Optimistic and Validity Rollups: Analysis and Comparison between Optimism and StarkNet

Luca Donno[1]

[1]University of Bologna, Via Zamboni 33, Bologna, 40126, Italy

Abstract

The paper addresses the problem of scalability in decentralized blockchains by analyzing the trade-off between transaction throughput and hardware requirements to run a node. Rollups, i.e. technologies for on-chain verification of blocks executed off-chain, are presented in the form of fault or validity proofs. We compare Optimistic Rollups and Validity Rollups with respect to withdrawal time, transaction costs, optimization techniques, and compatibility with the Ethereum ecosystem. Our analysis reveals that Optimism Bedrock currently has a gas compression rate of approximately 20:1, while StarkNet achieves a storage write cost compression rate of around 24:1. We also discuss techniques to further optimize these rates, such as the use of cache contracts and Bloom filters. Ultimately, our conclusions highlight the trade-offs between complexity and agility in the choice between Optimistic and Validity Rollups.

Keywords

Blockchain, Scalability, Rollup

1. Introduction

Blockchain technology has gained significant attention due to its potential to revolutionize various industries. However, scalability remains a major challenge, as most blockchains face a trade-off between scalability, decentralization, and security, commonly referred to as the Scalability Trilemma [1, 2]. To increase the throughput of a blockchain, a trivial solution is to increase its block size. In the context of Ethereum, this means increasing the maximum amount of gas a block can hold. As each full node must validate every transaction of every block, as the throughput increases, the hardware requirements also increase, leading to a greater centralization of the network. Some blockchains, such as Bitcoin and Ethereum, optimize their design to maximize their architectural decentralization, while others, such as the Binance Smart Chain and Solana, are designed to be as fast and cheap as possible. Decentralized networks artificially limit the throughput of the blockchain to lower the hardware requirements to participate in the network.

Over the years, attempts have been made to find a solution to the Trilemma, such as state channels [3] and Plasma [4, 5]. These solutions have the characteristic of moving some activity off-chain, linking on-chain activity to off-chain activity using smart contracts, and verifying

DLT 2023: 5th Distributed Ledger Technology Workshop, May 25-26, 2023, Bologna, Italy

luca.donno@studio.unibo.it (L. Donno)
https://lucadonnoh.github.io/ (L. Donno)
0000-0001-9221-3529 (L. Donno)

on-chain what is happening off-chain. However, both Plasma and state channels are limited in their support of general smart contracts.

Rollups are blockchains (called Layer 2 or L2) that publish their blocks on another blockchain (Layer 1 or L1) and therefore inherit its consensus, data availability and security properties. They, unlike other solutions, support arbitrary computation. Rollups have three main components:

- **Sequencers**: nodes that receive Rollup transactions from users and combine them into a block that is sent to Layer 1. The block consists of at least the state root (e.g. a Merkle root) and the data needed to reconstruct and validate the state. The Layer 1 defines the canonical blockchain of the L2 by establishing the ordering of the published data.
- **Rollup full nodes**: nodes that obtain, process and validate Rollup blocks from Layer 1 by verifying that the root is correct. If a block contains invalid transactions it is then discarded, which prevents Sequencers from creating valid blocks that include invalid transactions.
- **Rollup light nodes**: nodes that obtain Rollup blocks from Layer 1 but do not compute the new state themselves. They verify that the new state root is valid using techniques such as fault or validity proofs.

Rollups achieve scalability by decreasing the amortized cost of transactions as the number of users increases. This is because the cost of ensuring blockchain validity grows sub-linearly with respect to the cost of verifying transactions individually. Rollups differ according to the mechanism by which they ensure the validity of transaction execution at light nodes: in Optimistic Rollups it is ensured by an economic model and by fault proofs, while in Validity Rollups it is cryptographically ensured using validity proofs.

Light nodes can be implemented as smart contracts on Layer 1. They accept the root of the new state and verify validity or fault proofs: these Rollup are therefore called Smart Contract Rollups. If light nodes are independent, they are called Sovereign Rollups [6]. The advantage of using a Smart Contract Rollup is to be able to build a trust-minimized bridge between the two blockchains: since the validity of the L2 state is proven to L1, a system of transactions from L2 to L1 can be implemented, allowing withdrawals. The disadvantage is that the cost of the transactions depends on the cost of verifying the state on L1: if the base layer is saturated by other activities, the cost of transactions on the Rollup also increases.

