Tylix: The Next Generation of Gaming Payment System

Samet Yusuf İÇGEN

https://tylix.io/#

Disclaimer The information contained in this white paper does not constitute an offer to sell or a solicitation of an offer to buy any token. Tylix is publishing this whitepaper solely to solicit feedback and comments from the public. In the event that Tylix offers any tokens (or future contracts to be written) for sale, it will do so by means of a disclosure document and final offering documents, including risk factors. These final documents are expected to include an updated version of this white paper, which may differ materially from the current version. In the event that Tylix conducts an offering in the Republic of Türkiye, it is likely that such offering will be available exclusively to accredited investors. It is imperative to note that the information provided in this whitepaper should not be construed as a guarantee or promise of the performance of Tylix's business or its tokens, nor should it be regarded as a guarantee or promise of the utility or value of the tokens. This document details the current plans of Tylix, which are subject to change at the company's discretion, and the success of these plans will depend on many factors, including market-based elements beyond Tylix's control and factors in the data and cryptocurrency industries. All forward-looking statements are based solely on Tylix's analysis of the matters described in this whitepaper. The accuracy of this analysis is not guaranteed.

Summary

This paper provides a technical and theoretical discussion of the development of Tylix, a system that introduces a new perspective on digital assets and financial systems in gaming. The paper describes the development of a system using blockchain technology. The paper adopts a Blockchain architecture based on Proof of History (PoH), a method of verifying the order and passage of time between events. PoH utilizes a data structure integrated into a ledger to encode the passage of time, thereby ensuring its reliability and security.

1-Introduction

A key component of any successful game is the ability to attract and retain a strong player base. While game designers and developers prioritize adding "fun" to the game to attract more players, our role is to provide a set of well-designed tools to ensure that the player experience is optimal.

As the gaming industry evolves with technological innovations and increased digital interactions, traditional payment systems are no longer adequate due to slow transaction confirmation times, centralized architectures, and security vulnerabilities. In response, Tylix has integrated an advanced Proof-of-History (PoH) mechanism that cryptographically verifies the chronological order of transactions, providing a decentralized and highly reliable infrastructure that is validated in real-time and with high efficiency. The PoH mechanism ensures that every transaction is accurately and securely timestamped, enhancing system scalability, energy efficiency, and security standards. This integration represents a significant advancement in the realm of in-game payment processes.

2-Outline

The remainder of this paper will adhere to the outlined structure. The overall system design is described in Section 3. Section 4 provides a comprehensive overview of the PoH system. The tokens to be used in the games and general information are explained in Section 5.

3- General System Design

Tylix is an ERC-20 standard token that operates on the Ethereum mainnet. It will be utilized for in-game payments. The primary reasons for selecting the Ethereum network include its decentralization, security, and its role as part of a larger ecosystem. Initially, the token will be used for in-game payments, but it will transition to NFTs and decentralized governance in the future.

3.1 System Architecture and Layers

The Tylix is a token configured to run on the Ethereum mainnet. It consists of the following layers:

