage simply contains mappings associating the user and the amount of each
        assets they have
struct Storage {
    usersTokens: LegacyMap<ContractAddress, u256>,
    usersGems: LegacyMap<ContractAddress, u256>,
    usersGold: LegacyMap<ContractAddress, u256>
}

#[constructor]
fn constructor(masterAddress: ContractAddress) {
    usersTokens::write(masterAddress, 1000000); //make the owner rich
}

#[event] //signals loot box opening (currency: 0 = gems, 1 = gold)
fn lootBoxResult(user: ContractAddress, currency: u8, amount: u256, remainingTokens:
        u256){}

#[event] //signals error
fn notEnoughTokens(user: ContractAddress, remainingTokens: u256){}

#[event] //tokens sent
fn tokensSent(sender: ContractAddress, receiver: ContractAddress, remainingTokens: u256)
        {}


#[external] //extracts a random item and adds it to the player assets
fn buyLootBox() {
    let userAddress = get_execution_info().unbox().caller_address; //get user address
    let availableTokens = usersTokens::read((userAddress)); //extract how many tokens
            the user has

    //executes contract only if the user has enough tokens (assert is very buggy on this
             version of Starknet, it would be a better option)
    if availableTokens > 0 {
        //pay 1 token
        usersTokens::write(userAddress, usersTokens::read((userAddress)) - 1);
```

```
            //here we extract a pseudo random number, based on the hash of the incoming
                transaction
            let transactionHash = get_execution_info().unbox().tx_info.unbox().
                transaction_hash; //get the hash of the transaction
            let transactionHash256: u256 = transactionHash.into(); //cast the hash to a 256
                integer
            let random = transactionHash256 & 0xfff; //keep only the last 3 digits (range
                from 0 to 4096)

            //25% chance of getting gems (if the last hex digit (0-15) is less than 4
            if (random & 0xf) < 4 {
                usersGems::write(userAddress, usersGems::read((userAddress)) + random); //
                    add gems to account
                lootBoxResult(userAddress, 0, random, availableTokens-1);  //signal event
            }
            else {
                usersGold::write(userAddress, usersGold::read((userAddress)) + random); //
                    add gold to account
                lootBoxResult(userAddress, 1, random, availableTokens-1); //signal event
            }
        }
        else {
            notEnoughTokens(userAddress, availableTokens); //signal error (user doesn't have
                 enough tokens)
        }
    }

    #[external] //sends the specified amount of tokens to the specified account
    fn sendTokens(receiverAddress: ContractAddress, amount: u256) {
        let userAddress = get_execution_info().unbox().caller_address; //get user address
        let availableTokens = usersTokens::read((userAddress)); //extract how many tokens
            the user has

        if availableTokens < amount {
            notEnoughTokens(userAddress, availableTokens); //signal error (user wants to
                send more tokens than what he has)
        }
        else {
            usersTokens::write(userAddress, availableTokens - amount); //remove tokens from
                user
            usersTokens::write(receiverAddress, usersTokens::read((receiverAddress)) +
                amount); //add tokens to the receiver
            tokensSent(userAddress, receiverAddress, availableTokens - amount); //signal
                transfer
        }
    }

    #[view] //returns the assets of the specified player
    fn getUserAssets(userAddress: ContractAddress) -> (u256, u256, u256) {
        let tokens = usersTokens::read((userAddress));
        let gems = usersGems::read((userAddress));
        let gold = usersGold::read((userAddress));
        return (tokens, gems, gold);
    }