The data and consensus layers are the ones that determine the security of the system as they define the ordering of transactions, prevent attacks and make data available to prove state validity.

Paper contribution In this paper, we study Optimistic and Validity Rollups, two innovative solutions to the Scalability Trilemma, with a focus on notable implementations, such as Optimism Bedrock and StarkNet. Our contributions include a comprehensive comparison of these solutions, an analysis of withdrawal times, and a discussion of a possible attack on Optimism Bedrock. Additionally, we calculate their gas compression ratios, provide application-specific optimizations, and present the advantages and disadvantages of moving away from the Ethereum Virtual Machine (EVM).

Paper structure The paper is organized as follows. In section 2 Optimistic Rollups are introduced by analyzing Optimism Bedrock. In section 3 Validity Rollups are introduced by analyzing StarkNet. In section 4 we compare the two solutions. Finally, in section 5 we draw some conclusions.

2. Optimistic Rollups

The idea of accepting *optimistically* the output of blocks without verifying their execution is already present in the Bitcoin whitepaper [7], discussing light nodes. These nodes only follow the header chain by verifying the consensus rule, making them vulnerable to accept blocks containing invalid transactions in the event of a 51% attack. Nakamoto proposes to solve this problem by using an “alert" system to warn light nodes that a block contains invalid transactions. This mechanism is first implemented by Al-Bassam, Sonnino and Buterin [8] in which a fault proof system based on error correction codes [9] is used. In order to enable the creation of fault proofs, it is necessary that the data from all blocks, including invalid blocks, is available to the network: this is the Data Availability Problem, which is solved using a probabilistic data sampling mechanism. The first Optimistic Rollup design was presented by John Adler and Mikerah Quintyne-Collins in 2019 [10], in which blocks are published on another blockchain that defines their consensus on ordering.

2.1. Optimism Bedrock

Bedrock [11] is the latest version of Optimism, a Smart Contract Rollup. The previous version, the Optimistic Virtual Machine (OVM) required an ad hoc compiler to compile Solidity into its own bytecode: in contrast, Bedrock is fully equivalent to the EVM in that the execution engine follows the Ethereum Yellow Paper specification [12].

2.1.1. Deposits

Users can deposit transactions through a contract on Ethereum, the Optimism Portal, by calling the `depositTransaction` function. When a transaction is executed, a `TransactionDeposited` event is emitted, which each node in the Rollup listens for to process deposits. A deposited transaction is a L2 transaction that is derived from L1. If the caller of the function is a contract, the address is transformed by adding a constant value to it: this prevents attacks in which a contract on L1 has the same address as a contract on L2 but a different code. The inclusion on L2 of a deposited transaction is ensured by specification within a *sequencing window*.

Deposited transactions are a new EIP-2718 compatible transaction type [13] with prefix `0x7E`, where the rlp-encoded fields are:

- `bytes32 sourceHash`: hash that uniquely identifies the source of the transaction.
- `address from`: the address of the sender.
- `address to`: the receiver address, or the zero address if the deposited transaction is a contract creation.

- `uint256 mint`: the value to be created on L2.
- `uint256 value`: the value to be sent to the recipient.
- `bytes data`: the input data.
- `bytes gasLimit`: the gas limit of the transaction.

The `sourceHash` is computed as the keccak256 hash of the L1 block hash and the L1 log index, uniquely identifying an event in a block.

Since deposited transactions are initiated on L1 but executed on L2, the system needs a mechanism to pay on L1 for the gas spent on L2. One solution is to send ETH through the Portal, but this implies that every caller (even indirect callers) must be marked as `payable`, and this is not possible for many existing projects. The alternative is to burn the corresponding gas on L1.

The gas g allocated to deposited transaction is called *guaranteed gas*. The L2 gas price on L1 is not automatically synchronized but is estimated using a mechanism similar to EIP-1559 [14]. The maximum amount of gas guaranteed per Ethereum block is 8 million, with a target of 2 million. The quantity c of ETH required to pay for gas on L2 is $c = gb_{\text{L2}}$ where b_{L2} is the basefee on L2. The contract on L1 burns an amount of gas equal to c/b_{L2}. The gas spent to call `depositTransaction` is reimbursed on L2: if this amount is greater than the guaranteed gas, no gas is burned.

