Calyx: Privacy-Preserving Multi-Token Optimistic-Rollup Protocol

Dominik Apel
Common Prefix

Zeta Avarikioti
TU Wien & Common Prefix

Matteo Maffei
TU Wien

Yuheng Wang
TU Wien

Abstract

Rollup protocols have recently received significant attention as a promising class of Layer 2 (L2) scalability solutions. By utilizing the Layer 1 (L1) blockchain solely as a bulletin board for a summary of the executed transactions and state changes, rollups enable secure off-chain execution while avoiding the complexity of other L2 mechanisms. However, to ensure data availability, current rollup protocols require the plaintext of executed transactions to be published on-chain, resulting in inherent privacy limitations.

In this paper, we address this problem by introducing Calyx, the first privacy-preserving multi-token optimistic-Rollup protocol. Calyx guarantees full payment privacy for all L2 transactions, revealing no information about the sender, recipient, transferred amount, or token type. The protocol further supports atomic execution of multiple multi-token transactions and introduces a transaction fee scheme to enable broader application scenarios while ensuring the sustainable operation of the protocol. To enforce correctness, Calyx adopts an efficient one-step fraud-proof mechanism. We analyze the security and privacy guarantees of the protocol and provide an implementation and evaluation. Our results show that executing a single transaction costs approximately \$0.06 (0.00002 ETH) and incurs only constant-size on-chain cost in asymptotic terms.

1 Introduction

Due to the decentralized and security-preserving properties, blockchain platforms such as Ethereum have attracted significant attention in recent years as a foundation for digital transactions and payments. As of the end of April 2025, Ether (ETH), the native token of Ethereum, had a market capitalization of approximately 211 billion \$ [1]. Beyond native token transfers, Ethereum also supports the issuance and trading of non-native tokens conforming to the ERC-20 standard [34]. These tokens are typically created to serve the needs of decentralized applications (dApps) built on Ethereum and are widely used in decentralized finance (DeFi) and decentralized autonomous organizations (DAOs) within the Ethereum ecosystem.

To ensure transaction integrity and verifiability, most blockchain systems, such as Ethereum, require all transactions to be published in plaintext, allowing all participants to maintain a consistent view of the blockchain. However, this requirement introduces two well-known challenges that hinder broader adoption: privacy and scalability.

Privacy challenge. Privacy is a critical requirement in the blockchain field. Without it, users in a permissionless network can be easily identified and become vulnerable to censorship attacks [35], front-running attacks [31], etc. In Ethereum's account-based system, publishing plaintext transactions exposes sender, recipient, and value, enabling linkability and attacks such as censorship or front-running [32]. Privacy-preserving cryptocurrencies such as Zcash [14] and Monero [5] employ cryptographic techniques to mitigate linkability; however, these systems do not support non-native tokens as Ethereum does. Moreover, their scalability performance does not significantly differ from that of Ethereum.

Scalability challenge. Specifically, Ethereum achieves a throughput of approximately 15–30 transactions per second (TPS) [6], whereas Visa can process up to 60,000 TPS [33]. L2 protocols have emerged as a promising direction to address this scalability limitation by offloading transaction execution from the main chain and relying on efficient on-chain mechanisms to verify execution correctness [13]. Among these, Rollup protocols have received substantial attention. In a Rollup, the execution is performed solely by a designated operator (or sequencer), and only the state transition is published to the L1 blockchain [18]. However, most Rollup designs also require the executed transactions to be posted on-chain to realize *data-availability*, thereby offering limited or no privacy guarantees.

Given the limitations of existing solutions, a natural question arises: *Is it possible to design a privacy-preserving and scalable payment scheme that supports multi-token transactions using Rollup protocols?*

1.1 Related work

Achieving privacy while preserving the security guarantees of blockchain systems is a central research challenge and an active area of development. The mainstream approach relies on cryptographic techniques that enable the verification of transaction execution without revealing sensitive information. For example, Zerocoin [21] and Zerocash [8] use zero-knowledge proof systems to validate transactions while hiding private information. Monero [5] uses ring signatures [22] to protect linkability. However, these systems are tailored for their native token. In Ethereum, protocols such as ZETH [7] uses a transaction mixing smart contract to protect linkability; Tornado Cash [4] and Zether [10] employ transaction encryption and zero-knowledge proofs to enable privacy-preserving transfers. While these solutions enhance transaction privacy, they operate entirely at the L1 level; thus, the inherent throughput limitations remain.

Based on the verification mechanism, Rollup protocols are typically categorized into two types: zk-Rollups, such as zkSync [16], and optimistic-Rollups, such as Arbitrum [18] and Optimism [25]. As noted in prior work [28, 30], the zk-Rollups leverage zk-SNARK proof mainly to realize efficient and secure state transitions [26, 29], and the executed transaction data must still be published on the L1 blockchain to ensure *data availability*. Consequently, privacy guarantees in these systems remain limited in practice. Recently, zk-SNARK aggregation schemes designed for blockchain privacy transactions like [28] have begun to be adopted by novel zk-Rollup protocols such as Aztec [2], aiming to both protect transaction privacy and reduce the on-chain verification burden. However, without special techniques such as a preprocessing procedure, the computation loads still grow with the amount of aggregated proofs [19]. In addition to the on-chain computation burden, the potential security vulnerabilities introduced by deploying new zk-SNARK schemes on blockchains further limit the broader adoption of zk-Rollup–based solutions [24].

In contrast, optimistic-Rollups require on-chain execution by the smart contract only when a fraud-proof and verification procedure is triggered under a pessimistic scenario [18, 25]. The on-chain gas cost is therefore significantly lower than that of zk-Rollups, especially after the introduction of EIP-4844, which enables the use of BLOBs for low-cost data storage [11]. However, to the best of our knowledge, there has been no discussion on achieving privacy in the Optimistic-Rollup setting.

1.2 Contribution

In this paper, we address the above problems by introducing Calyx, the first privacy-preserving optimistic-Rollup protocol that also supports multi-token transactions. Calyx addresses scalability limitations by delegating execution to the L2 Rollup operator, following the Rollup paradigm. Transactions, each transferring a single ERC-20 token type, are submitted to the operator for off-chain execution, and the operator periodically posts succinct protocol state updates to the L1 blockchain as checkpoints.

