Security Analysis of Blockchain Layer-one Sharding based Extended-UTxO Model

Cayo Fletcher-Smith[1] and Muntadher Sallal[1]

Department of Computing and Informatics, Bournemouth University, Dorset BH12 5BB, UK
s5117135@bournemouth.ac.uk, msallal@bournemouth.ac.uk

Abstract. Blockchain technology facilitates the transfer of digital assets, accomplished through the distributed storage of a transaction ledger, allowing peer-to-peer participant nodes to agree on valid transactions based on their local records without the reliance on centralised infrastructure or trusted participants. Distributed ledgers are increasing in public adoption, which can be attributed to the permissionless infrastructure and a rise in decentralised finance (DeFi) protocols. In this growth, shortcomings in throughput and latency have been highlighted, especially when compared to traditional payment channels. The extended-UTXO (eUTXO) model offers the untapped potential to support a functionally scalable infrastructure by adopting qualities of both the account model, and directed acyclic graph-structured UTXO model. We identified the unique benefits of eUTXO as: the ability to bundle the transaction processing of non-conflicting input states, achieving parallelism at the validator nodes; and the ability to implement complex off-chain scaling solutions through smart contracts. This research examines the security impact of sharding when applied alongside an eUTXO ledger. To illustrate this we introduce S-EUTO, a novel proof-of-concept state-sharding protocol. It leverages distributed randomness to ensure unbiased node-to-shard distribution and introduces an input/output cross-shard transaction architecture to maintain global state synchronisation. Our model demonstrates the potential of sharding alongside eUTXO without comprimising security.

Keywords: Blockchain scalability · EUTXO· blockchain security.

1 Introduction

Distributed ledger technology (DLT) was incorporated in a decentralized environment as a solution to these challenges in the Bitcoin whitepaper published by Satoshi Nakamoto, introducing the world to blockchain technology. Since its inception in 2008 it's become one of the fastest growing, disruptive financial technologies since the digitization of currency. Blockchain technology leverages a ledger of transactions stored in a distributed manor across a network of peer-to-peer (p2p) nodes [1]. Transactions within this network facilitate the transfer

of digital currency and assets, without the need for centralised trusted authorities. The trustless nature of transactions relies on the use of cryptography to prove the authenticity of transfers, allowing trust to be placed on the underlying protocol [2]. Each transaction is approved or rejected by the consensus of network participants and stored within blocks. Blocks are appended to the chain state and linked through cryptographic mechanisms ensuring the ledger remains immutable, traceable, and trusted by users [3, 4].

While flagships like Bitcoin handle decentralisation and security well [5], the lack of scalability proves too dire for meaningful impact in the financial sector with its 7 transactions per second (TPS) compared to 170tps of PayPal and the peak 56,000tps provided by Visa [6]. These shortcomings are also observable in smart contract projects like Ethereum with a throughput of 11tps. These scalability issues are often solved off-chain, using smart contracts to ship processing onto parallel networks with specialised scalability mechanisms; allowing the primary network to focus on decentralisation and security, while sidechains account for scalability [7]. Smart contracts and sidechains are considered off-chain processes as the logic and architecture sit above the underlying protocol at layer-two (L2). L2 operations present an imperfect scaling solution since sidechains often present unintended consequences, such as increased centralised processes, when hyper-focused on efficiency. Instead, equal and less severe scaling solutions should be considered within both the main L1 network, and subsequent L2 chains [7]. L1 solutions often revolve around an approach known as sharding, where network segmentation and state partitioning occur to enable parallel processing and increase throughput. In essence, the smart contract functionality at layer-2 allows for complex possibilities in the way of off-chain scaling, as seen in the Ethereum ecosystem; alongside the potential for transaction parallelism at the validator node level, due to predefined input states. These two potential methods of scaling are not possible in unison on: account based blockchains, due to ambiguous state dependencies (pre-validation); or UTXO based ledgers, caused by a general lack of logical expression (beyond Bitcoin script).

2 Problem Statement and Contributions

Scalability is a serious problem in the blockchain ecosystem. Our in-depth analysis of the literature (See section 4) shows that previous attempts to overcome the blockchain scalability issue were mainly focusing on sharding as well as Extended-UTXO. However, these previous attempts do not take into consideration the integration of sharding with Extended-UTXO as a scalability solution which may achieve optimal results. Considering the benefits of extended-UTXO in supporting scaling at both L2 and at individual validator nodes: we hypothesize that eUTXO has the potential to be a natural scaling solution, that enables various performance mechanisms throughout the blockchain stack. However, the potential for inherent scaling is further amplified if extnded-UTXO is deployed alongside sharded network consensus, allowing parallel transaction validation.

We theories that in such a model, validator nodes could individually process transactions with non-conflicting states in parallel; validation shards could further scale the throughput at the consensus and data storage level; and smart contracts could facilitate L2 solutions, ensuring the ecosystem can adopt new scaling mechanisms. We consider if *" Extended-UTXO in blockchain sharding can improve blockchain scalability without compromising security"*. The main contributions of this paper can be summarised as follows:
Security Evaluation: This paper examines whether Sharding based Extended-UTXO can be done safely, without increasing the likelihood of certain classes of attacks, for instance, inconsistency between shards caused by malicious nodes. We propose and evaluate a new Sharding protocol based Extended-UTXO (S-EUTXO). S-EUTXO is designed by integrating blockchain Sharding protocol with the ledger state model EUTXO which is proposed to overcome the issues of the UTXO ledger state model. In this paper, we have designed and run extensive simulations to evaluate security of the S-EUTXO protocol.

3 Background

In this section, we provide a general overview of the blockchain scalability. We focus on the blockchain scalability techniques by discussing the sharding and EUTXO.

3.1 Ledger State Models

Unspent Transaction Output Model In the unspent-transaction-output (UTXO) model, transactions generate unspent-outputs associated with the recipient's address, representing the transferred asset. UTXOs are spendable, with the associated wallet tracking the sum of all unspent-outputs assigned to the address [8, 9]. To start a new transaction, the unspent-outputs are declared as inputs. To use these outputs, the owner of a wallet must sign the transaction with their private key, signifying: (i) the outputs are spendable by the wallet, and (ii) the user authorises the transaction. The digital signature is known as a witness to the transaction [10]. Wallet addresses are derived from the public key, and can be explicitly linked to the private key that controls authorization. Once processed the transaction will generate UTXOs of equal value, assigned to the recipient's address. If the transaction only consumes a fraction of the input, the output of the transaction will assign remaining funds back to the sender. These processes can be conceptualised as paying in cash, and getting change returned to you. This process is illustrated as a directed acyclic graph in Fig 1.

Account Model The account model supports smart contract accounts (controlled by code), and externally owned accounts (controlled by private-key), containing account balances. When transactions occur, the sender's account balance has the transaction value deducted, then the recipient's account is increased by the same amount. The global state is then updated on the blockchain to reflect

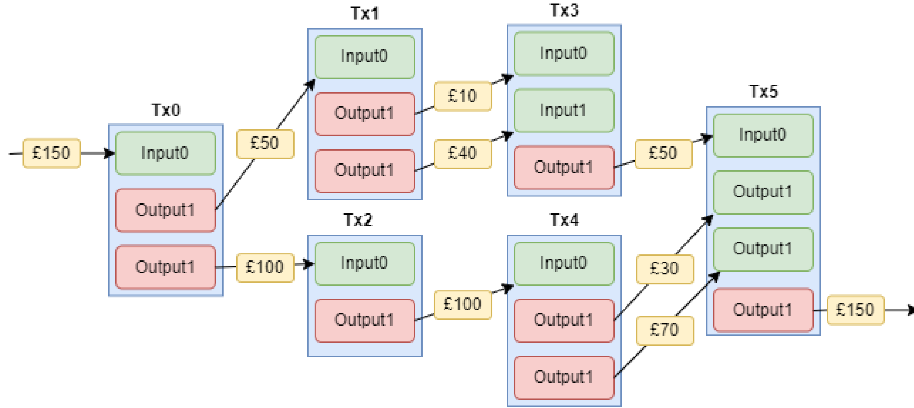

Fig. 1. UTXO Directed Acyclic Graph

the change [11]. Ethereum approaches smart contracts by isolating them under their own accounts. Both externally owned accounts (wallets) and contract accounts contain a public address, nonce, balance, storage hash and code hash. These account types differ in that the code and storage in wallets are empty, and contract accounts do not use private keys (because control is automated by the contract itself) [12, 13].

3.2 Sharding

Transaction throughput is difficult to scale at L1 since the requirements for participation in consensus produces high latencies with the involvement of numerous validators for sequential transaction processing. To solve this, sharding has been proposed by researchers as key feature to improve 'scale-out'. We define scale-out as the ability to continuously scale linear to network growth. Sharding is the process of segmenting certain network operations within the blockchain to increase performance. There are three types of sharding: (i) network sharding, (ii) transaction sharding, and (iii) state sharding. This section will provide an overview, detailing the motivations of each and their associated challenges.