The first transaction of a rollup block is a *L1 attributes deposited transaction*, used to register on a L2 predeploy the attributes of Ethereum blocks. The attributes that the predeploy gives access to are the block number, the timestamp, the basefee, the block hash and the sequence number, which is the block number of L2 relative to the associated L1 block (also called *epoch*); this number is reset when a new epoch starts.

2.1.2. Sequencing

The Rollup nodes derive the Optimism chain entirely from Ethereum. This chain is extended each time new transactions are published on L1, and its blocks are reorganized each time Ethereum blocks are reorganized. The Rollup blockchain is divided into epochs. For each n block number of Ethereum, there is a corresponding n epoch. Each epoch contains at least one block, and each block in an epoch contains a L1 attributes deposited transaction. The first block in an epoch contains all transactions deposited through the Portal. Layer 2 blocks may also contained *sequenced transactions*, i.e. transactions sent directly to the Sequencer.

The Sequencer accepts transactions from users and builds blocks. For each block, it constructs a batch to be published on Ethereum. Several batches can be published in a compressed manner, taking the name *channel*. A channel can be divided into several *frames*, in case it is too large for a single transaction. A channel is defined as the compression with ZLIB [15] of rlp-encoded batches. The fields of a batch are the epoch number, the epoch hash, the parent hash, the timestamp and the transaction list.

A sequencing window, identified by an epoch, contains a fixed number w of consecutive L1 blocks that a derivation step takes as input to construct a variable number of L2 blocks. For epoch n, the sequencing window n includes the blocks $[n, n+w)$. This implies that the ordering of L2 transactions and blocks within a sequencing window is not fixed until the window ends. A rollup transaction is called *safe* if the batch containing it has been confirmed on L1. Frames

are read from L1 blocks to reconstruct batches. The current implementation does not allow the decompression of a channel to begin until all corresponding frames have been received. Invalid batches are ignored. Individual block transactions are obtained from the batches, which are used by the execution engine to apply state transitions and obtain the Rollup state.

2.1.3. Withdrawals

In order to process withdrawals, an L2-to-L1 messaging system is implemented. Ethereum needs to know the state of L2 in order to accept withdrawals, and this is done by publishing on the L2 Output Oracle smart contract on L1 the state root of each L2 block. These roots are optimistically accepted as valid (or finalized) if no fault proof is performed during the *dispute period*. Only addresses designated as *Proposers* can publish output roots. The validity of output roots is incentivized by having Proposers deposit a stake that is slashed if they are shown to have proposed an invalid root. Transactions are initiated by calling the function `initiateWithdrawal` on a predeploy on L2 and then finalized on L1 by calling the function `finalizeWithdrawalTransaction` on the previously mentioned Optimism Portal.

The output root corresponding to the L2 block is obtained from the L2 Output Oracle; it is verified that it is finalized, i.e. that the dispute period has passed; it is verified that the Output Root Proof matches the Oracle Proof; it is verified that the hash of the withdrawal is included in it using a Withdrawal Proof; that the withdrawal has not already been finalized; and then the call to the target address is executed, with the specified gas limit, amount of Ether and data.

2.1.4. Cannon: the fault proof system

If a Rollup Full Node, by locally executing batches and deposited transactions, discovers that the Layer 2 state does not match the state root published on-chain by a Proposer, it can execute a fault proof on L1 to prove that the result of the block transition is incorrect. Because of the overhead, processing an entire Rollup block on L1 is too expensive. The solution implemented by Bedrock is to execute on-chain only the first instruction of disagreement of `minigeth`, compiling it into a MIPS architecture that is executed on an on-chain interpreter and published on L1. `minigeth` is a simplified version of geth [1] in which the consensus, RPC and database have been removed.

To find the first instruction of disagreement, an interactive binary search is conducted between the one who initiated the fault proof and the one who published the output root. When the proof starts, both parties publish the root of the MIPS memory state halfway through the execution of the block on the Challenge contract: if the hash matches it means that both parties agree on the first half of the execution thus publishing the root of half of the second half, otherwise the half of the first half is published and so on. Doing so achieves the first instruction of disagreement in a logarithmic number of steps compared to the original execution. If one of the two stops interacting, at the end of the dispute period the other participant automatically wins.