Calyx employs zk-SNARKs to ensure L2 transaction privacy, revealing no information about the sender, receiver, transferred amount, or token type. While this approach is conceptually simple, designing a practical protocol introduces three key challenges. First, solutions like Zerocash are tailored to L1 transactions with its native token on a different underlying blockchain; adapting it to the Ethereum-based L2 protocol and supporting multi-token requires careful redesign. Second, ensuring that the operator behaves honestly when publishing privacy-preserving transaction execution results on-chain, while keeping associated costs minimal, is nontrivial. Third, given the computation and storage overhead of on-chain smart contracts—even when using BLOBs for lower-cost data storage—it is necessary to design both the data structure and the interaction mechanism between BLOB data and the smart contract with efficiency in mind.

To address the first challenge, Calyx adopts a UTXO-based model for L2 transactions, embedding a specific token type and transaction fee in each transaction. The operator verifies transaction validity using zk-SNARKs, ensuring correctness without revealing sensitive information—such as the coin owner—while guaranteeing that all transactions are valid and consistent. Given that the L2 construction must interact with the L1, Calyx also defines special minting and burning transactions to handle protocol joining and leaving, respectively.

For the second challenge, Calyx follows the optimistic-Rollup model, allowing verifiers to challenge the operator if misbehavior is detected in the on-chain published data. This ensures that large gas costs—such as those for zk-proof verification—are incurred only in pessimistic cases. Moreover, Calyx introduces an efficient one-step fraud-proof mechanism that enumerates in advance all types of misbehavior that could compromise system security, thereby eliminating the need for multi-round interactions and reducing gas costs.

As for the third challenge, in addition to the conventional optimistic-Rollup approach of publishing full transaction batch data on-chain (including L2 transactions, zk-proofs, etc.) in a BLOB, Calyx also maintains two Merkle trees for efficient state checkpointing and verification: the **Coin tree** recording all coins that have ever existed and the **Nullifier tree** recording all nullified (spent) coins. It further employs a *pointer scheme* for the whole BLOB data structure, enabling the smart contract to efficiently access the relevant portions of BLOB data using the pointer provided by the verifier during the challenge period.

Conclusively, the contribution of this paper can be summarized as follows:

- We propose Calyx, the first privacy-preserving multi-token optimistic-Rollup protocol. Calyx adopts a UTXO-based transaction model and employs zk-SNARKs to enable privacy-preserving L2 transactions. To support efficient and privacy-preserving checkpointing and verification on L1 blockchain, we introduce *pointer scheme* for BLOB data and a one-step fraud-proof mechanism based on such design (Section 5).

- We formalize the security and privacy properties that Calyx aims to achieve, and analyze its guarantees under both the honest-but-curious/Byzantine model and the rational adversary model (Section 6).

- We implement a proof-of-concept prototype of Calyx on the Ethereum blockchain and evaluate the associated costs, including contract deployment and transaction publishment. An asymptotic comparison with the other two Rollup protocols, Aztec and Arbitrum, is also conducted. Our evaluation shows that Calyx can be deployed efficiently, with gas costs that make it a practical and feasible solution (Section 7).

2 Preliminaries

2.1 Optimistic-Rollup

An optimistic-Rollup protocol enables a group of clients to conduct off-chain transactions after joining the protocol. To join the protocol, each client must deposit a specified amount of tokens into the Rollup's smart contract deployed on the L1 blockchain. Once the deposit transaction is confirmed on-chain, a corresponding minting transaction that initializes the client's state within the Rollup is considered valid.

After a client successfully joins the protocol, it can initiate transactions with other participants. Unlike payment channel networks such as the Lightning Network [27], where participants must exchange a large number of signatures and messages, clients in an optimistic-Rollup protocol simply submit valid transactions to the operator and wait for execution. The operator aggregates the received transactions into batches and periodically publishes a checkpoint on the L1 blockchain. Each checkpoint represents the current state of the protocol, such as the coin balances of all clients. To ensure correctness, the optimistic-Rollup protocol leverages external verifiers to validate the operator's published data. If a verifier detects misbehavior from the operator—i.e., an incorrect state update—they may submit a fraud-proof. Upon successful verification of the fraud-proof, the operator is penalized, and the invalid state update is rejected. After completing all intended off-chain transactions, a client can initiate the process of leaving the protocol by submitting a leave request to the operator. The operator then executes a burning transaction that reflects the client's final state and publishes this transaction on the L1 blockchain. Once the burning transaction is confirmed on-chain, the client can withdraw its remaining funds from the Rollup smart contract, thereby completing the exit process.

2.2 zk-SNARK

A zk-SNARK (zero-knowledge succinct non-interactive argument of knowledge) [9] proof system is a **proof system** for a relation R, meaning that a Prover P can convince a Verifier V that an instance/witness pair (x,w) belongs to the relation R. As a proof system, it satisfies the following two fundamental security properties:

Completeness: For any $(x,w) \in R$, the honest Prover P can convince the Verifier V that $(x,w) \in R$ with a proof π.

Computational soundness: It is computationally infeasible for any malicious Prover P to convince the Verifier V that some $(x',w') \notin R$ is in R.

In addition to these, zk-SNARKs further provide the following properties:

Non-interactivity: The proof is generated and sent from P to V without any interaction beyond this one message.

Succinctness: The proof π is of constant size and can be verified in $O(1)$ time, independent of the complexity of the relation R.

Zero knowledge: The proof leaks no information about the witness w; the Verifier learns only that $(x,w) \in R$ for some valid w.

2.3 Merkle Tree

A Merkle tree [20] uses a collision-resistant hash function h to build a data structure that supports efficient data membership proofs. It is a binary tree where data elements are stored in the leaves. For any internal node n, let v_l and v_r be the values stored in the left and right children of n, respectively. The value stored in n is then defined as $h(v_l||v_r)$. As a result, the root of the tree acts as a commitment to all data stored in the tree. Moreover, it is possible to prove in logarithmic time that a specific leaf l_i in a tree with root r contains a given data item d. In our construction, we adopt a variant called an *append-only Merkle tree*, defined as follows. Initially, all leaf values are set to 0. The hash function h is modified to a new function h^* as follows:

$$h^*(x) = \begin{cases} 0 & \text{if } x = 0||0 \\ h(x) & \text{otherwise} \end{cases}$$

The function h^* remains collision-resistant if h is collision-resistant. This structure can be used to append and locate newly generated coins without requiring storage of the hash values of all zero leaves.