Network Sharding Network segmentation is fundamental to both state and transaction sharding. The network is segmented into shards representing a smaller number of network participants. The challenge associated with this is maintaining BFT within isolated subnets. If the pre-shard network is designed to withstand Byzantine threats comprising 51% of participants, producing shards representing 10% of the network raises the threat of decentralised security mechanisms being inefficient in this centralised distribution [**?**]. If malicious actors overload individual shards operating local consensus, blockchain validity could be undermined by Sybil attacks. Sybil attacks can occur when an individual

actor represents more than the consensus threshold, allowing the malicious validation of transactions [14]. For this reason network sharding is a delicate process requiring unbiased mechanisms to dictate segmentation.

Transaction Sharding Transaction sharding is the process of segmenting the transaction pool and assigning transactions to the aforementioned network shards for parallel processing. Network shards validate transactions by achieving consensus independently, allowing transaction throughput to be increased by the factor of active shards. Transaction parallelism increases scalability, with the potential for linear scale-out. Isolated consensus may result in multiple submissions of the same transaction to different shards, resulting in double spending [**?**]. Atomic commit protocols and shard communication are critical to mitigate these threats and avoid conflicts [15]. The mechanism used to assign transactions to shards must be clear and without conflict.

State Sharding State partitioning segments blockchain storage into smaller states, assigning partition maintenance to network shards. State distribution reduces strain associated with ledger storage which can reduce computational overheads associated with validator internal processes. Furthermore, operating on segments of the global state results in smaller transmission messages when bootstrapping nodes per epoch (Doton et al. 2021). Bootstrapping is the process of updating validator nodes, occurring on node reassignment [14].

3.3 EUTXO Model

The extended-UTxO model is an improvement on the UTxO model introduced by Nakamoto in Bitcoin. eUTxO introduces smart contract implementation which is not applicable in UTXO models previously. When smart contracts are compiled, a binary output is produced. This output is hashed, and used as an address to the contract, on-chain. Script addresses, don't have a public/private key pair. This is conceptually similar to smart contract accounts, in the account model, not having private keys to control transactions.

4 Related work

This section focuses on the research related to scaling the transaction throughput and latency in blockchain systems. Typically there are two categorical approaches to this, off-chain processes at layer-two (L2) and on-chain scaling at layer-one (L1). L2 scaling leverages smart contracts to offload transactions onto specialised side-chain infrastructures to increase transaction throughput. In this model, sidechains process transactions, and store the proof of validity on the main network, thereby inheriting the security and decentralisation of the parent chain.

L1 processes focus on adjustments and optimisation to be placed on the peer-to-peer operations, consensus mechanisms and underlying ledger state model of the main network. Sharding is the main approach for achieving significant scaling at L1 through network segmentation and network parallelism. This section mainly focuses on research related to L1 sharding and scalability.

4.1 Elastico

Luu et al. proposes Elastico, a Nakamoto-style sharding approach to network and transaction parallel computation [16]. Operating in epochs with interludes of node reassignment, network segmentation is intended to infinitely scale proportionate to network growth. Global state is maintained by a single randomly elected shard. Node assignment is based on verifiable PoW identity generation, randomising allocation and constraining actors' ability to overwhelm shards. Assignment occurs uniquely at random depending on the value of predetermined bits in the output hash from the proof-of-work algorithm [16]. While scaling increases linearly, local shard BFT is sacrificed.

4.2 Omniledger

The works of Kokoris-Kogias et al. detail a fault tolerant protocol, incorporating concepts from Elastico [15]. The main research contributions presented are: (i) cross-shard transaction atomicity, mitigating double spending and conflicts in global state maintenance; and (ii) verifiable random functions (VRF) for reassignment. Instead of binding distributed randomness to a biasable PoW function for node assignment, OmniLedger incorporates RandHound which relies on the temporary election of a central coordinator. Each node generates a VRF ticket using their private key and broadcasts it throughout the network, they then accept the creator of the lowest value ticket as the temporary coordinator to run RandHound [15]. The coordinator generates a seed dictating the parameters of the next epoch, based on the input VRF tickets distributed from the network. Therefore, epochs can be dictated democratically and verifiably without assumptions being made about the quality of coordinator [17]. Atomic commit operates in three stages, illustrated in Figure 7, ensuring transactions are either fully committed once or aborted, mitigating double spending.

1. **Initialisation**: Coordination is leveraged client-side where a final commit shard is designated, and the transaction is submitted to multiple input shards.
2. **Lock**: The input state is locked in each shard and validators execute an internal consensus function to either accept or reject the transaction. A verifiable hash traceable to the shard is produced, specifying either proof-of-acceptance or proof-of-rejection
3. **Unlock**: The client now has the power to commit once or abort the transaction. If the client holds proof-of-acceptance from all input shards, they issue an unlock-to-commit message containing the proofs and initial transaction.

The proofs are verified and the input state is unlocked for the transaction to be committed. If one or more shards returns proof-of-rejection, the client forwards this to all input shards where the state is unlocked and returned for future transactions (Kokoris-Kogias. 2018). If shards become unresponsive post-lock, the transaction is aborted at the end of the epoch.

4.3 RapidChain

Operating in epochs of reassignment, RapidChain is a synchronous protocol with atomic-commit functionality where state is partitioned between shards [18]. Cross-shard transactions rely on shard coordinators and input/output shards alongside a reference committee that generates epoch randomness [19]. Epoch randomness dictates the assignment of nodes and state within an epoch. Generated transactions are sent to a random node then forwarded to the appropriate output shard concerned with maintaining the state output of that transaction. The output shard coordinator gossips the transaction to shards responsible for the input state where consensus will be achieved in each, similar to Omniledger's approach [14]. In principle, the output shard is acting as a trusted party, although the power of commit relies on compliance from the input shards [19].

4.4 Chainspace

Similar to Omniledger, Chainspace offers cross-shard transactions [20]. As opposed to leveraging the client for temporary coordination, Chainspace's S-BAC protocol, a combination of Byzantine agreement and atomic commit, relies on peer-to-peer communication between shards. Operating an account model, immutable objects are used to represent the state of wallets on-chain, when transactions are approved the object is destroyed and re-initiated with the updated balance [21]. Transactions are conceptually similar to UTXO implementations, requiring an input object, which is destroyed, returning the new output object. In this case the entire account is used as an input, as opposed to a specific unspent-output. Transactions are sent to input-shards maintaining the associated object's state, each shard runs consensus, communicating the result of their individual validation between each other. The associated shards then run another round of consensus to inactivate the input-objects, before forwarding the transaction to a commit shard where the object is re-initialised [22].

4.5 Monoxide

Monoxide performs network segmentation, for transaction parallelism, and state partitioning, reducing storage overheads [23]. Account states are assigned to shards based on parameters on the bits of an output hash from the account's public key. Transactions affecting accounts split between shards rely on relay cross-shard transactions to update the account states in the neighbouring shard [23]. Consider a transaction from Alice to Bob, where Alice's account is maintained in shard A and Bob's in shard B. Alice derives the shard address from her

public key, sending her transaction to the location storing her account. Shard A performs consensus and updates Alice's balance, forwarding a relay transaction containing proof of consensus to shard B where Bob's balance is updated.

4.6 Zilliqa

Zilliqa employs network and transaction sharding on an account-based design [24]. Shards performs a PoW function determining their shard assignment. Operating a reject-and-retry mechanism, Zilliqa adopts security benefits associated with cross-chard transactions without the atomic-commit complexity found in similar projects. This is achieved through parameters based on sender addresses defining what transactions shards process. If transactions are sent to the wrong shard, nodes return a reject-and-retry response. This approach mitigates double-spending attacks without implementing a complex coordination mechanism for cross-shard consensus. Blocks are committed to a directory shard, that maintains the global state, in the form of "macroblocks" [24]. Signature aggregation combines digital signatures from all validating nodes, into one smaller signature and is sent alongside the macroblock. The aggregated signature is verified through reconstruction using the public keys from the macroblock's validator nodes [24]. Aggregation lowers network overheads by eliminating network traffic associated with nodes sending digital signatures individually. The directory shard does not reiterate the same validation, instead it verifies the aggregated signatures from validating nodes. Zilliqa does not allow for smart contract parallelisation since transactions affecting the same contract are processed sequentially. As the network grows, and adoptions increases at L2, dApps become bottlenecked to the processing power of a singular shard, as opposed to their optimisation requirements [25].

4.7 Extended-UTXO

This section covers the designs of state models categorised under the extended-UTXO umbrella. We do not review sharding approaches here as network segmentation has not yet been implemented at L1 on an extended-UTXO blockchain.