To process the instruction, the MIPS interpreter needs access to its memory: since the root is available, the necessary memory cells can be published by proving their inclusion. To access the state of the EVM, use is made of the Preimage Oracle: given the hash of a block it returns

[1] https://geth.ethereum.org/docs

the block header, from which one can get the hash of the previous block and go back in the chain, or get the hash of the state and logs from which one can get the preimage. The oracle is implemented by `minigeth` and replaces the database. Queries are made to other nodes to obtain the preimages.

3. Validity Rollups

The goal of a Validity Rollup is to cryptographically prove the validity of the state transition given the sequence of transactions with a short proof that can be verified sub-linearly compared to the time of the original computations.

These kind of certificates are called *computational integrity proofs* and are practically implemented with SNARKs (Succint Non-interactive ARgument of Knowledge), which use arithmetic circuits as their computational model. Different SNARK implementations differ in proving time, verification time, the need of a trusted setup and quantum resistance [16, 17]. STARKs (Scalable Transparent ARgument of Knowledge) [18] are a type of SNARKs that does not require a trusted setup and are quantum resistant, while giving up some efficiency on proving and verification compared to other solutions.

3.1. StarkNet

StarkNet is a Smart Contract Validity Rollup developed by StarkWare that uses the STARK proof system to validate its state to Ethereum. To facilitate the construction of validity proofs, a virtual machine different than the EVM is used, whose high-level language is Cairo.

3.1.1. Deposits

Users can deposit transactions via a contract on Ethereum by calling the `sendMessageToL2` function. The message is recorded by computing its hash and increasing a counter. Sequencers listen for the `LogMessageToL2` event and encode the information in a StarkNet transaction that calls a function of a contract that has the `l1_handler` decorator. At the end of execution, when the proof of state transition is produced, the consumption of the message is attached to it and it is deleted by decreasing its counter.

The inclusion of deposited transactions is not required by the StarkNet specification, so a gas market is needed to incentivize Sequencers to publish them on L2. In the current version, because the Sequencer is centralized and managed by StarkWare, the cost of deposited transactions is only determined by the cost of executing the deposit. This cost is paid by sending ETH to `sendMessageToL2`. These Ethers remain locked on L1 and are transferred to the Sequencer on L1, when the deposited transaction is included in a state transition. The amount of ETH sent, if the deposited transaction is included, is fully spent, regardless of the amount of gas consumed on L2.

StarkNet does not have a system that makes L1 block attributes available automatically. Alternatively, Fossil is a protocol developed by Oiler Network [2] that allows, given a hash of a block, any information to be obtained from Ethereum by publishing preimages.

[2] https://www.oiler.network/

3.1.2. Sequencing

The current state of StarkNet can be derived entirely from Ethereum. Any state difference between transitions is published on L1 as calldata. Differences are published for each contract and are saved as `uint256[]` with the following encoding:

- Number of field concerning contract deployments.
- For each published contract:
  - The address of the published contract.
  - The hash of the published contract.
  - The number of arguments of the contract constructor.
  - The list of constructor arguments
- Number of contract whose storage has been modified.
- For each contract that has been modified:
  - The address of the modified contract.
  - The number of storage updates.
  - The key-value pairs of the storage addresses with the new values.

The state differences are published in order, so it is sufficient to read them sequentially to reconstruct the state.

3.1.3. Withdrawals

To send a message from L2 to L1, the syscall `send_message_to_L1` is used. The message is published to L1 by increasing its hash counter along with the proof and finalized by calling the function `consumeMessageFromL2` on the StarkGate smart contract on L1, which decrements the counter. Anyone can finalize any withdrawal.

3.1.4. Validity proofs

The Cairo Virtual Machine [19] is designed to facilitate the construction of STARK proofs. The Cairo language allows the computation to be described with a high-level programming language, and not directly as a circuit. This is accomplished by a system of polynomial equations [3] representing a single computation: the FDE cycle of a von Neumann architecture. The number of constraints is thus fixed and independent of the type of computation, allowing for only one Verifier program for every program whose computation needs to be proved.

StarkNet aggregates multiple transactions into a single STARK proof using a shared prover named SHARP. The proofs are sent to a smart contract on Ethereum, which verifies their validity and updates the Merkle root corresponding to the new state. The sub-linear cost of verifying a validity proof allows its cost to be amortized over multiple transactions.