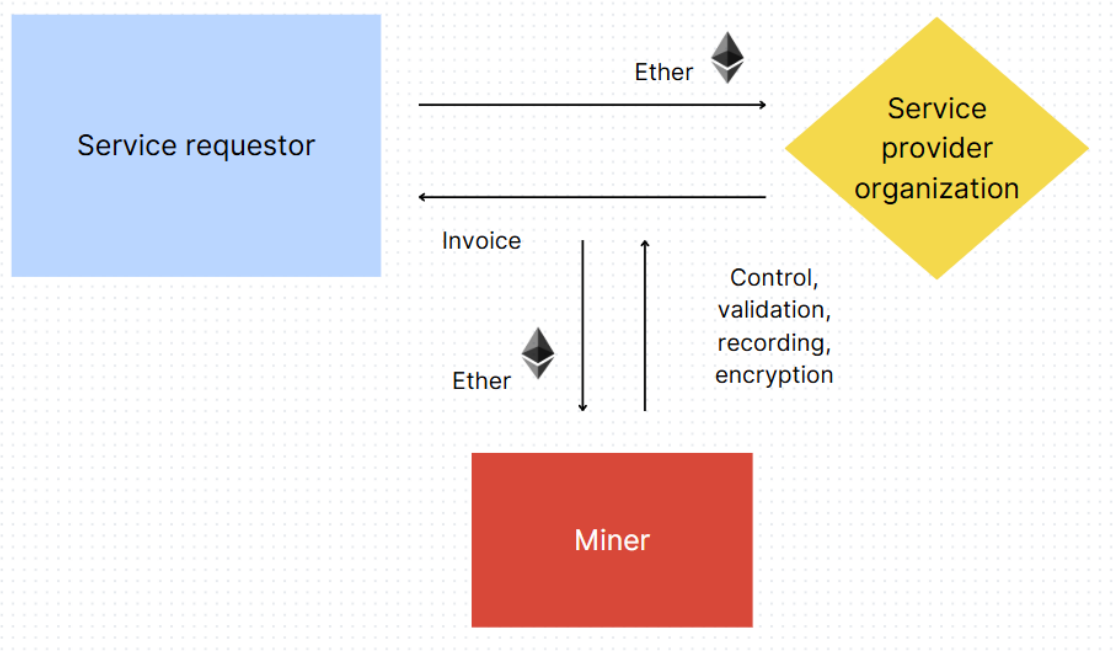


Figure 1: Ethereum Mainnet

3.1.1 Infrastructure Layer

The Ethereum mainnet (figure 1) creates and validates blocks using the PoS (Proof of Stake) mechanism. While this provides a decentralized and secure environment, the average block duration and dynamic gas fees directly affect the system performance. To address these challenges, Layer 2 solutions for high-frequency micropayments are under consideration.

Integration with Optimistic Rollups and zk-Rollups is planned to increase transaction throughput while reducing gas costs and alleviating on-chain congestion, and interoperability with a secure bridge mechanism for seamless data flow between the Ethereum mainnet and Layer 2.

3.1.2 Smart Contract Layer

Tylix provides essential functionalities such as aggregate supply, transfer transactions, and balance management in accordance with the ERC-20 standard, while ensuring the security of its code integrity using OpenZeppelin libraries and audited contract modules such as ERC20, Pausable, and Ownable for standard compliance.

A unique hash (generated with keccak 256) is recorded for each transaction. This is done with the “recordHistory()” function in the contract. An additional precaution is taken against replay attacks by ensuring time ordering with checks such as “require(block.timestamp > lastTimestamp)”. Considering the storage costs, mapping structure is preferred.

In addition, the “pausable” mechanism with emergency stop option is used against critical situations. With gas optimization and “state freeze” logic, all operations are stopped and intervention is possible in unusual situations in the system. ERC-20 transfer functions are protected against reentrancy attacks with inherent security measures (e.g. Solidity 0.8's auto overflow check).

In the “safeTransfer()” function, additional parameter checks (for example, that the address is non-zero) are applied to provide error handling.

ERC-721/ERC-1155 based NFT integration is planned in the future.

3.1.3 Application and API Layer

Integration with wallets such as Metamask, Trust Wallet, CoinMarketCap, CoinGecko will be listed and recognized on famous exchanges. For game developers, API’s that are compatible with libraries such as Web3.js and ethers.js will be offered.

3.2 Security and Performance Optimizations

The use of OpenZeppelin libraries, which are audited modules, helps to minimize known vulnerabilities. Formal verification processes can be applied for critical modules. Functions are safeguarded by state updates and the sequential order of external calls. Given that block timestamps are under miner control, additional logic checks have been implemented.

To ensure cost-effectiveness, we prioritize more efficient logging mechanisms based on mapping and events, as storing transaction history and other logs can lead to high gas fees.

3.2.1 Layer 2 Integration

Given the high transaction volume on the Ethereum main chain, scalability will be ensured with Optimistic Rollups or zk-Rollups. To address the transaction fee issues, game-specific sidechains will be utilized for special cases.

4-Proof of History

Proof of History, is a series of calculations that can be used to verify the time difference between two events using cryptography. It utilizes cryptographically secure functions that are designed so that the output cannot be deduced from the input. The function must be executed in its entirety to generate the output. The function is executed sequentially on a single core, taking its previous output as the current input. It periodically records the current output and the number of times it has been called. The output can then be recomputed and verified by external computers in parallel by checking each sequence segment on a separate core. The array can be timestamped by appending the data (or a hash of the data) to the state of the function. By recording the state, index, and data as they are added to the array, we can ensure that the data was created before the next hash is generated in the array. This design also supports horizontal scaling

because multiple generators can synchronize between each other by shuffling their states into each other's arrays.

4.1 Description

The system is processed using a cryptographically strong hash function (e.g. keccak256, sha256, etc.) that yields unpredictable output as follows.

A random initial value is determined. This value can be, for example, a viral news headline or a completely random string.