Cardano In the works of Chakravarty et al. a smart contract-capable implementation of the traditional UTXO model was proposed for the Cardano platform [10]. This extended-UTXO model introduces logic functionality, traditional to account-based implementations, to the UTXO model. In principle UTXO models operate a lock and key mechanism where public keys lock transaction ownership, and verifiable signatures provide the key to unlock and spend associated outputs. In eUTXO, addresses may contain logic that fulfills the same purpose as signatures (keys) by specifying the conditions for spending locked outputs [10]. Transactions may contain arbitrary data called the datum, alongside outputs, that can determine how the logic at the output address operates when the funds are used as an input. Contract logic defines additional parameters regarding

how outputs can be used, these parameters are checked at validation when a transaction uses the locked output. When the output is consumed as an input, additional parameters are passed within the transaction. These parameters are conceptually similar to function arguments; we call this the redeemer [26]. Redeemers dictate the parameters the contract operates under, and dictates how the logic will execute when validated. Imagine a scenario where assets are locked on a vesting schedule, and users can periodically unlock their assets after a period (specified in the datum) has elapsed. The redeemer arguments could be 'claim', signifying the user wishes to unlock their assets. This argument tells the contract what to do if the request is valid. In this case, the request is valid if a predefined period has elapsed since the assets were initially locked, and the user is entitled to unlock them. This implementation allows for the validity of transactions to be checked off-chain since the dependencies (inputs, datum and logic) are predefined. Therefore, transaction success is guaranteed if dependencies are unchanged upon validation on-chain. Since the blockchain is in motion transactions may not always succeed due to the state dependencies being invalid by the time validation occurs. The nature of transactions only being dependent on themselves, and their inputs protects them from external on-chain states that may causes unexpected consequences [10].

Ergo Ergo implements boxes into their state model, which are conceptually similar to unspent outputs [27]. Each box is registered to an address under the control of a private key. Boxes can only be used in an operation once, with the only possible operation being a transaction. Once used, they are marked as spent and new output boxes are initiated [28]. Each box contains 10 registers, 4 are designated for mandatory data and always remain full while the remaining 6 are reserved for client-defined data and may remain unused [27]. Mandatory data registers include the monetary value, serialised script protecting the box, asset types stored in the box, and transaction information. Transaction information includes (i) the creation size; (ii) a unique identifier associated with the transactions creating the box; and (iii) index of the box in the outputs created by the transaction [29]. Transactions in Ergo can take multiple inputs and create multiple outputs, essentially bundling transfer processes together.

Nervos Nervos operates using a hybrid UTXO account-based adaptation most easily classified under the niche eUTXO umbrella. Instead of using unspent-outputs, which can be restrictive on data structures, Nervos employs cells to act as the inputs and outputs of transactions [30]. Cells are immutable and contain arbitrary data, which could be state (such as tokens) or executable logic. Contrasting account-model implementations, where state is the internal property of a smart contract, cells can reference data in other cells, allowing for assets and governance logic to be separated. Application logic is split into two phases, generation and verification, running in different places [31]. This allows for additional algorithmic flexibility between phases.

The generation of new state (transaction) is run locally, client-side, and broadcast to the network. Assets stored in cell states must follow associated logic, specified by scripts. Verification assesses the generated state variables stored within the transaction. State variables include a reference to the previous cell storing the state which can only be used once as an input, and a signature verifying the client owns the input. Associated script logic dictates the parameters of cell usage and is executed by validators locally within the virtual machine (VM). If transaction variables coincide with the parameters of use, the validation is successful. The VM executes the type script upon cell creation to ensure the state is valid under certain parameters. The lock script is executed taking proof arguments when the cell is referenced as a transaction input [31]. Cells used in transaction inputs are removed from the active set of states, with output cells being created and included in the amended active global state. In this approach, similarly to UTXO projects, transactions act as the proof for state transition [30].

5 The proposed model

The proposed model, named as S-EUTO, is a novel smart contract-capable protocol, with state partitioning and transaction parallelism on an extended-UTXO model. The transaction pool is segmented and assigned to specific shards based on certain transaction parameters (see Section 5.5), and validated in accordance with the local shard state. Fig 2 illustrates the various layers of this protocol, from the transaction pool to validation in accordance with the UTXO directed acyclic graph. The following system model can be used as a reference point to understand, in high levels of abstraction, how the mechanisms fit together. In section 5.4 we illustrate the smart contract architecture implemented, and discuss the relationship between contract parameters and output states. In Section 5.5 we examine the unique challenges of applying sharding alongside eUTXO, discuss how the aforementioned contract architecture works contextually to sharding and provide insight into the mechanisms dictating node distribution.

5.1 Addresses and Ownership

Externally owned addresses have three key attributes: address, public and private key. UTXOs are verifiably associated with addresses and controlled by the user's private key. To generate key pairs, we used RSA asymmetric cryptography algorithm from the *cryptography.hazmat.primitives.asymmetric* libraries. We set the padding.PSS salt used in cryptographic processes to max length, ensuring encryption remains resilient to reverse engineering [32]. Each key pair is 2048bits in size, allowing for slightly shorter verification times compared to 3072bits or 4096bits, with the public exponent set to $e = 65537$ for compliance with cryptography standards [22].

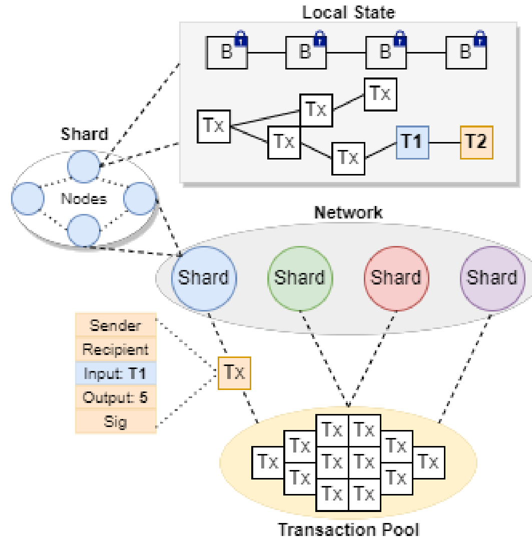

Fig. 2. System Design Architecture

5.2 Transaction Structure

This section outlines the transaction data structures, digital signatures and authenticity requirements, and input output calculation. Transaction identifiers are unique integers generated by validating nodes before block minting. Identifiers are referenced in the transaction input field of a subsequent transaction if the corresponding unspent output identified fulfills expenditure parameters. The sender address defines the transaction origin wallet and is used to find the associated sender public-key. The recipient address dictates the new ownership wallet controlling the specified assets upon transaction approval.

The token field specifies the unique identifiers associated with asset types; this dictates which asset the output field represents and the parameters of which inputs can fulfil the transaction. The output field represents the quantity of tokens being transacted. A digital signature contained in each transaction is produced by the sender using their private key and wallet address as input. The script, datum hash, datum and redeemer attributes are smart contract-dependent fields that remain empty in normal transactions.

Digital Signatures and Authenticity Digital signatures are produced using the sha256 hashing algorithm, predefined input, padding, and private key of the sender. The predefined input is a UTF-8 encoded byte string, produced from the value of the sender's address. All participants know this value and verify the authenticity of transactions using this input and the sender's public key.

Input Declaration Input calculation occurs in the validation function run by validating nodes. The node reads the output history associated with the sending address of a new transaction. It compares each transaction identifier from the history with the spent outputs stored in virtual memory. If the unique identifier does not match any spent object, and the output fulfils the input requirements of the new transaction, the validator appends the unspent output identifier as the input of the new transaction.

Output Calculation Output calculation by default is defined client-side by the sender. The exception to this is if the validator selects an input exceeding the specified output amount. In this case, a return transaction containing the value difference between the input/output is created and validated alongside the new transaction. The return transaction recipient is the sender of the new transaction (the owner of the input exceeding the output value).

5.3 Validity Conditions

When nodes validate the entire chain state, the bytes of each block are calculated, and the recreated output hash checked against the previous hash pointer of the next block. This ensures nodes can determine chain validity, and identify conflicts. In the event of conflict, there is no mechanism for resolution, such as hard forking or state roll-back, although this may be implemented in future research.

Before appending blocks, the chain validity is verified by computing the hash pointers, and the block is verified according to the ledger. The verification process for new blocks encompasses each validator verifying the included transactions according to the transaction validity conditions. Validators independently execute these validity functions, before aggregating votes and achieving consensus according to the consensus parameters.

Transaction Validity Transaction validation is determined based on the validity of the input used. Inputs are valid if four key parameters are met:

1. The input transaction is owned by the sender's address
2. The input UTXO is $\geq$ to the specified output
3. The digital signature is verifiable to the sender's key
4. The input identifier does not match any spent input previously used on-chain

Validators verify the signature by extracting the public key associated with the sender and creating a hash using the predefined global input parameters. This hash is checked against the digital signature included in the transaction.

Input validity is calculated by isolating the transaction history associated with the sender's address and iteratively locking a received transaction. Once locked, the spent transaction history is filtered for spent UTXOs associated with the sender. The locked transaction's identifier is checked against each associated

spent transaction. If no matches are found and the UTXO fulfils the output requirements the validator deems the input valid. If the locked transaction matches the identifier of a spent transaction, or does not fulfil the output requirements, a new transaction is locked and the process is repeated.

These conditions are mandatory for all transactions aside from validator-generated return transactions, which return any remaining assets from an input back to the sender. It was unnecessary for additional complexities related to return transactions (such as signatures), since the return transaction is the direct output of an already validated transaction. While these conditions are true across all transactions, transactions referencing smart contracts may have additional parameters necessary for expenditure 5.4.