3 Model

3.1 System Model and Assumptions

There are three types of participants in Calyx: (1) Clients, representing the users of the protocol; (2) $|V| \geq 1$ verifiers, who continuously monitor the blockchain to validate the operator's published state and detect any misbehavior; and (3) The operator, who verifies and publishes client transaction batches to the L1 blockchain. Calyx allows multiple operators and each operator is required to deposit more than $\frac{TVL}{|V|} + c_f$ stake on-chain, where TVL is the total client locked value in Calyx and c_f is the gas cost for publishing a fraud-proof transaction.

We assume that all participants are computationally bounded and that standard cryptographic primitives—including secure communication channels, digital signatures, cryptographic hash functions, encryption schemes, and zk-SNARKs—are correctly instantiated and secure. Communication between clients and the operator is assumed to be asynchronous, which means the message will eventually be delivered. All participants are guaranteed uncensored access to the L1 blockchain, with a known upper bound on the delay for both reading from and writing to the idealized secure blockchain that realizes *consistency* (*safety*) and *liveness* [12].

3.2 Threat Models

We analyze the security of Calyx under two threat models: the traditional honest-but-curious/Byzantine model and the rational model.

Byzantine security model. In this model, participants are classified into two types. Honest participants follow the protocol correctly but are honest-but-curious, meaning they may attempt to infer private information about others by analyzing any public data leaked during protocol execution. Byzantine participants may behave arbitrarily and deviate from the protocol in an adversarial manner, while also attempting to reconstruct private information held by honest parties. We assume the presence of at least one honest operator and one honest verifier who monitors protocol execution and responds in a timely manner.

Rational security model. We further analyze the security of Calyx under a rational adversary model, where all participants are assumed to be rational—that is, they deviate from the protocol, such as by misbehaving to steal coins or attempting to violate privacy, only if doing so yields greater benefits. Here we don't consider the external budget, as proving economic security in the presence of infinite shorting markets (e.g., Bitcoin or Ethereum) is infeasible.

3.3 Protocol Goals

Under the Byzantine security model, we first consider *balance security* in order to capture the fundamental requirement that any honest participant must not lose any tokens they are entitled to after leaving the protocol, regardless of the behavior of other parties. We define this property formally as follows:

Definition 1 (Balance security)**.** Any participant honestly following Calyx does not lose coins.

A protocol that does not execute any transactions can trivially satisfy *balance security*; however, such a protocol is functionally meaningless. To ensure that the protocol makes progress in response to valid requests from honest participants, we define the notion of *liveness* as follows:

Definition 2 (Liveness)**.** Any valid request from participants honestly following Calyx will eventually either be committed on-chain or invalidated.

In addition to *balance security* and *liveness*, which ensure that the protocol functions correctly in response to actions taken by honest participants, we define the privacy property that Calyx aims to achieve as *L2 transaction privacy*, meaning that only indistinguishable public leakage about L2 execution is observable through the published data on L1 blockchain. Our *L2 transaction privacy* property is defined based on the Layer 2 Indistinguishability (L2-IND) game inspired by [14]. Here we give a general idea of the L2-IND game design. The formal definition of the game and how the oracle works can be found in Appendix A.3.

The L2-IND game is defined as a round-based game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, where $\mathcal{A}$ attempts to distinguish which L2 protocol it is interacting with, based on the public information it can access. During the game, $\mathcal{C}$ can issue multiple types of requests representing all possible interactions in Calyx: $CommitBatch$ (generate a batch), $WriteOnL1$ (publish on the L1 blockchain), $CreateAddress$ (generate a client account), $PegIn_{L1}$ (peg in on the L1 blockchain), $Mint$ (create an initial L2 state), $Burn$ (invalidate an L2 state), $PegOut_{L1}$ (peg out on the L1 blockchain), $Transfer$ (propose a transaction), $ReadOnL1, ReadOnL2$ (check the L1 and L2 transactions and states), and finally $guess$, indicating $\mathcal{A}$'s guess of which protocol it is interacting with.

The two requests from $\mathcal{C}$ are required to guarantee *public consistency*, meaning that the public information related to the requests—such as size, public address, and total transaction value—must be identical. $\mathcal{C}$ answers $\mathcal{A}$ using five types of oracles: the L1 Ledger Oracle L_1, the L2 Ledger Oracle L_2, the Client Oracle $\mathcal{H}$, the Operator Oracle $\mathcal{S}$, and the Verifier Oracle $\mathcal{V}$. At each round, except for the final *guess* request, $\mathcal{A}$ submits two requests of the same type, denoted Q_0 and Q_1. For request types $CommitBatch$, $PegIn$, $PegOut$ and $WriteOnL1$, both requests are handled by oracles $\mathcal{V}$ and L_1 with the same sequence number. For other request types related to L2 execution, $\mathcal{C}$ then samples a random bit b, forwards Q_0 to the b-th oracle, and Q_1 to the $(1-b)$-th oracle, thereby shuffling the mapping to challenge whether $\mathcal{A}$ can distinguish the responses. Eventually, $\mathcal{A}$ ends the game with a *guess* request containing $b_{\mathcal{A}}$. $\mathcal{A}$ is considered to win the game if $b_{\mathcal{A}} = b$ with non-negligible probability greater than $\frac{1}{2}$.

Based on the L2-IND game defined above, we can further defined the *L2 transaction privacy* as follows:

Definition 3 (L2 transaction privacy)**.** A rollup protocol realizes *L2 transaction privacy* if for any polynomial-time (PPT) adversary $\mathcal{A}$, the probability of winning the L2-IND game is only negligibly greater than 1/2.

Under the rational security model, we define the following overarching security objective, which encompasses all previously stated goals:

Definition 4 (Rational security)**.** All rational participants will follow the protocol honestly and will not attempt to compromise privacy.

4 Protocol Overview

Before presenting the detailed design of Calyx, we first provide a high-level overview of the protocol. Figure 1 illustrates the structure of Calyx. We then describe the iterative development process, starting from a baseline construction of a standard optimistic-Rollup protocol.

Naive design. In a standard optimistic-Rollup, clients join by depositing into a smart contract and then propose account-based transactions of the form "*A* sends *x* coins to *B*." These transactions are submitted to an operator, published on the L1 blockchain, verified by verifiers, and finally committed. Clients can later exit the protocol with their latest state.

While this approach achieves *balance security* and *liveness*, it fails to provide any privacy guarantees, as transaction details are leaked through the on-chain published data required for verification.