Our example initial value:

“Today the weather in Izmir is clear, the temperature is 23 degrees”

The selected initial value is given as input to the cryptographic hash function. The resulting hash value is again used as input to the same function for the next calculation. This process continues sequentially, with the output value of the hash function at each step being the input for the next step. At each hash calculation step, the number of function calls (index) and the resulting hash value are recorded. This serves as a timestamp documenting the correctness and sequentiality of the hash chain. Example spreadsheet:

Index	Transaction Description	Calculation step	Sample Output Hash
1	Assignation of the initial value to hash function	Keccak256(“Wather in Izmir is clear today, with a temperature of 23 degrees.”)	0xA1B9s08
2	Utilizing the previous hash output	Keccak256(0xA1B9s08)	0xE51903AA

3	Utilizing the hash output of the second step	Keccak256(0xE51903AA)	0x56T2M45O
100	Utilizing the hash obtained after 99 consecutive calculations	Keccak256(hash_99)	0xT2462Hs97
250	Utilizing the hash obtained after 249 consecutive calculations	Keccak256(hash_249)	0x7QER43

Please note that the hash values assigned to the table are for illustrative purposes only. In actual calculations, unique and unpredictable values are generated depending on the output of the function.

It is not necessary for all steps in the subsequent process to be published. By periodically publishing selected indices and corresponding hash outputs, the system can prove the sequentiality and continuity of operations. For instance, a concise representation such as the following table can be shared:

Index	**Transaction Description**	**Sample Output Hash**
1	Assignation of the initial value to hash function	0x25YSA34G52D
75	Hash that has been calculated after 74. step	0xYU98GID99
150	Hash that has been calculated after 149. step	0x3O4HJ34J3K

Given the use of a collision-resistant hash function, it is imperative to calculate the hash value iteratively to ensure the integrity of the result. In other words, it is not possible to estimate the hash value at index 150 without running it 150 times. Each computation depends on the previous output, making parallel or randomized computations infeasible. This aspect of the process is instrumental in determining the elapsed time between computations, a crucial metric in real-time applications

Published hash and index pairs can be verified by monitoring parties. The published values are the result of successive calculations from the initial value, proving that transactions between two published points occurred in real-time. This structure enables the number of transactions since a given starting point to serve as a measure of time, as the sequence of each calculation in the chain can be tracked without disrupting the order. For instance, the difference between index 1 and 250 represents the total time spent on calculations within this interval, offering a reliable means to objectively document the system's progress over time.

4.2 Timestamp for Events

This hash array can also be used to record that some pieces of data were created before a particular hash index was created. It uses a 'combine' function to combine the piece of data with the hash in the current index. The data may simply be a cryptographically unique hash of random event data. The combine function can be a simple data append or any collision resistant operation. The next generated hash represents the timestamp of the data, since it could only have been generated after a particular piece of data was added.

For example:

Index	Operation	Output Hash
1	Sha256(“any randomly initialized value”)	Hash1
100	Sha256(hash99)	Hash100
300	Sha256(hash299)	Hash300