[3] called Algebraic Intermediate Representation (AIR)

4. Comparison

4.1. Withdrawal time

The most important aspect that distinguishes Optimistic Rollups from Validity Rollups is the time that elapses between the initialization of a withdrawal and its finalization. In both cases, withdrawals are initialized on L2 and finalized on L1. On StarkNet, finalization is possible as soon as the validity proof of the new state root is accepted on Ethereum: theoretically, it is possible to withdraw funds in the first block of L1 following initialization. In practice, the frequency of sending validity proofs on Ethereum is a trade-off between the speed of block finalization and proof aggregation. Currently StarkNet provides validity proofs for verification every 10 hours [4], but it is intended to be decreased as transaction activity increases.

On Optimism Bedrock it is possible to finalize a withdrawal only at the end of the dispute period (currently 7 days), after which a root is automatically considered valid. The length of this period is mainly determined by the fact that fault proofs can be censored on Ethereum until its end. The success probability of this type of attack decreases exponentially as time increases:

$$\mathbb{E}[\text{subtracted value}] = Vp^n$$

where n is the number of blocks in an interval, V is the amount of funds that can be subtracted by publishing an invalid root, and p is the probability of successfully performing a censorship attack in a single block. Suppose that this probability is 99%, that the value locked in the Rollup is one million Ether, and that the blocks in an interval are 1800 (6 hours of blocks with a 12 seconds interval): the expected value is about 0.01391 Ether. The system is made secure by asking Proposers to stake a much larger amount of Ether than the expected value.

Winzer et al. showed how to carry out a censorship attack using a simple smart contract that ensures that certain areas of memory in the state do not change [20]. Modeling the attack as a Markov game, the paper shows that censoring is the dominant strategy for a rational block producer if they receive more compensation than including the transaction that changes the memory. The p value discussed above can be viewed as the percentage of rational block producers in the network, where "rational" does not take into account possibly penalizing externalities, such as less trust in the blockchain that decreases its cryptocurrency value.

The following code presents a smart contract that can be used to perform a censorship attack on Bedrock. The attack exploits the incentives of block producers by offering them a bribe to censor the transactions that would modify specific parts of the state. The contract's main function, `claimBribe`, allows block producers to claim the bribe if they successfully censor the targeted transaction by checking that the invalid output root is not touched.

```
function claimBribe(bytes memory storageProof) external {
    require(!claimed[block.number], "bribe already claimed");
    OutputProposal memory current = storageOracle.getStorage(L2_ORACLE, block.number, SLOT,
        storageProof);
    require(invalidOutputRoot == current.outputRoot, "attack failed");
    claimed[block.number] = true;
    (bool sent, ) = block.coinbase.call{value: bribeAmount}("");
```

[4] https://etherscan.io/address/0xc662c410c0ecf747543f5ba90660f6abebd9c8c4

```
    require(sent, "failed to send ether");
}
```

Listing 1: Example of a contract that incentivizes a censorship attack on Bedrock.

The length of the dispute period must also take into account the fact that the fault proof is an interactive proof and therefore enough time must be provided for participants to interact and that any interaction could be censored. If the last move occurs at a time very close to the end of the dispute period, the cost of censoring is significantly less. Although censoring is the dominant strategy, the likelihood of success is lower because censoring nodes are vulnerable to Denial of Service attacks: an attacker can generate very complex transactions that end with the publication of a fault proof at no cost, as no fees would be paid.

In extreme cases, a long dispute period allows coordination in the event of a successful censorship attack to organize a fork and exclude the attacking block producers. Another possible attack consists in publishing more state root proposals than disputants can verify, which can be avoided using a frequency limit.

4.1.1. Fast optimistic withdrawals

Since the validity of an Optimistic Rollup can be verified at any time by any Full Node, a trusted oracle can be used to know on L1 whether the withdrawal can be finalized safely. This mechanism was first proposed by Maker [21]: an oracle verifies the withdrawal, publishes the result on L1 on which an interest-bearing loan is assigned to the user, which is automatically closed at the end of 7 days, i.e. when the withdrawal can actually be finalized. This solution introduces a trust assumption, but in the case of Maker it is minimized since the oracle operator is managed by the same organization that assumes the risk by providing the loan.