    #[view] //returns how many tokens a loot box costs
    fn getLootBoxPrice() -> u256 { 1 }
    }
```

Listing 1: Loot box smart contract code. Can also be found at https://github.com/Lapo9/Layer-2-Blockchains-for-Videogames/blob/master/LootBoxSmartContract/LootBoxSmartContract.cairo

The storage of the contract holds `<user, balance>` pairs for each item part of the loot box. In particular, the loot box can contain gems and gold. Tokens are the currency used to open loot boxes.

Opening a loot box By reading the `buyLootBox` function, it is possible to understand the probabilities of getting each item.

In particular, if the last hexadecimal digit of the hash of the incoming transaction is less than 4 (i.e. 0, 1, 2, 3), then gems are awarded. In the remaining 75% of cases, gold is awarded.

The amount of gold or gems awarded is chosen by truncating the incoming transaction hash after the 3 least significant hexadecimal digits. Therefore it is possible to get an amount that ranges from 0 to $16^3 - 1 = 4095$ gems or gold.

For example, if the incoming transaction hash is $0x2fb8...ca29$ then gold is awarded because $9 \geq 4$, and the amount is $a29_{hex} = 2061_{dec}$.

Viewing balances A videogame using this technique for loot boxes, needs to be able to read back the items players have. The `getUserAssets` function serves exactly this purpose. Balances, in this case, are public.

14 The client

It is possible to interact with this smart contract by directly using the Starknet primitives we described in section Deploying and interacting with a smart contract, but we also proposed a videogame-like interface written in Python[11].

We used a module called `starknet_py`[12] to interact with Starknet, and `pySimpleGui`[13] to create the graphically appealing[14] user interface, shown in Figure 4.

The main `starknet_py` objects are:

FullNodeClient A facade to interact with a full node of Starknet. It can be used to send and read transactions.

Account An helper object representing the user account. It can be used in tandem with `FullNodeClient` to send transactions to Starknet.

Contract An helper object encapsulating a contract. It makes it easy to invoke contract functions and get back the result of the invocation.

[11] The proposed client is very simple, it doesn't have error handling and expects correct input. It is good for a demo.

[12] https://pypi.org/project/starknet-py/

[13] https://pypi.org/project/PySimpleGUI/

[14] It ended up being less graphically appealing than expected...

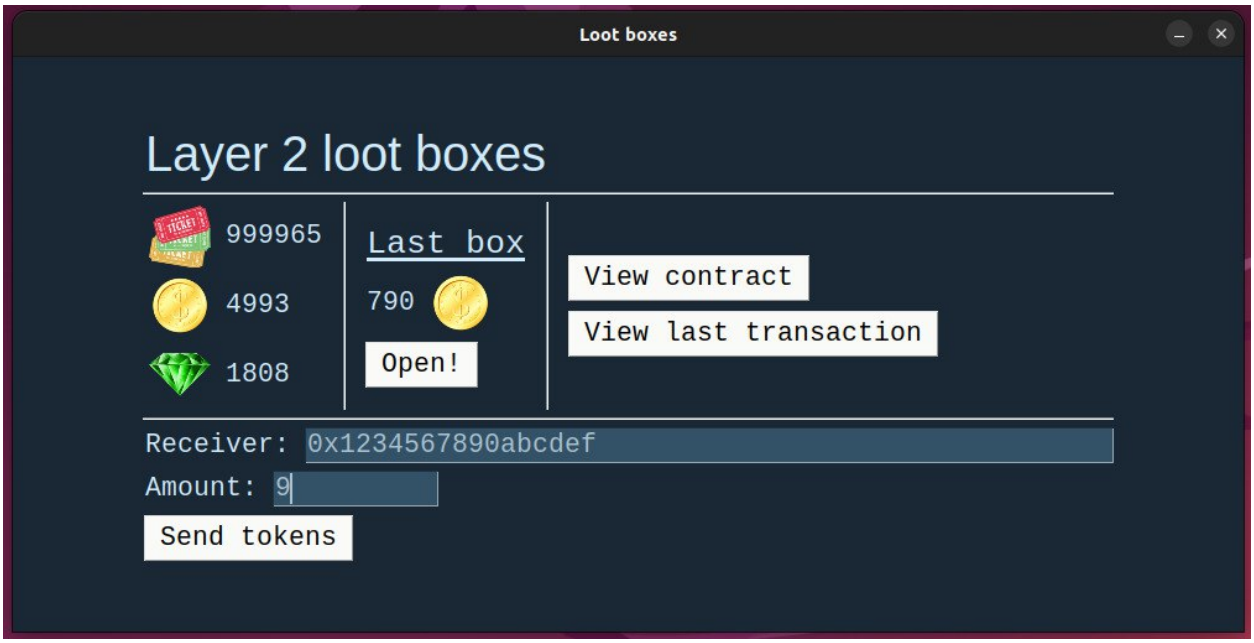

Figure 4: The graphically appealing user interface.

Interacting with Starknet takes time. For this reason most of `starknet_py` methods have an `async` version.

One of the most challenging part has been to fetch the transaction of a smart contract invocation. `starknet_py` proposes `Contract.wait_for_completion` method to do so, but this method waits for the transaction to be accepted on layer 2. In our case we just needed to wait for the transaction to be pending on layer 2, therefore we resorted to a polling loop, shown in Listing 2:

```
    invocation = contract.functions["buyLootBox"].invoke_sync(max_fee=int(1e15))

    #keep looking for the transaction hash
    transactionFound = False
    while (not transactionFound):
        try:
            event = node.get_transaction_receipt_sync(invocation.hash).events[0] #try to get
                    the transaction from the node
        except:
            time.sleep(1) #if you can't get it, wait 1 second and retry
        else:
            transactionFound = True
```

Listing 2: Polling loop.

As it is possible to note from Listing 2, we used the synchronous version of the methods of `starknet_py`. Indeed we used a `pySimpleGui` functionality (`Window.perform_long_operation`), to perform asynchronous tasks without blocking the GUI.

The functions we implemented in the GUI are:

Open loot box To open a loot box it suffices to click on the `Open!` button. After this, the button will be disabled until the Starknet transaction is performed on layer 2. At this point the result of the loot box opening will be displayed, the balances updated and the last transaction hash will be inserted in the `View last transaction` button.

Send tokens To send tokens to other players, it suffices to insert their address and the amount of tokens to send, then click on `Send`. After this, the button will be disabled until the Starknet transaction is performed on layer 2. Then, the tokens balance will be updated and the last transaction hash will be inserted in the `View last transaction` button.

View last transaction This button opens the Starkscan webpage relative to the last performed transaction. In this way, the user can make sure he got awarded the prize the smart contract actually randomly extracted by looking at the events details.

View contract This button opens the Starkscan webpage of the smart contract. Here it is possible to take a look at the code[15] of the smart contract and all its past invocations.

It is possible to look at the complete code on Github at: https://github.com/Lapo9/Layer-2-Blockchains-for-Videogames/blob/master/LootBoxSmartContract/LootBoxClient.py

[15] Actually it is possible to take a look only at the bytecode. Starkscan (and other Starknet explorers) have a functionality to verify and upload the code of a smart contract, but, as of July 2023, it only works with smart contracts written in Cairo 0.x

Part V

Appendix

15 Merkle trees

A Merkle tree is a binary tree data structure where [Buterin, 2014]:

Leaves A leaf is labeled with the hash of a block of data.

Internal nodes An internal node is labeled with the hash of the labels of its 2 children.

This kind of structure is very useful in distributed systems, because it allows to verify data integrity very efficiently.