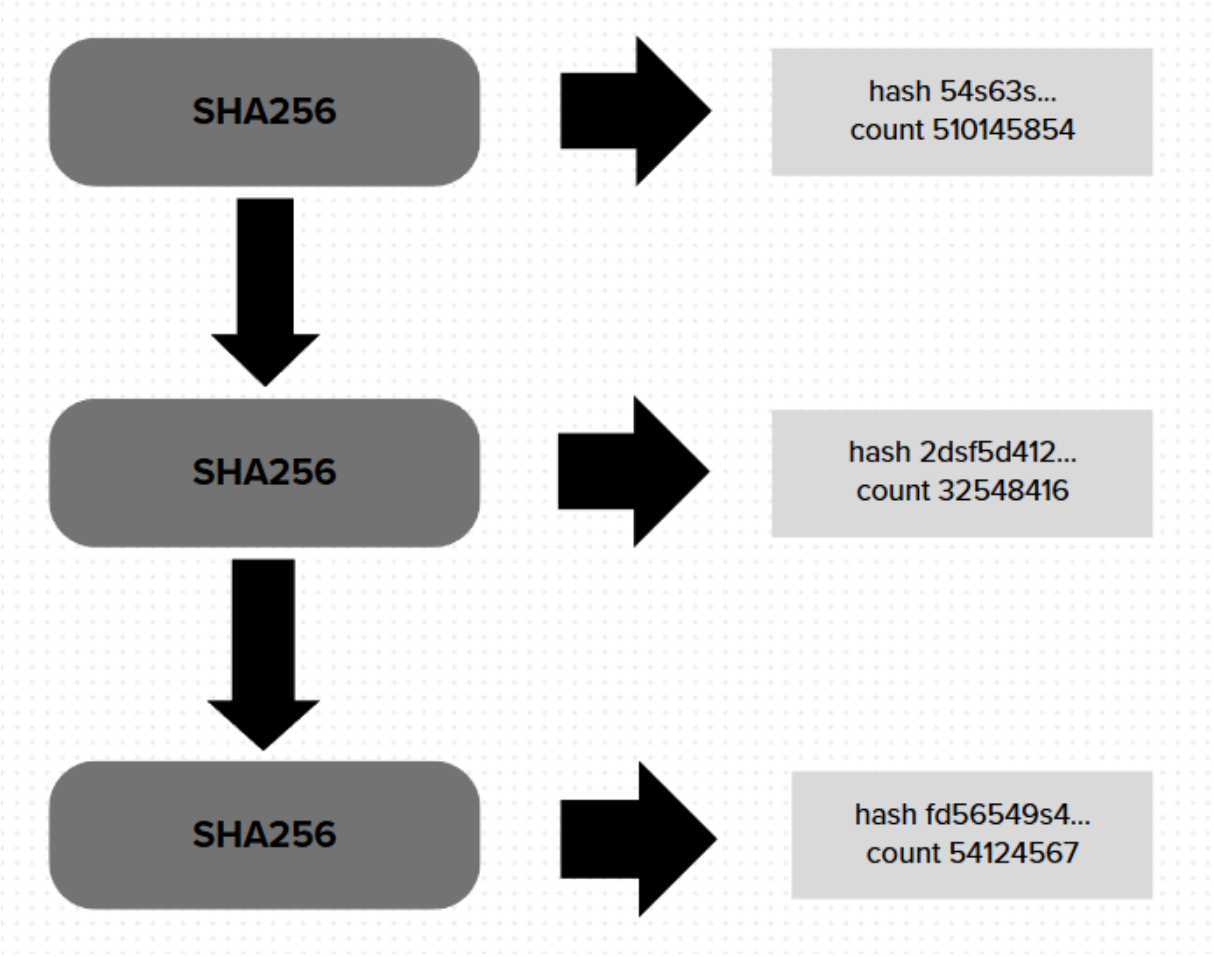


Figure 2: Proof of History sequence

The system is initialised using an example of a cryptographic hash function (e.g. blake2b) that cannot be guessed. A random initial value is set and the function continues by taking the output back to the input.

PoH sequence with data

Index	Operation	Output hash
1	Blake2b("random initial value")	Hash1
150	Blake2b(hash149)	Hash150
270	Blake2b(hash269)	Hash270

At this point an external event occurs. For example, a sensor records some instantaneous data and this data is added to the PoH chain.

2-event PoH sequence

Index	Operation	Output Hash
1	Blake2b(“random initial value”)	hash1
150	Blake2b(hash149)	Hash150
310	Blake2b(hash309)	Hash310
400	Blake2b(hash399)	Hash400
500	Blake2b(hash499)	Hash500
650	Blake2b(hash649)	Hash650
725	Blake2b(hash724)	Hash725

Sample Explanations:

310.hash => Sensor data added

500.hash => A file (PDF, JSON or any data) has been added with a SHA-256 hash value.

725.hash => A transaction record has been included in the chain.

This system allows any node to verify the sequence and confidently determine the time interval in which events occurred. Due to the collision resistant nature of the hash function, future hash values cannot be predicted.

The data scrambled into the array can be the raw data itself or a hash of the data with accompanying metadata. In the figure below, the entry rte8w54q... has been added to the Proof of History index. The number it was added to is 6548215454235 and the state it was added to is 65sdsa35sa694d. All future hashes generated are replaced by this change in the array, which is highlighted by the color change in the figure. Each node observing this sequence can determine the order in which all events were inserted and estimate the actual time between insertions.