4.2. Transaction costs

The cost of L2 transactions is mostly determined by the interaction with the L1. In both solutions the computational cost of transactions is very cheap as it is executed entirely off-chain.

Optimism publishes L2 transactions calldata as calldata and rarely (or never) executes fault proofs, therefore calldata is the most expensive resource. On January 12, 2022 a Bedrock network has been launched on the Ethereum's Goerli testnet. A gas compression rate can be calculated by tracking the amount of gas used on Bedrock in a certain period and by comparing it to the amount of gas spent on L1 for the corresponding blocks. Using this method a gas compression rate of $\sim 20:1$ is found, but this figure may differ with real activity on mainnet.

StarkNet publishes on Ethereum every change in L2 state as calldata, therefore storage is the most expensive resource. Since the network does not use the EVM, the transaction cost compression cannot be trivially estimated. By assuming the cost of execution and calldata to be negligible, it is possible to calculate the compression ratio of storage writes compared to L1. Assuming no contract is deployed and 10 cells not previously accessed on StarkNet are modified, a storage write cost compression rate of $\sim 24:1$ is found. If a cell is overwritten n times between data publications, the cost of each write will be $1/n$ compared to the cost of a single write, since only the last one is published. The cost can be further minimized by

compressing frequently used values. The cost of validity proof verification is divided among the transactions it refers to: for example, StarkNet block 4779 contains 200 transactions and its validity proof consumes 267830 units of gas, or 1339.15 gas for each transaction.

4.2.1. Optimizing calldata: cache contract

Presented below is a smart contract that implements an address cache for frequently used addresses by taking advantage of the fact that storage and execution are much less expensive resources, along with a `Friends` contract that demonstrates its use. The latter keeps track of the "friends" of an address that can be registered by calling the `addFriend` function. If an address has already been used at least once, it can be added by calling the `addFriendWithCache` function: the cache indices are 4-byte integers while the addresses are represented by 20 bytes, so there is a 5:1 saving on the function argument. The same logic can be used for other data types such as integers or more generally bytes.

```
contract AddressCache {
    mapping(address => uint32) public address2key;
    address[] public key2address;

    function cacheWrite(address _address) internal returns (uint32) {
        require(key2address.length < type(uint32).max, "AddressCache: cache is full");
        require(address2key[_address] == 0, "AddressCache: address already cached");
        // keys must start from 1 because 0 means "not found"
        uint32 key = uint32(key2address.length + 1);
        address2key[_address] = key;
        key2address.push(_address);
        return key;
    }

    function cacheRead(uint32 _key) public view returns (address) {
        require(_key <= key2address.length && _key > 0, "AddressCache: key not found");
        return key2address[_key - 1];
    }
}
```

Listing 2: Address cache contract.

```
contract Friends is AddressCache {
    mapping(address => address[]) public friends;

    function addFriend(address _friend) public {
        friends[msg.sender].push(_friend);
        cacheWrite(_friend);
    }

    function addFriendWithCache(uint32 _friendKey) public {
        friends[msg.sender].push(cacheRead(_friendKey));
    }

    function getFriends() public view returns (address[] memory) {
        return friends[msg.sender];
```

```
    }
}
```

Listing 3: Example of a contract that inherits the address cache.

The contract supports in cache about 4 billion (2^{32}) addresses, and adding one byte gives about 1 trillion (2^{40}).

4.2.2. Optimizing storage: Bloom's filters

On StarkNet there are several techniques for minimizing storage usage. If it is not necessary to guarantee the availability of the original data then it is sufficient to save on-chain its hash: this is the mechanism used to save data for an ERC-721 (NFT) [22], i.e., an IPFS link that resolves the hash of the data if available. For data that is stored multiple times, it is possible to use a look-up table similar to the caching system introduced for Optimism, requiring all values to be saved at least once. For some applications, saving all the values can be avoided by using a Bloom filter [23, 24, 25], i.e., a probabilistic data structure that allows one to know with certainty whether an element does not belong to a set but admits a small but non-negligible probability of false positives.