L2 with UTXO model. Since publishing transactions on the L1 blockchain is essential for ensuring the security and verifiability of the optimistic-Rollup protocol, the main challenge is to minimize the information leakage from these published transactions. The UTXO model discloses only the linkability between inputs and outputs. Therefore, Calyx adopts the UTXO model for transactions within the L2 system. Furthermore, each transaction includes a token type parameter and a transaction fee, enabling Calyx to support multi-token transactions while ensuring the sustainable operation of the protocol. Finally, a corresponding mint and burn UTXO transaction is designed to interact with the L1 blockchain for client joining and leaving.

However, the validity of UTXO transaction is usually achieved with digital signatures. Nonetheless, such signatures still reveal linkability through the associated public keys used for verification, thereby leaking metadata that can be exploited by adversaries.

Zero-Knowledge verification. To protect linkability while ensuring that only the coin owner can spend it, we first hide the source of the input coin using a commitment, and then adopt a zk-SNARK proof system to prove transaction validity. The key distinction in applying this idea to an L2 protocol is that, unlike L1 protocols that achieve consensus directly among all participants, the optimistic-Rollup relies on the L1 blockchain as the consensus layer. Consequently, the public inputs required for verifying zk-SNARK proofs must be published on the L1 blockchain by the operator along with the periodic checkpoints.

Considering the high transaction throughput that a Rollup protocol is expected to handle, a new challenge arises: how to efficiently verify the data published by the operator on the L1 blockchain. This includes not only ensuring correctness and availability but also enabling timely verification to support latency-sensitive applications.

Efficient One-step fraud-proof. To address the computational overhead and verification delays introduced by directly combining the optimistic-Rollup framework with a ZK-proof system, this paper proposes an efficient one-step fraud-proof scheme.

Concretely, instead of relying on multiple rounds of interactive verification to locate a faulty execution within the entire published execution history in the BLOB—as is common in standard optimistic-Rollup protocols—our scheme employs a *pointer* mechanism. The fraud-proof transaction includes a pointer that directly identifies the erroneous location in the operator's published BLOB data. This design eliminates the need for the smart contract to conduct extensive on-chain searches over large datasets, thereby reducing computational overhead, gas costs, and overall latency.

5 Protocol Design

In this section, we present the detailed design of the Calyx protocol. The protocol behavior is described from the perspective of a client's life cycle, consisting of three phases: *protocol joining*, *transaction execution*, and *protocol leaving*. We assume that the on-chain smart contract for the protocol has been deployed in advance and is accessible to all participants. The precise structure of the contract and the interaction semantics will be formally defined in conjunction with the explanation of each phase of the protocol.

For a better understanding of our protocol design, we first list the notations used in the protocol construction, as shown in the following table 1.

5.1 Joining Protocol

As a prerequisite for the subsequent procedures, a client can join Calyx in two steps: (1) deposit coins on the L1 blockchain; (2) the initial L2 state corresponding to the client is created.

For the first step, a client sends a denomination *value* of some ERC-20 compliant token type *token* (including the native token ETH) as initial asset and the corresponding public key used for creating signatures in the following procedures pk_{sig} to the on-chain smart contract. The contract will lock these deposit assets, record the corresponding information on-chain, and reply with a fresh nonce value n, which can be used as evidence for creating the L2 initial state in the second

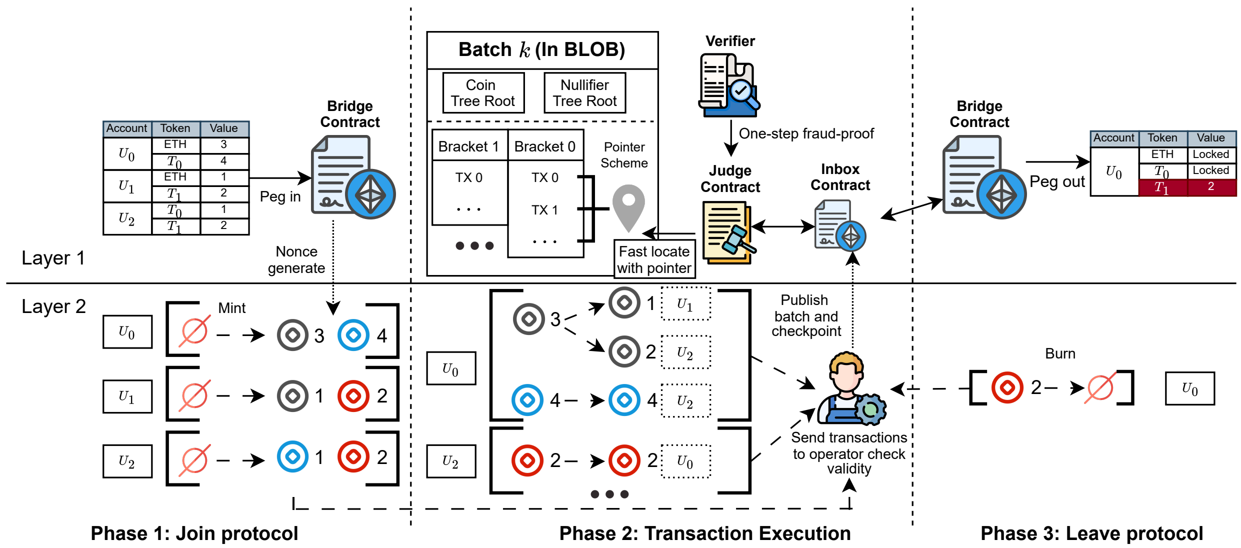

Figure 1: Calyx Overview. Consider three clients U_0, U_1, U_2 joining the protocol with three types of tokens (ETH, T_0, T_1). Then U_0 privately pays U_1 and U_2, while U_2 pays U_0. Transactions grouped in a bracket are executed atomically. Execution occurs after the transactions are published on-chain and no fraud-proof is submitted by a verifier within a specified time window. Finally, U_0 retrieves the 2 tokens T_1 from U_2 through the contract.

Table 1: Notation explanation

Notation	Explanation
h	hash function
Enc	Encryption function
sk_{coin} $pk_{coin} = h(sk_{coin})$	secret key and public key to spend a coin generated with hash function
sk_{Enc}, pk_{Enc}	secret key and public key for encryption
p	random identifier
$token$	token type
$value$	transaction value
sn	coin serial number
crt	coin Merkle tree root
id_{L1}	L1 address
pk_{sig}	public key for signing
fee	L2 transaction fee provided by participants

step. The on-chain smart contract for the joining process is detailed in Algorithm 1. Note that in the following, multiple contract codes are presented. In practice, they can either be deployed separately or integrated with each other.