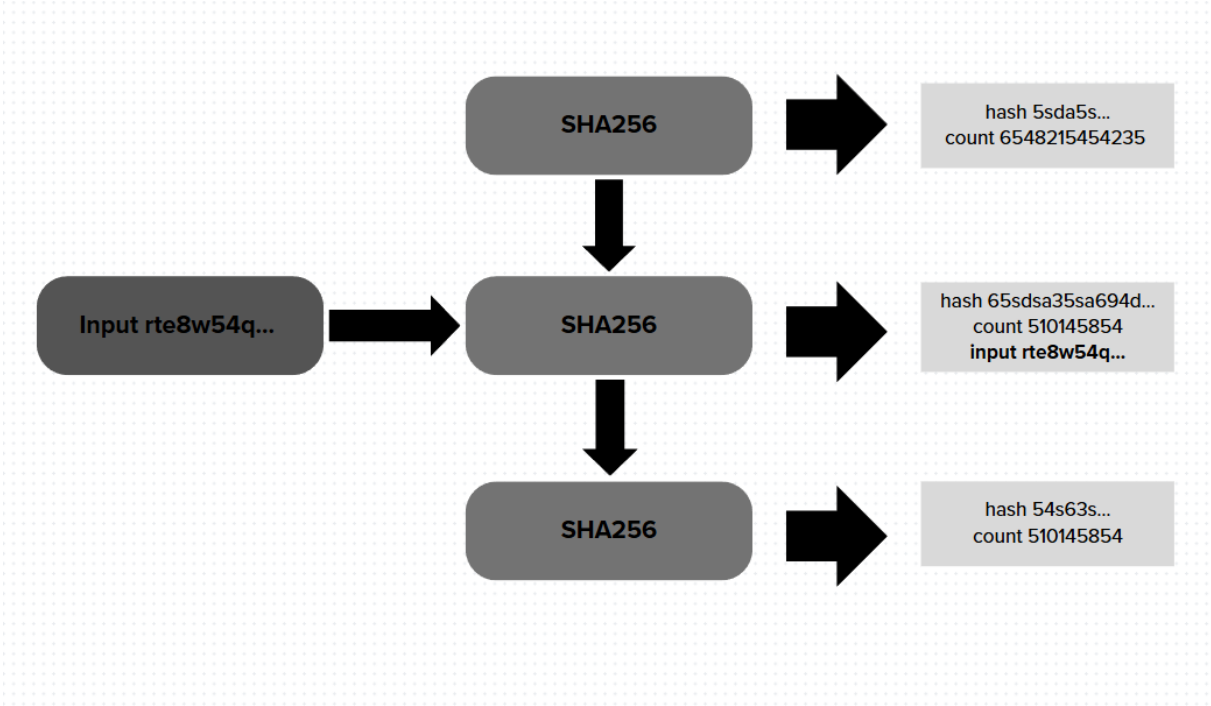


Figure3: Inserting data into Proof of History

4.3 Verification

The validity of the array can be verified by a multi-core computer in a fraction of the time it takes to build the array.

For example:

Core 1

Index	Data	Output hash
100	Sha256(hash99)	Hash100
200	Sha256(hash199)	Hash200

Core 2

Index	Data	Output hash
200	Sha256(hash199)	Hash200
300	Sha256(hash299)	Hash300

Given a given number of cores, such as a modern GPU with 4000 cores, the verifier can split the hash array and its indexes into 4000 slices and in

parallel ensure that each slice is correct from the start hash to the last hash in the slice. If the expected time to generate the index were as follows:

Total number of hashes

Hash for 1 core per second

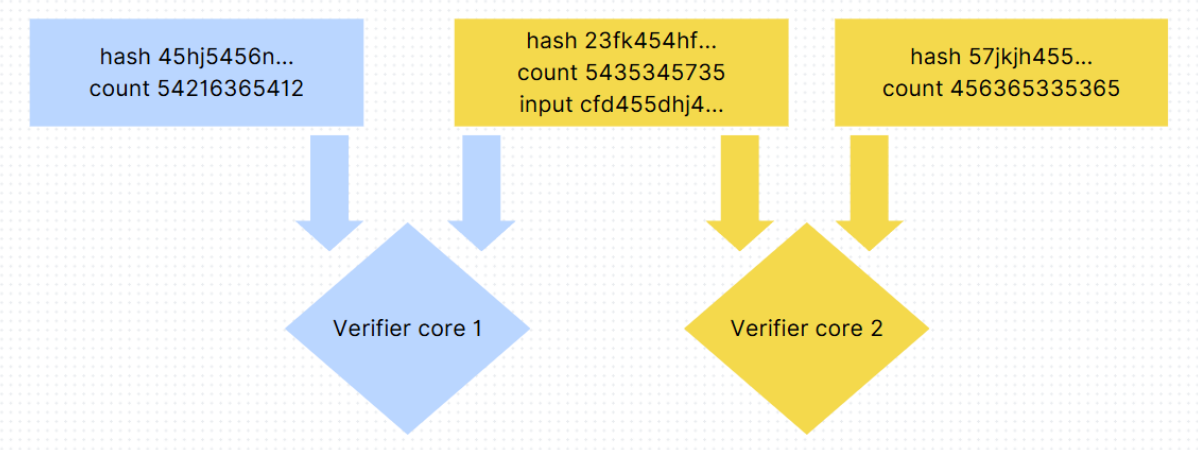


Figure 4: Validation by using multiple cores

The expected time to check that the ranking is correct will be:
Total number of hashes