A Bloom filter is initialized as an array of m bits at zero. To add an element, k hash functions with a uniform random distribution are used, each one mapping to a bit of the array that is set to 1. To check whether an element belongs to the set we run the k hash functions and verify that the k bits are set to 1. In a simple Bloom's filter there is no way to distinguish whether an element actually belongs to the set or is a false positive, a probability that grows as the number of entries increases. After inserting n elements:

$$\mathbb{P}[\text{false positive}] = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

assuming independence of the probability of each bit set. If n elements (of arbitrary size!) are expected to be included and the probability of a false positive tolerated is p, the size of the array can be calculated as:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

While the optimal number of hash functions is:

$$k = \frac{m}{n} \ln 2$$

If we assume to insert 1000 elements with a tolerance of 1%, the size of the array is 9585 bits with $k = 6$, while for a tolerance of 0.1% it becomes 14377 bits with $k = 9$. If a million elements are expected to be inserted, the size of the array becomes about 1170 kB for 1% and 1775 kB for 0.1%, with the same values of k, since it depends only on p [26].

In a game where players must not be assigned to an opponent they have already challenged, instead of saving in storage for each player the list of past opponents one can use a Bloom filter. The risk of not challenging some players is often acceptable, and the filter can be reset periodically.

4.3. Ethereum compatibility

The main advantage of being compatible with EVM and Ethereum is the reuse of all the available tools. Ethereum smart contracts can be published on Optimism without any modification nor new audits. Wallets remain compatible, development and static analysis tools, general analysis tools, indexing tools and oracles. Ethereum and Solidity have a long history of well-studied vulnerabilities, such as reentrancy attacks, overflows and underflows, flash loans, and oracle manipulations. Because of this, Optimism was able to capture a large amount of value in a short time.

Choosing to adopt a different virtual machine implies having to rebuild an entire ecosystem, with the advantage of a greater implementation freedom. StarkNet natively implements *account abstraction*, which is a mechanism whereby each account is a smart contract that can implement arbitrary logic as long as it complies with an interface (hence the term *abstraction*): this allows the use of different digital signature schemes, the ability to change the private key using the same address, or use a multisig. The Ethereum community proposed the introduction of this mechanism with EIP-2938 in 2020, but the proposal has remained stale for more than a year as other updates have been given more priority [27].

Another important benefit gained from compatibility is the reuse of existing clients: Optimism uses a version of geth for its own node with only ~ 800 lines of difference, which has been developed, tested, and maintained since 2014. Having a robust client is crucial as it defines what is accepted as valid or not in the network. A bug in the implementation of the fault proof system could cause an incorrect proof to be accepted as correct or a correct proof for an invalid block to be accepted as incorrect, compromising the system. The likelihood of this type of attack can be limited with a wider client diversity: Optimism can reuse in addition to geth the other Ethereum clients already maintained, and development of another Erigon-based client is already underway. In 2016 a problem in the memory management of geth was exploited for a DoS attack and the first line of defense was to recommend the use of Parity, the second most used client at the time [5]. StarkNet faces the same problem with validity proofs, but the clients have to be written from scratch and the proof system is much more complex, and consequently it is also much more complex to ensure correctness.

5. Conclusion

Rollups are the most promising solution available today to solve the scalability problem in decentralized blockchains, paving the way for the era of modular blockchains as opposed to monolithic blockchains.

The choice of developing either an Optimistic Rollup or a Validity Rollup is mainly shown as a trade-off between complexity and agility. StarkNet has numerous advantages such as fast withdrawals, structural inability to have invalid state transitions, lower transaction cost at the expense of a longer development period and incompatibility with EVM, while Optimism has leveraged the network economy to quickly gain a major share of the market.

Optimism Bedrock, however, possesses a modular design that allows it to become a Validity

[5] https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack

Rollup in the future: Cannon currently uses `minigeth` compiled to MIPS for its fault proof system, but the same architecture can be used to obtain a circuit and produce validity proofs. Compiling a complex machine such as the EVM for a microarchitecture results in a simpler circuit that does not need to be modified and re-verified in case of upgrades. RISC Zero is a verifiable microarchitecture with STARK proofs already in development based on RISC-V that can be used for this purpose as an alternative to MIPS [28].

One aspect that should not be underestimated is the complexity in understanding how the technology works. A strength of traditional blockchains is to be able to verify the state of the blockchain without trusting any third party entity. However, in the case of StarkNet, it is necessary to trust the implementation when it is not possible to verify the various components based on cryptography and advanced mathematics. This may initially create friction for the adoption of the technology, but as the tools and the usage of integrity proofs advance even outside the blockchain field this problem will be hopefully solved.