Consensus Parameters Consensus parameters define how independent nodes perform validation functions and agree on the result in a Byzantine network. Transaction validation and block population occurs on the minting node; upon reaching the threshold for block size, blocks are broadcast to validators. Validators check the blockchain state validity, validate each individual transaction, and reach a binary conclusion of true or false. This conclusion is broadcast to consensus participants; if the aggregation of all votes is above the consensus threshold the network is updated with the new state.

5.4 Smart Contract Architecture

Smart contracts are deployed under contract addresses. Unlike wallets smart contracts are not controlled via a public private key pair, since assets are only conditionally bound to the address the transaction references, not owned. Contract addresses contain immutable logic, stored on-chain. Contracts can be referenced by transactions between network participants, thereby altering the parameters of output spending [33].

Smart contract logic only impacts how associated inputs are used. Transactions can be sent from one participant to another without the requirement of meeting script parameters. When the recipient of a transaction referencing contract logic spends that output, additional parameters are applied to the validation based on the script referenced. This is accomplished through the transaction attributes:

1. script
2. datum hash
3. datum
4. redeemer

If the sender of a transaction wishes to apply additional parameters to the output, the smart contract address is referenced in the script field. We refer to the sender as the initiator, since they instigate the transaction. Arbitrary data is passed in the datum field, which can dictate the parameters the output is used under, while the datum hash verifies the signature of the sender and the arbitrary

data as being authentic. The datum hash is always necessary when referencing a contract, while the full datum can optionally be included in the initiator transaction (contract dependant). The initiator transaction follows standard validation requirements.

The transaction recipient is called the redeemer, and must meet contract parameters to redeem and spend the output. Redeemers must include the script reference, their datum hash, the full datum, and additional script arguments in the redeemer field. By verifying the datum hashes provided by the initiator and the redeemer, the script ensures the data is authentic. The full datum is optional in the initiator transaction but not the redeemer transaction, as the script must be passed the full datum at some stage to compute datum hash validity. As a method of privately passing data on-chain to the redeemer, the initiator can encrypt the datum using the redeemer's public key. The datum may also be withheld by another contract for a vesting period, locking the redeemer's assets. Finally, the redeemer must include script arguments in the redeemer field, determining how the input should be used. Parameters depend on the structure of contract logic and the expected script arguments. The architecture of these transactions is illustrated through an expansion on the directed acyclic graph in Fig 3.

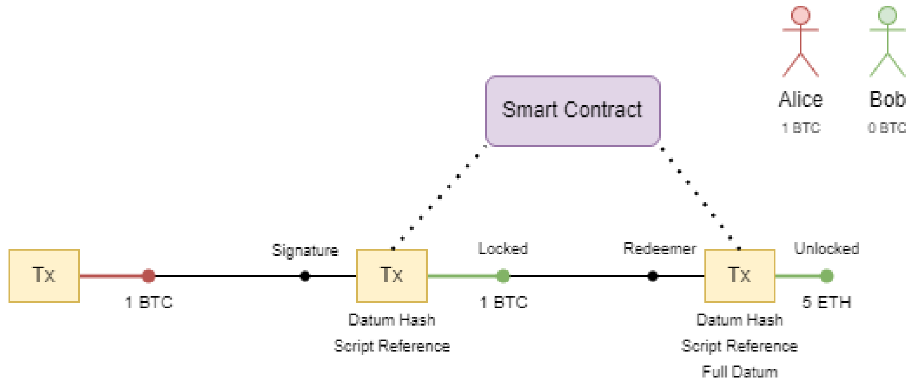

Fig. 3. S-EUTO: Smart Contract Architecture

Example Exemplifying this process, contextual to a currency exchange would be: the initiator specifying the amount while the redeemer specifies the output asset class.

Logic Incorporation and Spending Conditions Smart contract logic sits above on-chain validation functions in that primary validation must return true, before smart contract validation occurs. In the result a transaction references a smart contract, both the primary validation function and contract logic must return true for the transaction output to be spent.

5.5 Network Sharding

Network segmentation in S-EUTO assigns network participants to specific validation groups. Each node maintains a routeing table detailing all other nodes within their specified shard used for communication associated with distributed consensus functions.

Introducing sharding to eUTXO presents some additional challenges that must be considered in the design architecture of the system. One key consideration in any sharding protocol is the maintenance of the global state, ensuring individual shard state does not folk between epochs of bootstrapping and redistribution. Maintaining the state is critical in ensuring UTXOs are either spendable or spent across all shards, and smart contracts logic accesses the most recent version of the ledger to avoid exploitation.

To maintain the eUTXO state, we propose the use of state partitions, where individual shards maintain the state of a set amount of external addresses. To enable shards to distinguish between transactions in the pool, the public key of a transaction sender determines the shard responsible for validating the transaction. This can be defined before submission with public data from the blockchain state, calculated by the most recent redistribution epoch. Considering that redistribution is determined by distributed randomness, transaction to validator assignment also maintains randomness. If a transaction reaches an incorrect shard, the recipient node will respond with an error code, indicating transaction rejection. This mechanism ensures UTXO transactions and shards have a uniform way of interacting with each other, and transactions reach the appropriate validators.

Shards associated with the state of a transaction sender will achieve consensus on the validity of that transaction, these are referred to as input shards. Upon reaching consensus validator signatures are aggregated and sent as proof to the output shard that maintains the state associated with the recipient address. The output shard achieve consensus on the validation proof, and generates a UTXO tied to the recipient. To facilitate smart contract transactions, our model requires that shards maintaining the state of smart contract address also receive proof of validation from the input shard. This enables smart contracts to update their state with the transaction data critical to the instance. When redeeming a smart contract transaction the shard maintains the newly generated output, produced by the initiating transaction, achieves consensus on the output spending then forwards the transaction to the smart contract shard where additional parameters are applied, in accordance to the contract state. The smart contract shard then responds with proof of validation, allowing the redeeming transaction to continue in validation. The approach effectively makes a shard that is responsible for an associated contract state aware of the existence of a transaction, while ownership is still withheld in the output shard maintaining the state of the redeemer's address. We illustrate this process in 4.

Shard Assignment and Node Redistribution To determine each shard's participant nodes, we propose a distributed randomness function (DRF) to pro-

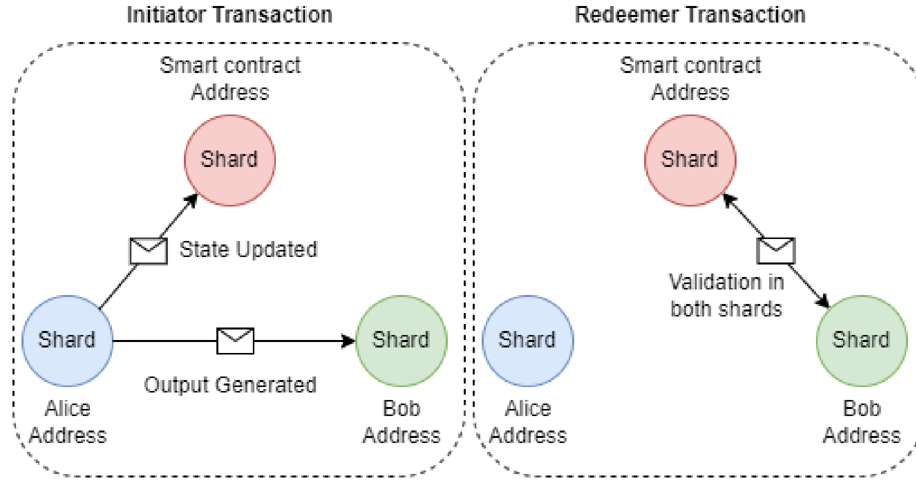

Fig. 4. Cross-shard Smart Contract Transactions

duce unbiased node allocation. DRF executes within each shard independently, operating in iterations equal to the number of nodes per shard. Each iteration operates in two phases: argument collection, and ticket generation, requiring three participants: a temporary coordinator node, an operator node, and subject node. At the start of the function, each node randomly generates an input ticket and an operator argument. A coordinator node is elected each iteration based on tickets generated from the previous epoch. In the argument collection phase, the coordinator randomly selects two nodes from the shard route table and defines each of them as either the subject node or operator node for this iteration, broadcasting the selection to the network. Upon receiving the broadcast message, the reallocation node responds by broadcasting their input ticket while the operator broadcasts their argument back to the coordinator. In the ticket generation phase, the coordinator randomly selects one of the DRF predefined functions, which takes the provided inputs and aggregates them, producing the final output ticket for the reassignment node. The output ticket, function and inputs used in its generation and associated reassignment node are broadcast to the shard. Shard participants external to the process observe all stages through the message broadcasts, allowing for traceable and verifiable generation. At the end of the iteration, direct participants are removed from potential selection for the role they filled. Throughout the DRF execution all nodes act as the operator, coordinator, and reassignment node once. On DRF completion, all tickets are broadcast to the entire network, where node reassignment occurs based on the byte value of tickets. Three iterations of this process is visualised in Fig 5. Dotted lines indicate broadcasts, while full lines indicate specific communication between iteration participants. Based on initial simulation results, we further optimised this process by implementing overlapping iterations. In this overlap