For the second step, the initial state of Calyx for a client is created by generating and executing a special UTXO transaction called a minting transaction, mint. Since it initializes the off-chain state for the client, the minting transaction has no inputs. Its output consists of a single value $c = h(token||value||fee||k)$, where $k = h(p||pk_{coin})$ is the compressed client identity and p is the randomly generated coin identifier, and pk_{coin} represents the owner of the coin who knows the preimage sk_{coin}. The body of the transaction includes two critical pieces of information for identifying and verifying its validity: the mint nonce n obtained from the previous on-chain deposit step, and the compressed identity k. To ensure *economic sustainability* and guarantee the rational security of the protocol, we introduce a transaction fee scheme: the fee in the coin commitment denotes the L2 transaction fee paid to the operator as compensation for the gas costs of publishing transactions on the L1 blockchain. These fees are included in the collateral deposit beforehand.

Once a valid minting transaction is constructed by the client and executed according to the procedure described later, the client is considered to have successfully joined Calyx. Note that the validity of a mint transaction is not verified using zk-SNARK proof; instead, it is checked directly against the records maintained in the smart contract.

5.2 L2 Transaction Execution

As an L2 protocol, Calyx's primary role is to offload the execution of transactions to the off-chain environment. Here, we describe the transaction execution procedure from the perspective of a regular *transfer transaction*, which also applies to special transactions such as the mint transaction described above.

As a UTXO-model transaction, a transfer transaction in Calyx contains two lists of M inputs and M outputs (with unused elements left empty), all of the same token type. The output of the transaction is the coin commitment $c =$

Algorithm 1: L1 to L2 Bridge (Contract 1)

Initialize nonce value $curNonce = 0$;
Record nonce mapping
  $nonces = \mathbb{N} \rightarrow (\mathbb{N}, \mathcal{T}, \mathcal{V}, \mathcal{G}, \mathcal{P})$
  `// (public key, token type, token value, L2 transaction fee, included block number)`
Function $toL2(v, t, pk_{sig})$ **is**
  $g = msg.value$;
  if $t.allowance(id_{L1}, this) < v$ **then**
    revert
  end
  $t.transferFrom(id_{L1}, this, v)$;
  $curNonce \leftarrow curNonce + 1$;
  $nonces[curNonce] \leftarrow (pk_{sig}, t, v, g, block.number)$;
  return $n := curNonce$;
end

Function $feeToL2(pk_{sig})$ **is**
  $g = msg.value$;
  $curNonce \leftarrow curNonce + 1$;
  $nonces[curNonce] \leftarrow (pk_{sig}, 0, 0, g, block.number)$;
  return $curNonce$;
end

Function $getMintData(n)$ **is**
  if $nonces[n] = \perp$ **then**
    return $\perp$;
  end
  return $nonces[n]$;
end

$h(token||value||fee||k = h(p||pk_{coin}))$ of the following information: (1) **token type** $token$; (2) **coin value** $value$ for the output coin; (3) **fee value** fee, measured in the native currency of the L1 chain; (4) **identifier** p, which is generated randomly; (5) **coin owner public key** pk_{coin}. If the sender and receiver are not able to reach agreement themselves about the private value, all the information can be sent to the receiver through an encrypted message privately with the receiver's encryption key pair $C_{inf} = Enc_{pk^{receiver}_{Enc}}(token, value, fee, p)$ and attached to the coin commitment.

And accordingly each input includes the privacy-preserving information of the coin it uses:

- The **coin tree root** crt is the Merkle tree root that represents all coins ever created during Calyx's lifecycle.
- The unique **serial number** $sn = h(p||sk_{coin})$ is unique for every coin that is only known to the true owner of the coin.
- The **signature public key** pk_{sig} that belongs to the owner of the input coin who knows the corresponding secret key.
- The **coin commitment** $cm = h(token||value||fee||pk_{auth})$ to the coin's private properties, which includes: (1) **Signing Key Authorizer** $pk_{auth} = h(sk_{coin}||pk_{sig})$; (2) **Token Type** $token$; (3) **Coin Value** $value$ used for transaction; (4) **Fee Value** fee.

To further support clients in transferring multiple types of tokens in different UTXO transactions at the same time. We further extend the idea of transaction into transaction brackets. Transactions are put into transaction brackets, acting as an "all-or-nothing" primitive, allowing the atomic execution of multiple transactions. One of the use cases for such an atomic primitive is to swap different types of tokens. Furthermore, a hash of all the transactions' hash values is contained in the bracket to guarantee integrity. Finally, a transaction bracket must contain a signature from every public key contained in an input on the bracket hash, i.e., all coin owners must agree that their coins are spent in this specific way.

By leveraging a commitment scheme, the source of the spent coin is hidden, thereby protecting against linkability. However, this introduces a new challenge: how to ensure transaction correctness while keeping data hidden. Specifically, two correctness properties must be satisfied: *Input correctness* and *Input-output relation correctness*.

5.2.1 zk-SNARK Circuits

In Calyx, we leverage the zk-SNARK proof, which is included in the transfer transaction body, to enforce both properties mentioned above. In the following, we present the corresponding circuits that realize these two correctness goals.

Input proof. Here we further define the *input correctness* should realize the following goals:

1. There exist such outputs that are selected to spend from. This is done by privately providing a Merkle tree proof that asserts that the coin is a leaf in the tree with the provided root. The Merkle tree depth must be a fixed constant, called D.
2. The prover is the owner of the coin and knows the sk_{coin}.
3. The provided serial number was calculated correctly.
4. *The owner approves of the provided signature verification key:* This is done by deriving a pk_{auth} value from both the signature verification key and the secret key of the coin. This calculated value is private in order to achieve the hiding property of the provided coin commitment.
5. *The provided commitment was constructed correctly.*

The inputs for the input proof's zk-SNARK circuit are as follows, with public values underlined, all values of scalar type *field* except otherwise noted, and $T[N]$ denoting an array type containing N elements of type T:

$$(\underline{crt},\ \underline{sn},\ \underline{cm},\ \underline{pk_{sig}},\ pk_{auth},\ c,\ bool[D]\ dir,\ field[D]\ path,\ r,\ token,\ value,\ fee,\ p,\ pk_{coin},\ sk_{coin})$$