References

[1] A. Altarawneh, T. Herschberg, S. Medury, F. Kandah, A. Skjellum, Buterin's scalability trilemma viewed through a state-change-based classification for common consensus algorithms, in: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2020, pp. 0727–0736.

[2] A. Hafid, A. S. Hafid, M. Samih, Scaling blockchains: A comprehensive survey, IEEE Access 8 (2020) 125244–125262.

[3] L. D. Negka, G. P. Spathoulas, Blockchain state channels: A state of the art, IEEE Access 9 (2021) 160277–160298.

[4] J. Poon, V. Buterin, Plasma: Scalable autonomous smart contracts, White paper (2017) 1–47.

[5] G. Konstantopoulos, Plasma cash: towards more efficient plasma constructions, arXiv preprint arXiv:1911.12095 (2019).

[6] E. N. Tas, J. Adler, M. Al-Bassam, I. Khoffi, D. Tse, N. Vaziri, Accountable safety for rollups, arXiv preprint arXiv:2210.15017 (2022).

[7] S. Nakamoto, Bitcoin whitepaper, URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019) (2008).

[8] M. Al-Bassam, A. Sonnino, V. Buterin, Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities, arXiv preprint arXiv:1809.09044 160 (2018).

[9] P. Elias, Error-free coding (1954).

[10] J. Adler, M. Quintyne-Collins, Building scalable decentralized payment systems, arXiv preprint arXiv:1904.06441 (2019).

[11] The Optimism Collective, The Optimism Monorepo, ???? URL: https://github.com/ethereum-optimism/optimism.

[12] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) 1–32.

[13] M. Zoltu, Eip-2718: Typed transaction envelope, 2020.

[14] V. Buterin, E. Conner, R. Dudley, M. Slipper, I. Norden, A. Bakhta, Eip-1559: Fee market change for eth 1.0 chain, 2019.
[15] P. Deutsch, J.-L. Gailly, ZLIB Compressed Data Format Specification version 3.3, RFC 1950, RFC Editor, 1996. URL: https://www.rfc-editor.org/rfc/rfc1950.html.
[16] T. Chen, H. Lu, T. Kunpittaya, A. Luo, A review of zk-snarks, arXiv preprint arXiv:2202.06877 (2022).
[17] D. Čapko, S. Vukmirović, N. Nedić, State of the art of zero-knowledge proofs in blockchain, in: 2022 30th Telecommunications Forum (TELFOR), IEEE, 2022, pp. 1–4.
[18] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev, Scalable, transparent, and post-quantum secure computational integrity, Cryptology ePrint Archive (2018).
[19] L. Goldberg, S. Papini, M. Riabzev, Cairo–a turing-complete stark-friendly cpu architecture, Cryptology ePrint Archive (2021).
[20] F. Winzer, B. Herd, S. Faust, Temporary censorship attacks in the presence of rational miners, in: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), IEEE, 2019, pp. 357–366.
[21] S. MacPherson, Announcing the optimism dai bridge with fast withdrawals, 2021. URL: https://forum.makerdao.com/t/announcing-the-optimism-dai-bridge-with-fast-withdrawals/6938.
[22] W. Entriken, D. Shirley, J. Evans, N. Sachs, Eip-721: Non-fungible token standard, 2018. URL: https://eips.ethereum.org/EIPS/eip-721.
[23] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (1970) 422–426.
[24] K. Christensen, A. Roginsky, M. Jimeno, A new analysis of the false positive rate of a bloom filter, Information Processing Letters 110 (2010) 944–949.
[25] S. Agarwal, A. Trachtenberg, Approximating the number of differences between remote sets, in: 2006 IEEE Information Theory Workshop-ITW'06 Punta del Este, IEEE, 2006, pp. 217–221.
[26] D. Starobinski, A. Trachtenberg, S. Agarwal, Efficient pda synchronization, IEEE Transactions on Mobile Computing 2 (2003) 40–51.
[27] V. Buterin, A. Dietrichs, M. Garnett, W. Villanueva, S. Wilson, Eip-2938: Account abstraction, 2020. URL: https://eips.ethereum.org/EIPS/eip-2938.
[28] risc0, Risc zero, 2022. URL: https://github.com/risc0/risc0.