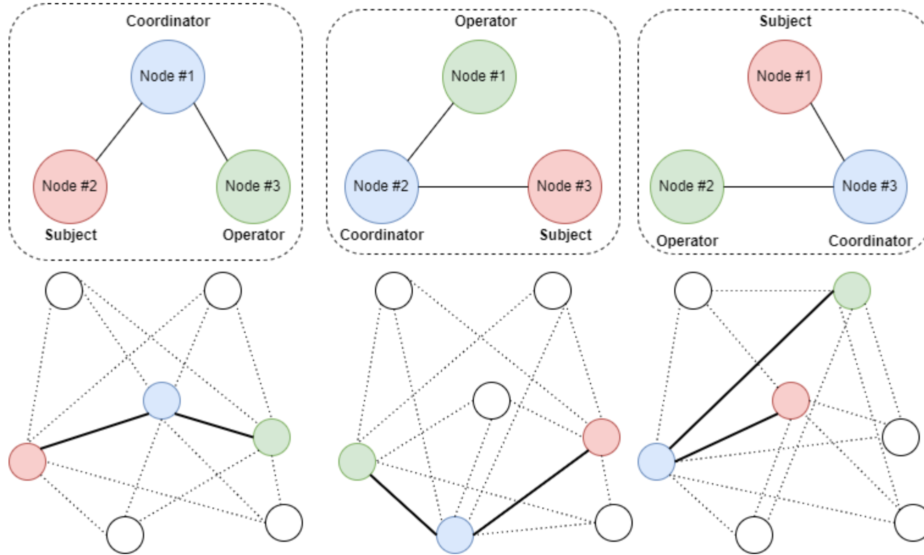

Fig. 5. Iterations of Distributed Randomness Function

feature, the next iteration starts as soon as the next predefined coordinator (based on tickets) receives the initial broadcast from the current coordinator. It may be possible to execute iterations in parallel, with nodes performing all roles simultaneously, although we leave this to future research.

6 Evaluation Methodology

The evaluation was conducted on quantitative data generated through simulations. The measurements we identified in this research were redistribution latency (R_L) and number of malicious validations. Based on link latency data extracted in [34, 35] from the Bitcoin network we define the edge-to-edge latency at 2500 milliseconds . We used this data to inject realism into our network environment since the data was extracted from a blockchain with comparable implementations to our model, such as transaction size T_s and state model [36].

We extracted measurements under the following variable parameters:

s =number of shards
m =number of malicious nodes
S_n =nodes per shard
B_s =block size
T_{pool} =transaction pool (network usage per second)

Equation 1: We factored in the impact of distance between participants and message size when calculating C_L. The distance between participant nodes

was calculated using the following equation:

$$d = ((x_2 - x_1)^2 + (y_2 - y_1)^2)$$

The simulation environment maintained an edge-to-edge distance of $d = 50.9$ with a propagation delay of $2500ms$. Therefore we assume $d = 1$ to have a latency of $49ms$. We introduced artificial serialisation delay to evaluate the relationship between B_s and C_L. Serialisation delay S_d is latency associated with serialising a packet into a transmissible format and waiting in a traffic queue for appropriate bandwidth to transmit. We assumed S_d to incur 5.12µs delay per 64-byte packet under a throughput of 100Mb/s [37].

Equation 2: Block size B_s was calculated using the following formula:

$$B_s = T_{pb}(T_s) + E_{bs}$$

where T_{pb} is transactions per block, T_s is transaction size and E_{bs} represents the empty block baseline size based on mandatory attributes such as index, timestamp, and hash.

Method 1: We evaluated our intra-shard consensus by implementing malicious transactions (M_T) alongside introducing malicious nodes. M_T were illegitimate for one of several reasons:

1. invalid output value
2. wrong signature input
3. wrong signature private key

These values were selected randomly before transaction submission to assess various parameters of the validation function.

Equation 3: We used the following equation to represent the threshold of malicious nodes the network can securely contain under various levels of shard deployment.

$$G_r = m \leq \frac{m}{n}(n)$$

6.1 Simulation Structure

We considered discrete event simulation (DES) where state is constant between events since it is commonly applicable for computer systems since the state entirely depends on events, indicating changes in the system. In practice, simulations progress directly to the next event, bypassing the actual time taken to complete the processes [9]. Since blockchain state remains unchanged between processes we implemented DES to model the blockchain system, allowing complex processes to be expressed as simple sequential event cycles, simplifying mechanisms such as consensus and redistribution. Such events included transaction propagation, redistribution iterations, and consensus processes [2, 38].

Table 1. The percentage of the network an adversary controlled where m represents the malicious nodes, and n represents the network size(total number of nodes.)

Malicious Parameters	
$m = 0.1$	10% of n
$m = 0.2$	20% of n
$m = 0.3$	30% of n
$m = 0.4$	40% of n

Table 2. Five levels of shard deployments where s represents number of shards, and S_n represents the percentage of nodes per shard.

Shard Deployments	
$s = 4$	$S_n = 0.25(n)$
$s = 8$	$S_n = 0.125(n)$
$s = 12$	$S_n = 0.083(n)$
$s = 16$	$S_n = 0.0625(n)$
$s = 32$	$S_n = 0.03125(n)$

6.2 Experiment Setup

Malicious node thresholds determined the percentage of the network an adversary controlled. This was used to identify shard failure and the overall level of Byzantine fault tolerance in the network. We did not implement a threshold above 50% since Byzantine fault tolerance would be compromised by default. Thresholds are detailed in Table 1.

We introduced five levels of shard deployments ranging from conservative to extreme. This was used in conjunction with the malicious thresholds to identify the baseline adversary model required to cause shard failure at each deployment, outlined in equation (**Equation 3**). The parameters of shard quantities s and nodes per shard S_n are detailed in Table 2.

Network usage was defined as the number of transactions generated by clients and sent to the transaction pool T_{pool} per second. Usage thresholds were used in conjunction with B_s and s deployment parameters to generate transaction latency data using method (M2). These were the following parameters: $T_{pool} = (1000, 2500, 5000, 10000, 20000)$.

Each test-case was run across 10 epochs of redistribution, ensuring statistical relevance, with varying unique parameters per simulation. Latency tests were executed with 1000 measured transactions per epoch.

Baseline Data This section outlines how we defined the baseline data to calculate the measurements used in our evaluation.

We used the *sys.getsizeof* package to identify T_s and $E_b s$ in bytes of the associated data structures.

We identified the baseline block size as $E_b s = 232$ (bytes) and the transaction size as $T_s = 360$ (bytes). This was necessary to determine latency incurred in block and transaction propagation and produce accurate simulation results. This data provided a baseline to determine the block capacity, outlined in equation (**Equation 2**).

We used the same method to identify the sizes of messages per iteration of our distributed randomness function. We had four key messages: coordinator assignment message $C_{assignment}$, subject response $S_{response}$, operator response $O_{response}$, and coordination ticket issuance C_{ticket}.

Smart Contract Test Structure We tested our extended-UTxO smart contracts by probabilistically instigating contract instances throughout the simulated network usage. This was achieved by modifying a preset transaction, before being submitted by the client, to initiate an instance. The initiating transaction referenced the associated test script and was validated in accordance with normal parameters.

After a smart contract exchange was started, and the initiating transaction was distributed on-chain, another preset transaction would be modified and sent to redeem the funds and close the instance. The redeeming transaction would specifically select the unspent-output sent by the initiator as an input, and apply redeemer arguments, the full datum, and datum hash. In our test instance, the datum was resolved by decrypting the datum field in the initiator transaction. Upon decryption the redeemer transaction provides the full datum for the script for validation.

Alongside baseline validation parameters, contract logic is executed on the redeemer transaction. The validator node computes the datum hash validity of both participants, using their public keys, and the full datum provided by the redeemer. Bob knew the full datum since Alice encrypted it in the on-chain initiator transaction. Assuming datum signatures are valid, ensuring data authenticity, the script unlocks Bob's assets in USD as specified by the redeemer arguments.

7 Evaluation

7.1 Results

This section details our simulation results which have been collected by running methods outlined in Section 6 within the experiment structure introduced in Section 6.2. Several measurement results are collected based on shard failure rate, redistribution latency, transaction throughput, and transaction latency.

We use the following **Definitions** to illustrate the results of the tests outlined in section 6
$s =$number of shards
$n =$total nodes

m =malicious nodes
h =honest nodes
m_r =malicious rate
tps =transactions per second
L =transaction latency (seconds)
R_L =redistribution latency (seconds)
B_s = block-size
T_{pool} =transaction pool
E_i =specific epoch

Shard Failure Rates We define shard failure as performance that compromises the integrity of the global network, resulting in the authorisation of illegitimate transactions. We expect to see the overall Byzantine fault tolerance of the network decrease linearly to shard scale-out and malicious adversary growth. In ambitious shard deployments, we predict growth in malicious nodes will have an increasingly detrimental effect causing observable shard failure. We discuss peak concentrations and outliers, while we provide figures illustrating average distributions across 10 epochs where necessary.