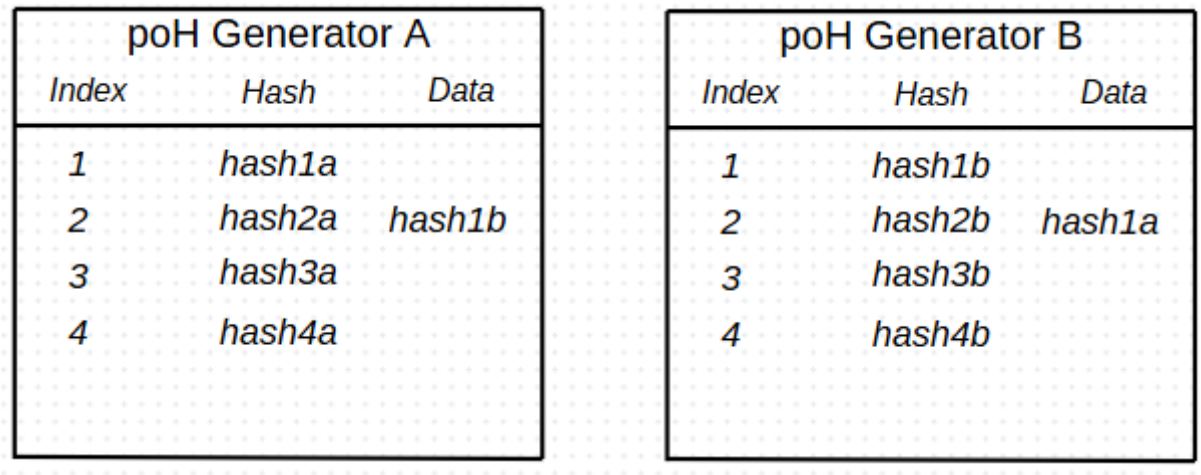
(Aggregation per second per core * Number of cores available for validation)

As in the example in Figure 4, each core can verify each slice of the array in parallel. Since all input strings are stored at the output, along with the counter and the state to which they are appended, the verifiers can replicate each slice in parallel. Yellow hashes indicate that the array has been modified by data insertion.

4.4 Horizontal Scaling

It is possible to synchronise multiple Proof of History generators by merging the queue state of each generator into the other generator, thus ensuring horizontal scaling of the Proof of History generator. This scaling occurs

without fragmentation. The output of both generators is required to reconstruct the full sequence of events in the system.



poH Generator A

Index	Hash	Data
1	hash1a	
2	hash2a	hash1b
3	hash3a	
4	hash4a	

poH Generator B

Index	Hash	Data
1	hash1b	
2	hash2b	hash1a
3	hash3b	
4	hash4b	

Given generators A and B, A receives a data packet from B (hash1b) containing the last state of generator B and the last state that generator B observed from A. The next state hash in generator A depends on the state of generator B, so we can infer that hash1b occurred some time before hash3a. This property can be transitive, so if three generators are synchronised via a single common generator A, B, C, we can trace the dependency between A and C even if they are not directly synchronised.

By periodically synchronising the generators, each generator can handle a portion of the external traffic. This allows the overall system to handle a greater number of events, which would otherwise be monitored at the expense of real-time accuracy due to network delays between generators. Global ordering can still be achieved by choosing a deterministic function, such as the hash value itself, to order the events within the synchronisation window.

The two generators in Figure 5 add each other's output state and record the operation. The colour change indicates taht data from the other has changed the sequence The generated hashes mixed into eacj stream are hlighlighted in bold. Synchronisation is transitive. There is a sequence in which events between A and C can be made avbailable A B C via A B B. In this way, sacaling occurs at the expense of availability. 10 x 1 Gbps links with 0.999 availability will have 0.99910 = 0.99 availability.

4.5 Availability

Users are expected to be able to enforce the consistency of the generated sequence and make it resistant to attack by adding to their input the last observed output of the sequence they consider valid.

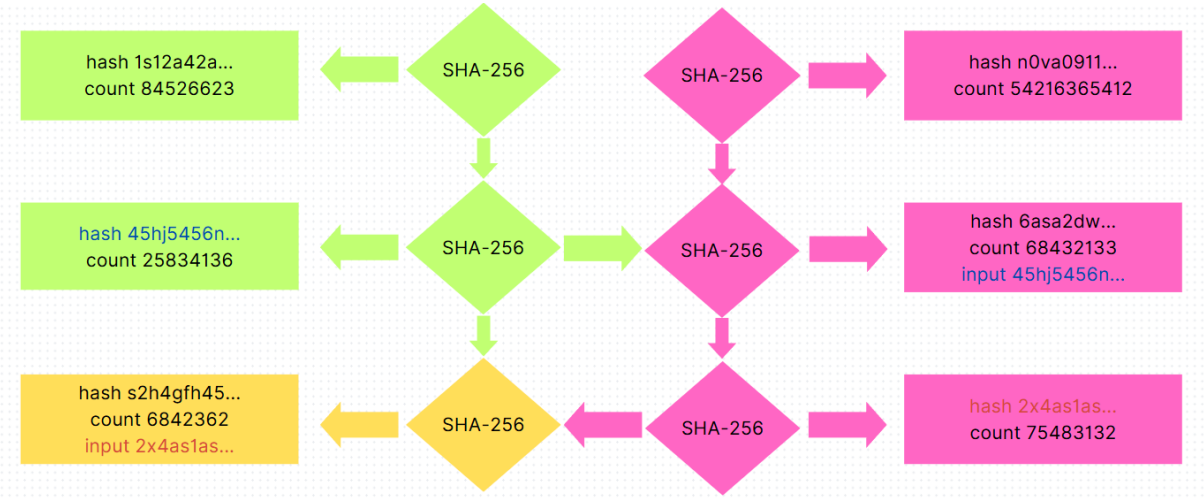


Figure5: Synchronisation of two generators

poH Sequence A		
Index	*Data*	*Output Hash*
10		*hash10a*
20	*Event1*	*hash20a*
30	*Event2*	*hash30a*
40	*Event3*	*hash40a*