Indeed, to check data integrity, it is sufficient to compare the hash contained in the roots of the Merkle trees, since it is dependant on all of its children (direct and indirect). Moreover, in case the roots don't match, it is possible to rapidly identify the corrupted data, by descending in the tree and following the paths where the hashes do not correspond.

In the blockchain context Merkle trees are extensively employed, for example:

Block headers Block headers contain the root of the Merkle tree built with all the transactions inside the block.

Simple payment verification The presence of the Merkle tree root in the header of blocks, allows for a lightweight algorithm to verify the existence of a transaction `T`:

1. The lightweight node `L` fetches only the headers of all the blocks of the blockchain from full nodes (headers are orders of magnitude smaller than full blocks).
2. `L` asks the Merkle proof (see section Merkle proof) for the transaction `T` to full nodes.
3. Now `L` can easily verify whether the Merkle proof is valid, by simply calculating the Merkle root and searching for it in one of the headers he previously fetched.

15.1 Merkle proof

A Merkle proof is a cryptographically sound demonstration that a certain item belongs to a particular Merkle tree whose root is known.

If a verifier `V` wants proof from `W` that item `C` is part of the Merkle tree `M` whose root is known, it can ask `W` to send, for each level of `M`, the *missing children*.

Let's take the example of Figure 5:

1. `W` sent to `V`: $< D, H_{A-B}, H_{E-H} >$.

2. Now V has all the information compute $H_{C-D} = hash(C \,.\, D)$.
3. Now V has all the information compute $H_{A-D} = hash(H_{A-B} \,.\, H_{C-D})$.
4. Finally V has all the information compute the Merkle root $H_{A-H} = hash(H_{A-D}.H_{E-H})$.
5. At this point V can verify if the known Merkle root matches with H_{A-H}.

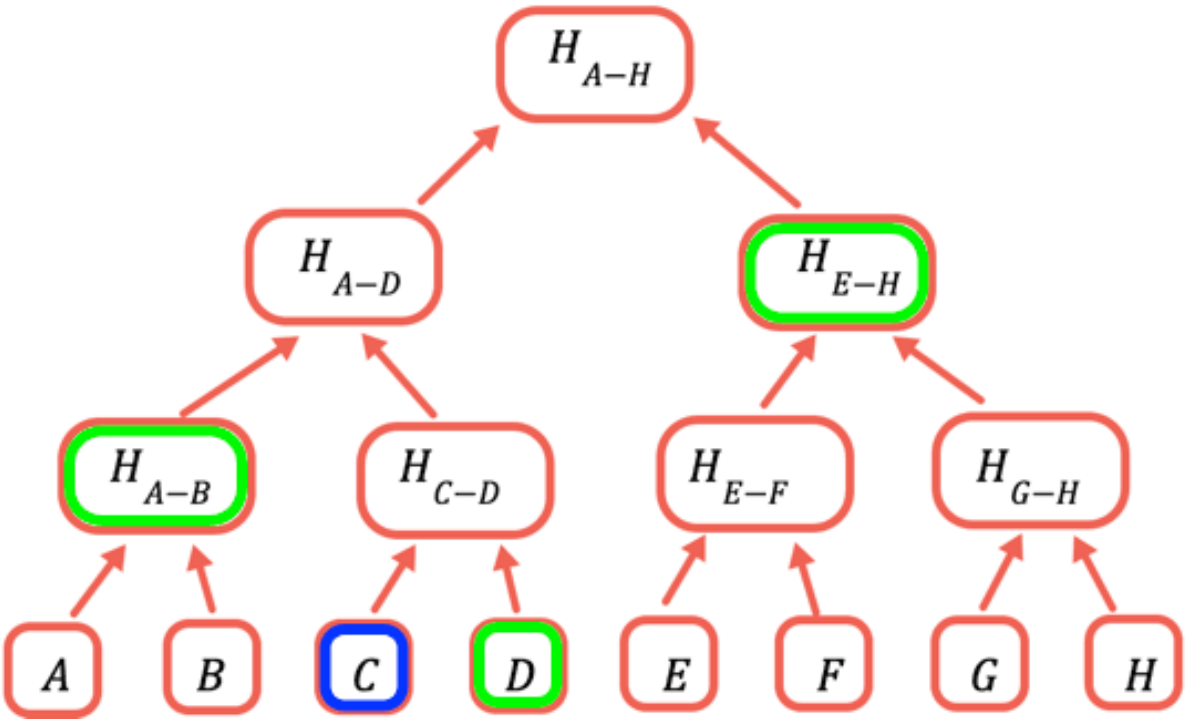

Figure 5: An example of a Merkle proof from [Pomerantz, 2021].

This works because W cannot produce fake *missing children* in order to obtain a desired root, since the hash function is non-invertible.

The usefulness of Merkle proofs lies in the fact that it is sufficient to exchange an amount of information that is logarithmic compared to the number of items in the set.

In the previous example, only 3 items were exchanged: $< D, H_{A-B}, H_{E-H} >$. If data wasn't organized in a Merkle tree, all the items $< A, B, D, E, F, G, H >$ should have been exchanged to compute the final hash.

References

[alemanaa, 2022] alemanaa (2022). Bridges. https://ethereum.org/en/developers/docs/bridges/.

[Atzei and Bartoletti, 2013] Atzei, N. and Bartoletti, M. (2013). A formal model of bitcoin transactions. *Univeristà degli Studi di Trento/Cagliari*.