The circuit of the zk-SNARK can be described by Algorithm 2:

Algorithm 2: Input Proof (zk-SNARK Circuit 1)

Data: $\underline{crt} \in \mathbb{F}, \underline{sn} \in \mathbb{F}, \underline{cm} \in \mathbb{F}, \underline{pk_{sig}} \in \mathbb{F}, pk_{auth} \in \mathbb{F}, c \in \mathbb{F}, dir \in \mathbb{B}^D, path \in \mathbb{B}^D, token \in \mathbb{F}, value \in \mathbb{F}, fee \in \mathbb{F}, p \in \mathbb{F}, pk_{coin} \in \mathbb{F}, sk_{coin} \in \mathbb{F}$

`// Finite field F, Binary Set B`

assert $c = h(token||value||fee||h(p||pk_{coin}))$;
assert $\underline{sn} = h(p||sk_{coin})$;
assert $pk_{coin} = h(sk_{coin})$;
assert $pk_{auth} = h(sk_{coin}||\underline{pk_{sig}})$;
assert $\underline{cm} = h(token||value||fee||pk_{auth})$;
$tmp \leftarrow c$;
for $i = 1$ **to** $D-1$ **do**
  if $dir[i]$ **then**
    $tmp \leftarrow h(tmp||path[i])$;
  else
    $tmp \leftarrow h(path[i]||tmp)$;
  end
end
assert $tmp = \underline{crt}$;

Transaction proof. We use a transaction proof to define the correct relationship between the transaction outputs and inputs. The transaction proof aims to guarantee the following goals:

1. The prover knows the secret preimage of the commitment values.
2. All the inputs and outputs of a transfer transaction should belong to the same token type. The number of inputs and outputs is fixed to a constant number M.
3. The sum of the input values should match the sum of the output values.
4. In order to use the zk-SNARK scheme, we further require that the input and output are positive.

The inputs are (public values are bold):

$$(field[M]\ \underline{C},\ bool[M]\ \underline{isConnected},\ bool[M]\ \underline{isInput},\ field\ \underline{fee},\ field[M]\ token,\ field[M]\ value,\ bool[M]\ valueDecomp,\ field[M]\ fee,\ bool[M]\ valueDecomp,\ field[M]\ cmr)$$

Here, the two boolean vectors *isConnected* and *isInput* of size M have the following semantics: If $isConnected[k] =$ False, then both the k^{th} input and output are empty. Otherwise, $isInput[k]$ determines whether slot k is connected to an input or an output. This affects the sign of the value and fee of the coin when computing the transaction balance: inputs add value to the balance, while outputs subtract value. The provided inputs and outputs for a transaction must follow the structure specified by *isConnected* and *isInput* for the transaction to be valid. Finally, the circuit checks that the balances sum to zero, ensuring that the total output values match the total input values. The zk-SNARK circuit can be described by Algorithm 3.

5.2.2 Transaction execution with one-step Fraud-proof

After receiving transactions from the client, the operator first verifies the proofs generated by the zk-SNARK scheme based on the circuit defined above. In order to execute these valid transactions, two steps to follow in Calyx: (1) the operator needs to publish the valid transactions it wants to execute to the L1 blockchain, including the BLOB data and the smart contract stored data; (2) the published transactions are only considered to be executed after there exists no fraud-proof for a certain time period.

On-chain published data. Once the operator has completed a batch of transaction brackets pending execution, it first generates a *fee-collecting transaction* to collect all the fees included in the transactions and appends it to the end of the batch, This transaction contains an output with commitment $h(0||0||ck_f||k)$, where ck_f is the transaction fee checkpoint at the beginning of the batch, and k is a random value chosen by the operator and included in the body of the transaction for verification. Then a L1 transaction carrying this batch as BLOB data is then submitted to the Inbox contract on the L1 blockchain.

In addition to the data written to the BLOB, the transaction records metadata in the Inbox contract, including the commitment to the BLOB data (serving as the checkpoint of the L2 state) and information about the coins burned in this batch for clients leaving the protocol. Specifically, an array of elements of the form $(txbid, txid, t, v, g, id_{L1})$ is passed to the Inbox contract. Each element represents a claim that the transaction at index *txid* within the transaction bracket at index *txbid* is a burn transaction that burns v tokens of type t and g units of ETH, while specifying *id* as the receiver address. Algorithm 4 shows the working of the Inbox contract.

Two types of data are recorded in the blockchain BLOB part. First, at the beginning of the batch, the updated root of the **Coin Tree**, which records all coins that have ever existed, and the root of the **Nullifier Tree**, which records all coins that have been spent, are stored as the current L2 state. The Merkle roots of these two trees can then be used to efficiently prove or dispute the validity of a coin. Note that the leaves of

Algorithm 3: Transaction Proof (zk-SNARK Circuit 2)

Data: $\underline{C} \in \mathbb{F}^M, \underline{isConnected} \in \mathbb{B}^M, \underline{isInput} \in \mathbb{B}^M, \underline{fee} \in \mathbb{F}, token \in \mathbb{F}^M, value \in \mathbb{F}^M, valueDecomp \in (\mathbb{B}^l)^M, fee \in \mathbb{F}^M, feeDecomp \in (\mathbb{B}^l)^M, cmr \in \mathbb{F}^M$

```
// Finite field 𝔽, Binary Set 𝔹
```

$bal_v \leftarrow 0;$
$bal_g \leftarrow 0;$
for $i = 0$ **to** $M-1$ **do**
  if $\underline{used}[i]$ **then**
    assert $\underline{C}[i] = h(token[i] \parallel value[i] \parallel fee[i] \parallel cmr[i]);$
    if $\underline{isInput}[i]$ **then**
      `// This is an input`
      $bal_v \leftarrow bal_v + value[i];$
      $bal_g \leftarrow bal_g + fee[i];$
    else
      `// This is an output`
      $bal_v \leftarrow bal_v - value[i];$
      $bal_g \leftarrow bal_g - fee[i];$
      `// Check range`
      $value' \leftarrow 0;$
      $fee' \leftarrow 0;$
      for $j = 0$ **to** $l-1$ **do**
        if $valueDecomp[i][j]$ **then**
          $value' \leftarrow value' + 2^j;$
        end
        if $feeDecomp[i][j]$ **then**
          $fee' \leftarrow fee' + 2^j;$
        end
      end
      assert $value' = value[i];$
      assert $fee' = fee[i];$
    end
  end
  if $token[i] = 0$ **then**
    `// A fee token must not hold value`
    assert $value[i] = 0;$
  else
    `// There can only be one non-universal token type`
    assert $token[i] = token[0];$
  end
end
assert $bal_g \leftarrow bal_g - \underline{fee};$
assert $bal_g = 0;$
assert $bal_v = 0;$