poH Sequence B		
Index	*Data*	*Output Hash*
10		*hash10b*
20	*Event3*	*hash20b*
30	*Event2*	*hash30b*
40	*Event1*	*hash40b*

If a malicious PoH generator can access all the events at once, or generate a faster hash, it will also generate a second hash containing the events in reverse order. To prevent this attack, each event generated by the client must contain the last hash that the client observed from what the client considers to be a valid array. So when a client generates "Event1" data, it must include the last hash it observed.

PoH A Sequence

Index	Data	Output Hash
10		Hash10a
20	Event1 = append(event1 data, hash10a)	Hash20a
30	Event2 = append(event2 data, hash20a)	Hash30a
40	Event3 = append(event3 data, hash30a)	Hash40a

When the sequence is sent, event3 references hash30a, and if it was not in the sequence before this event, the consumers of the sequence will know that it is an invalid sequence. The partial reordering attack is then limited to the number of hashes generated when the client observes an event and when the event is inserted. Clients should then be able to write software that does not assume the sequence is correct for the short hash period between the last observed and inserted hash. To prevent a malicious PoH generator from rewriting client event hashes, clients can send a signature of the event data and the last observed hash instead of just the data.

PoH A Sequence

Index	Data	Output Hash
10		Hash10a
20	Event1 = sign(append(event1 data, hash10a), Client Private Key)	Hash20a
30	Event2 = sign(append(event2 data, hash20a), Client Private Key)	Hash30a
40	Event3 = sign(append(event3	Hash40a

data, hash30a), Client
Private Key)

Verification of this data requires signature verification and a search for the hash in the preceding hash sequence..

For validation:

(Signature, PublicKey, hash30a, event3 data) = Event3 Verify(Signature, PublicKey, Event3) Lookup(hash30a, PoHSequence)