4 Shards: The distributions of nodes under $s = 4$ were well beneath the consensus threshold in all shards across each epoch. The peak concentration was observed in s_1 (E_6) under $m_r = 0.4$, at $h = 77$ and $m = 67$ with m comprising 46.5% of the shard participants.

8 Shards: We observe the distribution of $s = 8$ to be within a safe threshold across all epochs although peaks were found at $m_r = 0.4$ in s_5 (E_4) where m comprised 44.7% of the shard. The average distributions at $m_r = 0.4$ are illustrated in Fig 6.

12 Shards: Distributions under $s = 12$ became more concentrated around $m_r = 0.3$ and $m_r = 0.4$. High malicious concentrations in $m_r = 0.3$ were observed in $s_1 1$ (E_1) and $s_1 1$ (E_6) where m was 9.5% higher than m_r comprising 39.5% of the shard (see Fig 7).

In $m_r = 0.4$ we started to see increased frequency in concentrated distributions per epoch as the average baseline was now observed to sit at similar levels as the peak outliers in $m_r = 0.3$ (see Fig 8). Peak concentrations were recorded where m comprised 47.9% of a shard across 6 epochs.

16 Shards: Disproportionate malicious concentration was observed earlier than previous deployments, at $m_r = 0.2$. We find that concentration was 10.5% higher than m_r in s_2 (E_1) and s_7 (E_6) at 30.5%, peaking once at $h = 23$ and $m = 13$ in s_{14} (E_1) with m comprising 36.1% of the shard participants (see Fig 9).

In our simulation of $m_r = 0.3$ we observe m concentration 14.4% higher than m_r in s_6 (E_1), s_3 (E_5) and s_{11} (E_6) at 44.4%, and peak concentrations at $h = 19$ and $m = 17$ in , s_3 (E_6) at 47.2% (see Fig 10).

Shard failure, where $m \leq \frac{1}{2}(n)$, was observed on 14 occasions in the deployments with $m_r = 0.4$ across 9 epochs, peaking at 58.8% m concentration in , s_2

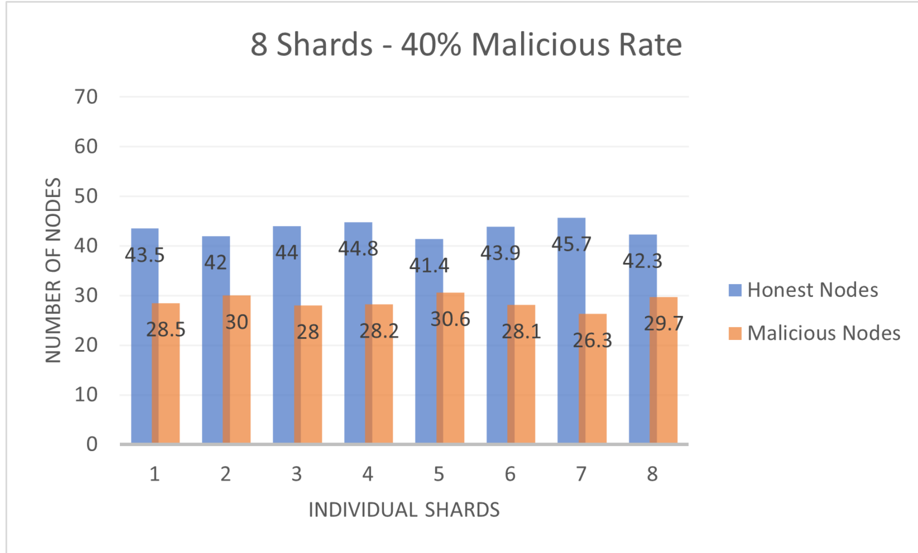

Fig. 6. Average Distribution: 8 Shards, 40 Malicious Rate

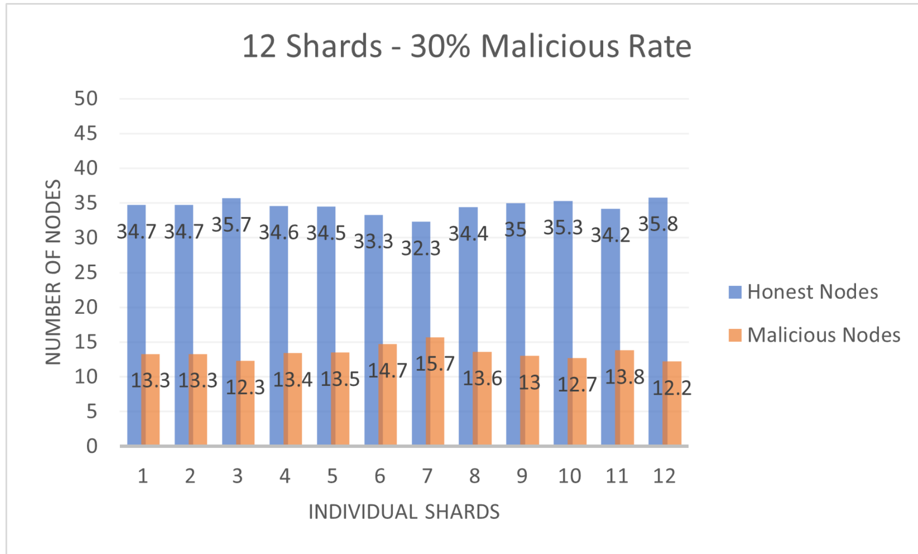

Fig. 7. Average Distribution: 12 Shards, 30% Malicious Rate

(E_1),s_3 (E_3) and s_1 (E_{10}). An equal distribution, where $m = \frac{1}{2}(n)$ was observed on 10 occasions across 7 epochs (see Fig 11).

32 Shards: No specific malicious concentration was recorded at $m_r = 0.1$ although from $m_r = 0.2$ onwards outlier distributions represented a dispropor-

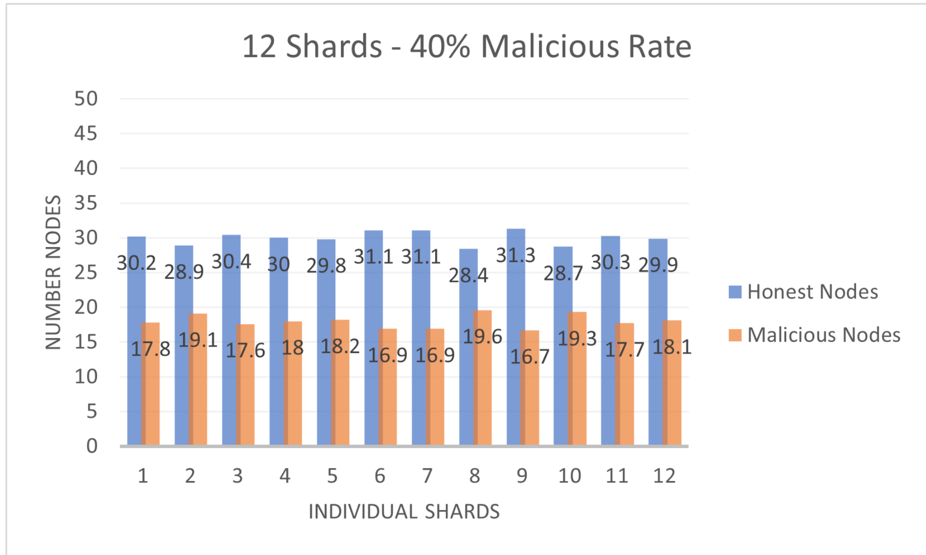

Fig. 8. Average Distribution: 12 Shards, 40% Malicious Rate

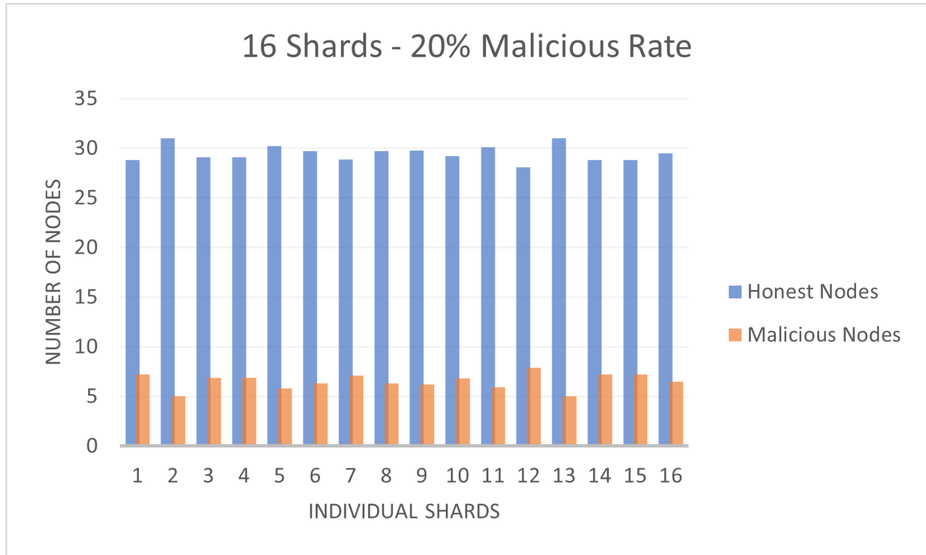

Fig. 9. Average Distribution: 16 Shards, 20% Malicious Rate

tionate percentage of individual shards. In our deployment at $m_r = 0.2$ we observe peak m concentration to be 22.4% higher than m_r in $s_1 7$ (E_1), s_{26} (E_4) and s_7 (E_8) at 44.4% (see Fig 12).