Algorithm 4: Inbox Contract (Contract 2)

Data: Fraud-proof Period $fpp : \mathbb{N}$
`// Persistent storage`
$batchCommitments = \mathbb{N} \rightarrow (\mathcal{C}, I);$
$curHeight = \mathbb{N};$
$hnb = \mathbb{N};$ `// highest non-final batch`
$blockFin = \mathbb{N} \rightarrow \mathbb{N};$
$burnData = \mathbb{N} \rightarrow (\mathbb{N}^2 \rightarrow \mathcal{B});$
Function *newBatch*($blockBurnData : (\mathbb{N}^2 \rightarrow \mathcal{B}), prevBatch : \mathcal{B}$) **is**
  assert $Judge.hasStaked(msg.sender);$
  assert $batchCommitments[curHeight][0] == prevBatch;$
  while $(blockFin[hnb] < block.number) \wedge (hnb \leq curHeight)$ **do**
    $blockFin[hnb] \leftarrow \perp;$
    $hnb \leftarrow hnb + 1;$
  end
  $curHeight \leftarrow curHeight + 1;$
  $bh \leftarrow blobhash(0);$
  `// Force exactly one BLOB attached`
  assert $bh \neq 0 \wedge blobhash(1) = 0;$
  $batchCommitments[curHeight] = (bh, msg.sender)$;
  $burnData[blockHeight] = blockBurnData;$
  $blockFin[curHeight] \leftarrow block.number + fpp;$
end

Function *consumeBurnData*($batchId, bracketId, txId$) **is**
  assert $msg.sender = L2\_to\_L1\_Bridge;$
  assert $burnData[batchId][bracketId][txId] \neq \perp;$
  $tmp \leftarrow burnData[batchId][bracketId][txId];$
  $burnData[batchId][bracketId][txId] \leftarrow \perp;$
  return $tmp;$
end

Function *blockFinalized*($batchId$) **is**
  return $hnb > batchId;$
end

Function *revertL2Chain*($height : \mathbb{N}$) **is**
  assert $msg.sender = Judge;$ **assert** $blockFin[height] \neq \perp;$
  assert $blockFin[height] \geq block.number;$
  for $i = height$ **to** $curHeight$ **do**
    $blockFin[i] \leftarrow \perp;$
    $blockCommitments[i] \leftarrow \perp;$
    $burnData[i] \leftarrow \perp;$
  end
  $curHeight = height - 1;$
end

the tree, i.e., the coin commitments, are also published in the BLOB. This allows the construction of the tree path and the generation of the corresponding Merkle proof. Additionally, a transaction fee checkpoint ck_f is also at the beginning of the batch to record transaction fee collection.

The second part is the executed transaction brackets batch. Given the high transaction throughput and the fraud-proof scheme used to guarantee the security of Calyx, the goal here is to carefully define the published data structure so that a fraud-proof can be verified by referencing only a small, constant number of data words in BLOB. In Calyx, each published batch begins with a header containing the number m of transaction brackets, followed by m pointers to the beginning of the respective transaction bracket tx_i. A bracket serialization then begins with the bracket hash and the root of the coin tree after its execution, followed by the number of transactions and the number of required signatures.

To record transactions in a way that enables efficient access to any part of the transaction information, we first serialize the Inputs and Outputs, defined as the concatenation of the public information for inputs $s_{input}(I)$ and outputs $s_{output}(O)$ introduced earlier. The input information includes: the Coin Tree root crt, the serial number of the coin sn, the coin commitment cm, the public key for signatures pk_{sig}, and the zk-SNARK proof π. The output information includes: a bit indicating whether the serialization contains encrypted extra data about the private value, the coin commitment of this output c, and optional encrypted extra data C_{inf}. With such serialization, each transaction within a bracket begins with its transaction hash, which is defined as $h(s_{input}(I_0)||s_{input}(I_1)||\cdots||s_{output}(O_0)||s_{output}(O_1)||\cdots||fee||proof)$, ensuring that the commitment captures both the public components and the zero-knowledge proofs.

The transaction published along with the batch contains all public data required for verification. Each component is assigned a fixed byte length to enable efficient access under the pointer scheme. The transaction first specifies its token type, the number of inputs and outputs, pointers to the serialized inputs and outputs, and a reference that authenticates the coin tree root used by the input proof. This is followed by the fee field, the transaction proof, the serialized inputs and outputs, and finally the transaction signature.

For transfer transactions, the serialization follows this complete structure, explicitly specifying the number of inputs and outputs and including the transaction proof. In contrast, mint, burn, and fee-collecting transactions use implicit input or output structures. These transactions do not carry a transaction proof in the same form; instead, they include revealed inputs or outputs, such as a revealed output in a mint transaction or a revealed input in a burn or fee-collecting transaction.

On-chain fraud-proof. The verifier must generate a fraud-proof based on the data published by the operator. Leveraging the published data structure described above, the verifier can pinpoint a specific error by revealing only a small, constant number of BLOB data words to the Judge contract. The Judge contract first checks that these revealed data words are indeed part of the BLOB at the claimed positions by verifying them against the BLOB commitment recorded in the Inbox contract. It then parses the fraud-proof using the provided data, which allows the Judge contract to validate the proof. If the proof is valid, the verifier may claim the operator's stake as both punishment for the dishonest publisher and a reward for successful verification. The transactions published in the BLOB are considered executed only if no fraud-proof challenge is raised by a verifier within a specified time window.