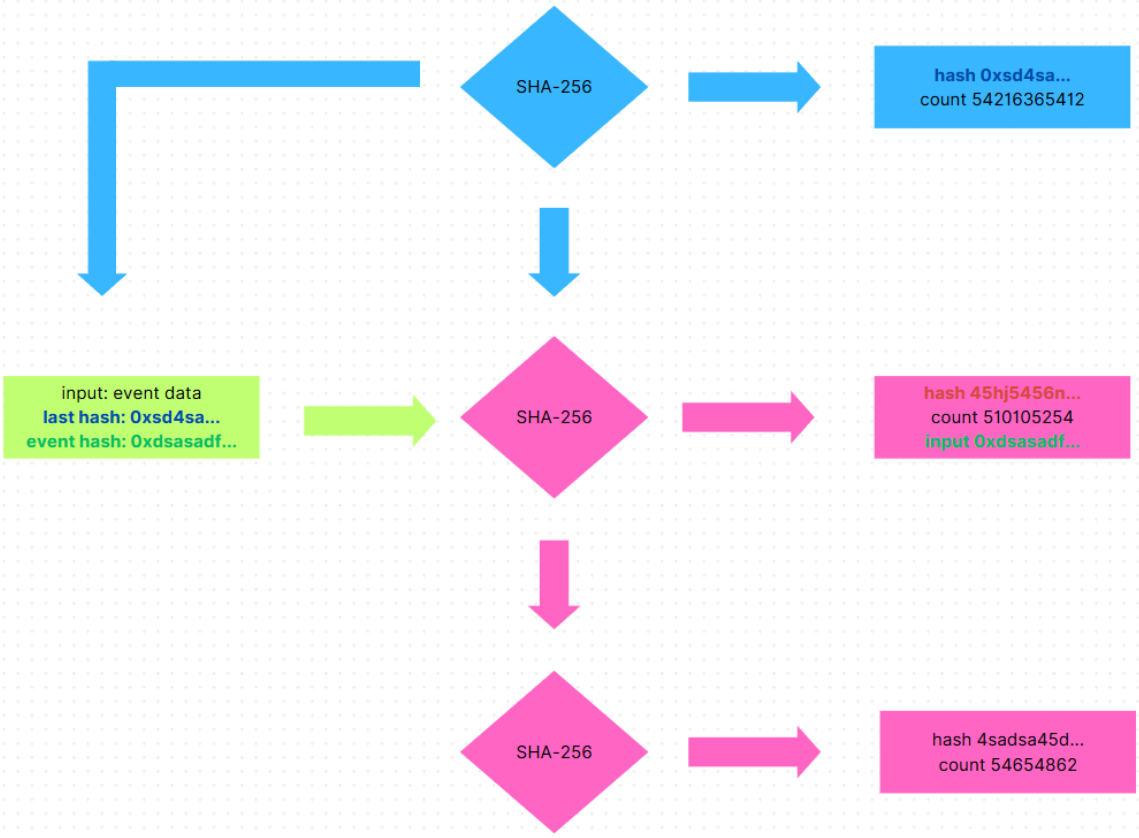


Figure6: Back referenced input.

In Figure 6, the input provided by the user depends on the hash 0xsd4sa... which exists in the generated array some time before it is added. The blue arrow in the top left-hand corner indicates that the client is referencing a previously generated hash. The client's message is only valid on an array containing the hash 0xsd4sa..... The red colour in the array indicates that the array has been modified by client data.

4.7 Attacks

4.7.1 Reversal

Creating a reverse order requires an attacker to start the malicious sequence after the second event. This delay is to allow any non-malicious peer-to-peer nodes to communicate about the original sequence.

4.7.2 Speed

Having multiple generators can make the deployment more resilient to attack. One generator can be high bandwidth and take many events to mix in its queue. Another generator can be a high-speed, low-bandwidth generator that periodically mixes with the high-bandwidth generator. The high-speed sequence creates a second sequence of data that an attacker must reverse.

4.7.3 Long Ranged Attacks

Long-range attacks include obtaining old, discarded client private keys and creating a fake ledger. Proof of history provides some protection against remote attacks. A malicious user who gains access to old private keys must recreate a history record that takes as long as the original record they are trying to forge. This requires access to a faster processor than the network is currently using, otherwise the attacker will never be able to capture the length of the history. In addition, a single time source allows for a simpler Proof of Replication. Since the network is signed in such a way that all participants in the network trust a single historical event record, PoRep and PoH together should provide both space and time defences against a forged ledger.

5-Games and Tokens

5.1 Introduction: Evolution of Games Economy

The game industry is undergoing a major transformation. Whereas in-game assets used to be limited to that game, blockchain technology now allows players to actually **own the digital assets they earn**. But this shift is far from complete. Most games still use **closed-economy models** and do not offer real ownership to players.

This is where **Tylix** comes in. Our goal is to provide players with a **decentralised**, secure and portable in-game economy. We are building a complete in-game economy not just limited to in-game coins, but supported by NFTs, staking mechanisms and DAO governance.

5.2 Initiate with Game Mechanics

Game mechanics determine how assets are used in and out of the game, the response rate required from the blockchain and ultimately the type of connection required. The first step in choosing a connection is to answer the question "What mechanics will my game include?"

Let's look at a few examples of how different mechanics work with different connections.

A read-only connection is sufficient for simple game mechanics that require only one-way between the game and the Blockchain. For example, looking at the cards in a player's inventory, or checking the costumes a player has for a particular character. Reading and writing is more convenient when the game mechanics require certain information to be loaded into the blockchain at certain times or with low frequency. For example, winning an in-game tournament can add new cards to the wallet of the top 10 players.

If your game relies on complex, unpredictable, high-frequency game mechanics, then a read-write constant would be the best. For example, if you have created an open-world game where card trading is a mini-game within the game world; here the gameplay depends on the frequent, unpredictable replacement or addition of the player's assets. Link types can be applied to each mechanic individually (for example, in an FPS game you can put skins on the blockchain, but you can decide not to put every bullet on the blockchain).

5.3 Why Should You Make Your In-Game Tokens On-Chain?

On-chain in-game coins incentivise ownership of real assets, as the key differentiator in web3 gaming. On-chain assets can participate in transparent, engaging game economies driven by smart contracts. While using an on-chain token (such as ERC-20 or ERC-1155) for your in-game coins adds complexity, the potential benefits may outweigh the additional effort. Tylix recommends considering at least one on-chain token for your game, provided your team has the right skills, including expertise in economy management and consumer marketing.

The benefits of using an on-chain token as your in-game coins are:

Automatic Balance Management: Balances are processed automatically and require no additional effort.

True Asset Ownership: Players fully own the asset both in and out of the game.

Reduced Double Spend Risk: All transactions are immutable, reducing the risk of double-spending.

Guaranteed Transactions: Trades are atomic, meaning that if any part of the trade fails, the entire trade is stopped.

Incentivize Development: Tokens can be used to incentivize developers and creators to build on top of your game.

Using on-chain tokens can create a more robust and engaging game economy, but it is important to balance these benefits against the added complexity they introduce.