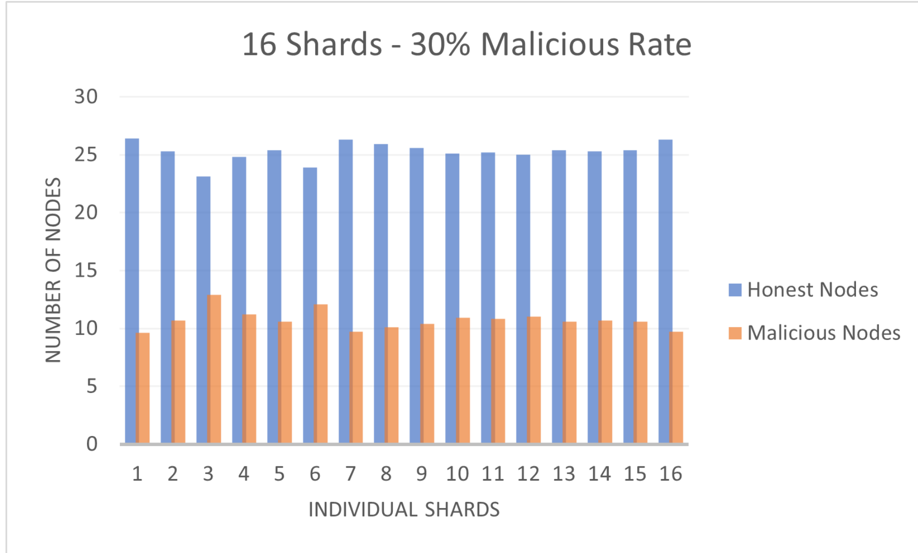

Fig. 10. Average Distribution: 16 Shards, 30% Malicious Rate

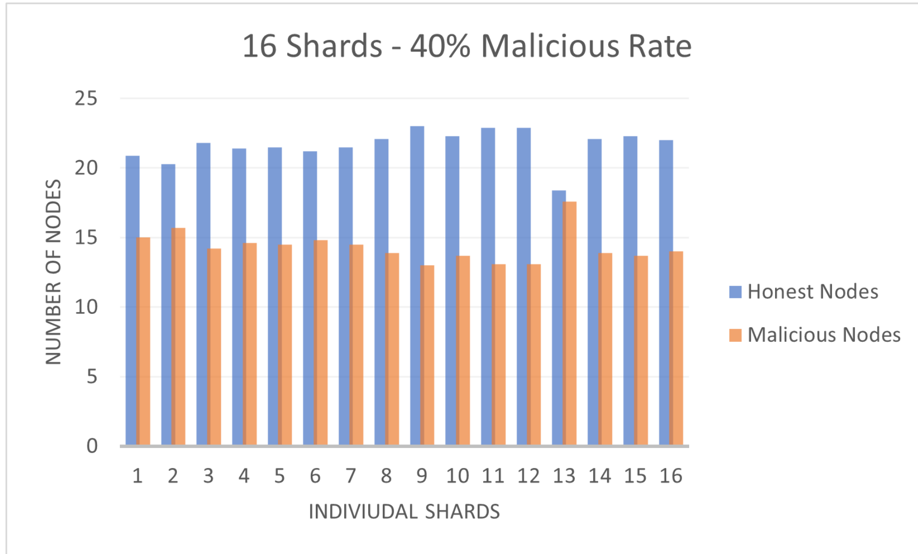

Fig. 11. Average Distribution: 16 Shards, 40% Malicious Rate

At $m_r = 0.3$ we recorded 7 instances of shard failure across 4 epochs. This resulted in 25.5% higher concentration than m_r with m representing 55.5% of s_{10}, s_{12}, s_{32} in E_8 and s_1, s_2, s_3 E_9, peaking at 61.1% concentration in s_{19} in E_7 (see Fig 13).

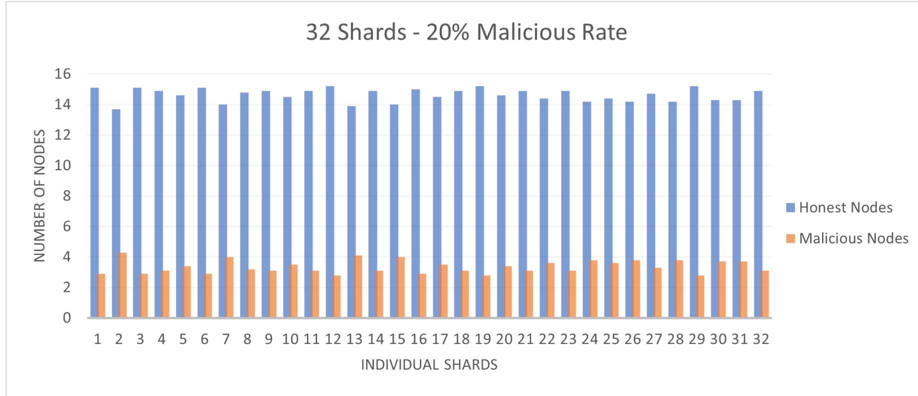

Fig. 12. Average Distribution: 32 Shards, 20% Malicious Rate

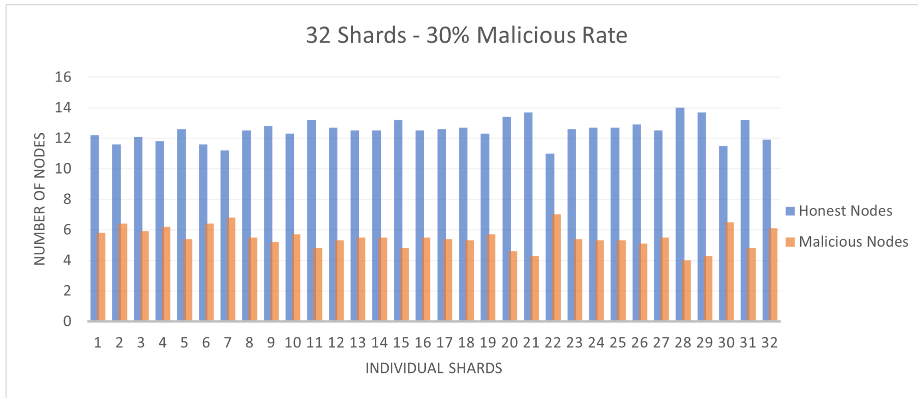

Fig. 13. Average Distribution: 32 Shards, 30% Malicious Rate

At $m_r = 0.4$ we observe 45 accounts of shard failure across all epochs. Peaking at 72.2% m concentration (32.2% higher than m_r) in s_{15} (E_1), s_{30} (E_6) and s_{25} (E_7) (see Fig 14).

Peak Concentrations Fig 15 illustrates the peak concentrations of m across all s deployments and m_r variations.

The following table (Table 3) details our observations of peak concentrations across each shard deployment.

8 Conclusion

Through the research conducted in this study we identified a lack of layer one scaling solutions being implemented on extended-UTXO state modelled blockchains. We proposed S-EUTO, a novel proof of concept sharding protocol, which incorporated an extended-UTXO state ledger, bias-resistant distributed