[Berentsen et al., 2022] Berentsen, A., Lenzi, J., and Nyffenegger, R. (2022). A walk-through of a simple zk-stark proof. *SSRN*.

[Buterin, 2014] Buterin, V. (2014). Ethereum: A next-generation smart contract and decentralized application platform. *Ethereum website*, pages 9–10.

[Buterin, 2019] Buterin, V. (2019). Understanding plonk. https://vitalik.ca/general/2019/09/22/plonk.html.

[CairoCommunity, 2023] CairoCommunity (2023). Cairo documentation. https://book-cairo-lang.org.

[Carvalho, 2021] Carvalho, A. (2021). Bringing transparency and trustworthiness to loot boxes with blockchain and smart contracts. *Miami University*.

[corwintines, 2023] corwintines (2023). Scaling. https://ethereum.org/en/developers/docs/scaling/.

[d1onys1us, 2023] d1onys1us (2023). Zero-knowledge rollups. https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[Easley et al., 2018] Easley, D., O'Hara, M., and Basu, S. (2018). From mining to markets: The evolution of bitcoin transaction fees. *Cornell University*.

[EthereumCommunity, 2022] EthereumCommunity (2022). Ethereum layer 2. https://ethereum.org/en/layer-2/.

[EthereumCommunity, 2023] EthereumCommunity (2023). Optimistic rollups. https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/.

[Falcone, 2023] Falcone, L. (2023). Loot boxes are gambling. https://docs.google.com/presentation/d/11YnzHm33CbXt3qrp9v2pxWxf2oTpD9RE/edit?usp=sharing&ouid=116576255526134259427&rtpof=true&sd=true.

[Gilbert and Lynch, 2012] Gilbert, S. and Lynch, N. A. (2012). Perspective on the cap theorem. *Massachussets Institute of Technology*.

[Marchesi and Bruschi, 2022] Marchesi, A. A. and Bruschi, F. (2022). *Rewarding gaming achievements in a decentralized way*. PhD thesis, Politecnico di Milano.

[minimalsm, 2023] minimalsm (2023). State channels. https://ethereum.org/en/developers/docs/scaling/state-channels/#:~:text=State%20channels%20allow%

20participants%20to,open%20and%20close%20the%20channel.

[Musharraf, 2021] Musharraf, M. (2021). What is the blockchain trilemma. https://www.ledger.com/academy/what-is-the-blockchain-trilemma.

[Nakamoto, 2009] Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. *bitcoin.org*.

[Pomerantz, 2021] Pomerantz, O. (2021). Merkle proofs for offline data integrity. https://ethereum.org/en/developers/tutorials/merkle-proofs-for-offline-data-integrity/.

[Skinner, 1950] Skinner, B. F. (1950). Intermittent reinforcement. *Amer. Psychol.*

[StarknetCommunity, 2022] StarknetCommunity (2022). Starknet's structure: Sequencers, provers, and nodes. https://book.starknet.io/chapter_4/topology.html.

[StarknetCommunity, 2023] StarknetCommunity (2023). Account abstraction. https://book.starknet.io/chapter_5/index.html.

[Zohar, 2015] Zohar, A. (2015). Bitcoin: Under the hood. *acm*, page 9.