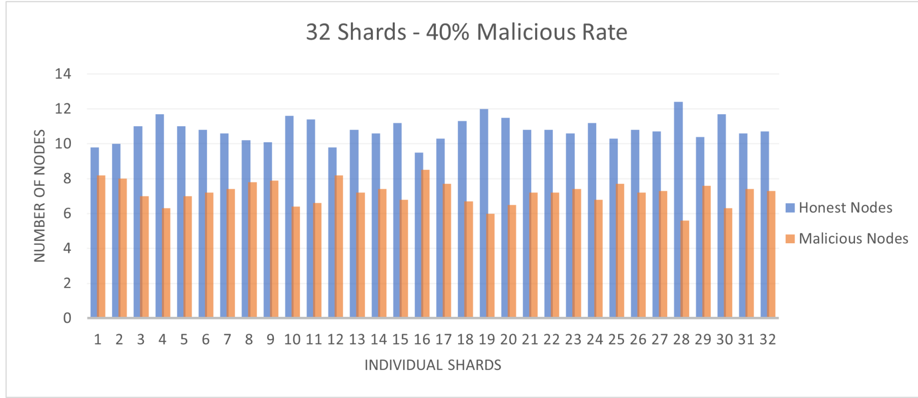

Fig. 14. Average Distribution: 32 Shards, 40% Malicious Rate

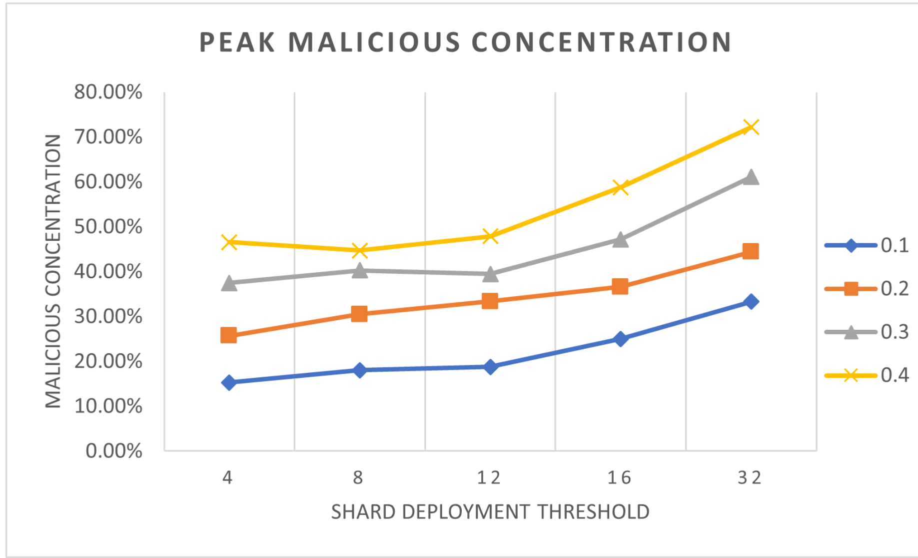

Fig. 15. Peak Malicious Concentrations

Table 3. Peak concentrations across each shard deployment where m_r represents the variations, and s represents the number of shards

m_r	$s = 4$	$s = 8$	$s = 12$	$s = 16$	$s = 32$
0.1	15.27%	18.05%	18.75%	25%	33.33%
0.2	25.69%	30.5%	33.33%	36.6%	44.4%
0.3	37.5%	40.27%	39.5%	47.2%	61.1%
0.4	46.6%	44.7%	47.9%	58.8%	72.2%

randomness function, smart contract framework, and sharded network to achieve transaction parallelism. Despite slight sacrifices regarding the size of malicious adversaries the network can withstand, our findings illustrated a significant increase to scalability brought about when sharding is implemented on an eUTXO blockchain. This was accomplished without sacrificing state properties such as traceability, anonymity and ownership, while data immutability was maintained. Shard allocation involved aspects of centralisation, however the impact was managed by distributing elements of centralised trust equally throughout the network over various iterations, thereby minimising the impact of central authorities to a purely semantic observation. Security evaluation determined that S-EUTO is resistant to the mlacious nodes distribution across Shards based Eutxo model. Certain limitations were outlined in that additional complexities associated with tracking state validity may cause higher computational overheads, although this was ultimately concluded as comparably insignificant.

References

1. S. Nakamoto, "Bitcoin whitepaper.," 2008.
2. M. F. Sallal, *Evaluation of Security and Performance of Clustering in the Bitcoin Network, with the Aim of Improving the Consistency of the Blockchain.* PhD thesis, University of Portsmouth, 2018.
3. J. Golosova and A. Romanovs, "The advantages and disadvantages of the blockchain technology.," *In 2018 IEEE 6th workshop on advances in information, electronic and electrical engineering (AIEEE)*, pp. 1–6, 2018.
4. M. Sallal, S. Schneider, M. Casey, C. Dragan, F. Dupressoir, L. Riley, H. Treharne, J. Wadsworth, and P. Wright, "Vmv: augmenting an internet voting system with selene verifiability," *arXiv e-prints*, pp. arXiv–1912, 2019.
5. M. Sallal, G. Owenson, and M. Adda, "Security and performance evaluation of master node protocol in the bitcoin peer-to-peer network," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–6, IEEE, 2020.
6. J. Göbel and A. Krzesinski, "Increased block size and Bitcoin blockchain dynamics.," *In 2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–6, 2017.
7. C. Worley and A. Skjellum, "Blockchain tradeoffs and challenges for current and emerging applications: generalization, fragmentation, sidechains, and scalability.," *In 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. pp. 1582–1587, 2018.
8. P.-S. C. N.-A. G. Delgado-Segura, S. and J. Herrera-Joancomartí, "Analysis of the bitcoin UTXO set.," *In International Conference on Financial Cryptography and Data Security*, pp. 78–91, 2018.
9. M. Sallal, R. de Fréin, A. Malik, and B. Aziz, "An empirical comparison of the security and performance characteristics of topology formation algorithms for bitcoin networks," *Array*, vol. 15, p. 100221, 2022.
10. C. J.-M. K. M.-O. P. J. M. Chakravarty, M.M. and P. Wadler, "The extended UTXO model. In International Conference on Financial Cryptography and Data Security," pp. 525–539, 2020.

11. J. Golosova and A. Romanovs, "BlockMaze: An efficient privacy-preserving account-model blockchain based on zk-SNARKs.," *IEEE Transactions on Dependable and Secure Computing.*, 2020.
12. V. E.-I. I. T.-R. M. E. Tikhomirov, S. and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts.," *In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pp. pp. 9–16, 2019.
13. M. Sallal, S. Schneider, M. Casey, F. Dupressoir, H. Treharne, C. Dragan, L. Riley, and P. Wright, "Augmenting an internet voting system with selene verifiability using permissioned distributed ledger," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1167–1168, IEEE, 2020.
14. M. M. Zamani, M. and M. Raykova, "Rapidchain: Scaling blockchain via full sharding.," *In Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 931–948, 2018.
15. J.-P. K. G.-L. G. N. S. E. Kokoris-Kogias, E. and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding.," *In 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 583–598, 2018.
16. N. V.-Z. C. B.-K. G. S. Luu, L. and P. Saxena, "A secure sharding protocol for open blockchains.," *In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 17–30, 2016.
17. Q. Nguyen, "Implementing OmniLedger sharding.," 2008.
18. M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 931–948, 2018.
19. K.-K.-E. Avarikioti, G. and R. Wattenhofer, "Divide and scale: Formalization of distributed ledger sharding protocols.," 2019.
20. M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," *arXiv preprint arXiv:1708.03778*, 2017.
21. S.-A. B. S.-H. D. Al-Bassam, M. and G. Danezis, "Chainspace: A sharded smart contracts platform.," 2017.
22. Y. J.-L. H. C.-S. Han, R. and P. Esteves-Veríssimo, "On the security and performance of blockchain sharding.," *Cryptology ePrint Archive.*, 2021.
23. J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones.," *In 16th USENIX symposium on networked systems design and implementation (NSDI 19)*, pp. pp. 95–112, 2019.
24. N.-J. M. B.-A. A. T. E. Schwarz-Schilling, C. and D. Tse, "The Zilliqa Project: A Secure, Scalable Blockchain Platform.," 2017.
25. A. Skidanov, "Limitations of Zilliqa's sharding approach", Near Protocol.," 2018.
26. M. A. Pomogalova, A.V. and K. Yesalov, "Key Features and Formation of Transactions in the Case of Using UTXO, EUTXO and Account Based Data Storage Models.," *In 2022 International Conference on Modern Network Technologies (MoNeTec)*, pp. pp. 1–7, 2022.
27. D. Ergo., "Ergo: A resilient platform for contractual money," 2019.
28. A. Chepurnoy and A. Saxena, "On Contractual Money," 2019.
29. A. Slesarenko, "ErgoTree Specification for Ergo Protocol 1.0.," 2020.
30. J. Xie, "Cell Model, A generalized UTXO as state storage.," 2019.
31. M. A.-K. J. Yang, I., "The Nervos Network Positioning Paper.," *IEEE Network*, pp. pp.166–173, 2019.
32. S. Nielson and C. Monson, "Asymmetric Encryption: Public/Private Keys.," *In Practical Cryptography in Python*, pp. 111–163, 2019.

33. M. Sallal, R. de Fréin, and A. Malik, "Pvpbc: Privacy-and verifiability-preserving e-voting based on permissioned blockchain," *Future Internet*, vol. 15, no. 4, p. 121, 2023.
34. M. Sallal, G. Owenson, and M. Adda, "Bitcoin network measurements for simulation validation and parametrisation," in *11th International Network Conference*, 2016.
35. G. Owenson, M. Adda, *et al.*, "Proximity awareness approach to enhance propagation delay on the bitcoin peer-to-peer network," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2411–2416, IEEE, 2017.
36. M. Fadhil, G. Owenson, and M. Adda, "A bitcoin model for evaluation of clustering to improve propagation delay in bitcoin network," in *2016 IEEE intl conference on computational science and engineering (CSE) and IEEE intl conference on embedded and ubiquitous computing (EUC) and 15th intl symposium on distributed computing and applications for business engineering (DCABES)*, pp. 468–475, IEEE, 2016.
37. A. M.-L. J. Sloderbeck, M. and M. Steurer, "High-speed digital interface for a real-time digital simulator.," *In Proceedings of the 2010 Conference on Grand Challenges in Modeling Simulation*, pp. pp. 399–405, 2010.
38. M. Fadhil, G. Owenson, and M. Adda, "Locality based approach to improve propagation delay on the bitcoin peer-to-peer network," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 556–559, IEEE, 2017.