We enumerate all possible violations of the protocol that could impact the benefits of honest clients as a set of rules from two perspectives: (1) *semantic validity*, i.e., the published data structure correctly follows the specified format; and (2) *proof validity*, i.e., validity proofs such as signatures, hash values, and zk-SNARK proofs can be correctly verified. The verifier can trigger the appropriate rule by submitting the corresponding fraud-proof evidence, requiring only a single round of communication with the Judge contract, rather than multiple rounds to localize the violation. We classify all rules into three categories:

1. Rules about the validity of **transactions**
   (a) Invalid transaction type. A transaction's type is not one of the three allowed types: mint, transfer, and burn.
   (b) Invalid mint transaction commitment. Can be disputed by recomputing commitment.
   (c) Invalid burn transaction commitment. Can be disputed by recomputing commitment.
   (d) Invalid input zk-SNARK proof. Can be disputed by verifying a specific input I_k's zk-SNARK proof $\pi_{I;k}$ with published parameters.
   (e) Invalid transaction zk-SNARK proof invalid. Can be disputed by verifying transaction proof π_{tx} with the published parameters.
   (f) Invalid transaction hash. Can be disputed by recomputing the hash value.
   (g) Transaction input or mint nonce double spending (not in the same bracket). Can be disputed by providing the double-spending serial number or bridge nonce s, the previous nullifier tree root ntr, and providing a Merkel tree inclusion proof that $s \in ntr$.
   (h) Transaction input or mint nonce double spends (in the same bracket). Can be disputed by providing two serial numbers s, s' in two inputs of transactions in the same bracket, and checking $s = s'$.
   (i) Invalid coin tree root. Can be disputed by providing the coin tree root ctr' referenced by the revealed transaction input $I_k = (\dots, ctr, \dots)$, and checking that $ctr \neq ctr'$.
2. Rules about the validity of transaction **brackets**

Algorithm 5: Judge Contract Pseudocode (Contract 3)

```
Data: Required minimal stake S, Fraud-proof Period fpp
stake = (ID → ℕ);
unstakeRequest = ID → (ℕ → ℕ);
Function hasStaked(id) is
    return stake[id] ≥ S;
end

Function stake() is
    stake[msg.sender] ←
      stake[msg.sender] + msg.value;
end

Function unstakeRequest() is
    tmp = stake[msg.sender];
    assert tmp > 0;
    unstakeRequest[msg.sender][block.number +
      fpp] ← tmp;
    stake[msg.sender] ← 0;
    // Can't retrieve stake before next
       fraud-proof period
end

Function unstake(sid) is
    tmp = unstakeRequest[(msg.sender, sid)];
    assert tmp > 0;
    assert sid ≥ block.number;
    unstakeRequest[msg.sender][sid] ← 0;
    msg.sender.transfer(tmp);
end

Function slash(loser, winner) is
    loserStake ← stake[loser];
    foreach key ∈ unstakeRequest[loser] do
        loserStake ←
          loserStake + unstakeRequest[loser][key];
        unstakeRequest[loser][key] ← 0;
    end
    stake[loser] ← 0;
    winnerReward ← loserStake;
    winner.transfer(winnerReward);
end

Function disputeBlock(bID, data, indizes, disputeType,
  auxData) is
    checkFraudProof(bID, data, indices, disputeType,
        auxData);// According to the rules
    Inbox.revertL2Chain(bID);
    slash(id_S, msg.sender);
end
```

(a) Invalid TX Bracket hash. Can be disputed by computing with all transaction hashes.

(b) Mismatch between input number and signature number. Can be disputed by providing the number of inputs and the signature number for checking.

(c) Invalid signature error. Can be disputed by verifying the signature σ_k with the public key pk_k of the input on the bracket hash.

(d) Empty or oversize transaction bracket. Can be disputed by providing the transaction count field for this transaction bracket.

3. Rules about the validity of transaction **batches**

(a) Burn transaction data on the contract does not correspond to the burn transaction in the batch. Can be disputed by providing the burn transaction in BLOB and comparing with the value (t, v, g, id_{L1}) stored in the Inbox Contract.

(b) Burn transaction not published to L1 blockchain but burn data is recorded in contract. Can be disputed by showing the transaction with index *txID* in BLOB is not a according burn transaction.

(c) Last transaction in batch is not a fee-collecting transaction or not alone in a bracket. Can be disputed by providing the type of the last transaction *txLast* and the number n_{tx} of transactions in the last bracket, and checking if *txLast* is not a fee-collecting transaction or $n_{tx} > 1$.

(d) fee-collecting transaction present in batch not at last position. Can be disputed by providing a transaction *tx* and revealing its type, checking that this transaction is not in the last position, and is a fee-collecting transaction.

(e) Intermediate fee calculation invalid. Can be disputed by recalculating the collected transaction fee based on the previous transaction fee checkpoint at the beginning of the batch.

(f) Invalid fee-collecting transaction. Can be disputed by recalculating the commitment value with the published parameter.

(g) Intermediate coin tree root calculation invalid. It can be disputed by providing the front of the Merkle tree before the disputed transaction bracket and recalculating the updated root based on the outputs.

(h) Intermediate nullifier tree root invalid. Can be disputed by providing the front of the Merkle tree before the disputed transaction bracket and recalculating the updated root based on the inputs.

5.3 Leaving Protocol

As the final procedure for Calyx, clients can leave the protocol by generating the burn transaction. After the transaction is executed, following the execution procedure defined before, the client can further retrieve the corresponding coins from the on-chain smart contract, which is shown in Algorithm 6.

A Burn Transaction consists of one coin with commitment *cm* that the client wants to peg out with. A Burn Transaction does not have an output. The body of a burn transaction contains: (1) the L1 address id_{L1} that is used to claim the burned assets on L1 through a contract. The bridge contract will check if the withdrawal request comes from this identity; (2) the private values $token, value, fee$ of the burnt coin; (3) the zk-SNARK proof to show the validity of such input; (4) a fee value fee for publishing the burn transaction. Note that the fee for burn transactions is higher than for transfer transactions, because the operator must pay for L1 storage for every burn transaction. This further protects against spamming L1 by financially incentivising users to first join their unused coins with the same token type by means of transfer transactions before burning them once.

Algorithm 6: L2 to L1 Bridge (Contract 5)

```
Function retrieve(batchId, bracketId, txId) is
    assert Inbox.blockFinalized(batchId);
    (t, v, g, id_L1) ←
      Inbox.consumeBurnData(batchId, bracketId, txId);

    if id_L1 = msg.sender then
        if t ≠ 0 then
            t.transfer(id, v); // Transfer tokens
                from bridge to user
        end
        if g ≠ 0 then
            id_L1.transfer(g); // Transfer ETH
                from bridge to user
        end
    end
end
```

6 Security Analysis

In this section, we prove the security and the privacy-preserving property of our Calyx protocol. We separately analyze these properties in the traditional Byzantine security model and the rational security model. Due to the page limit, we only show the general proof idea here, the detailed proof can be found in Appendix A and B.

6.1 Byzantine security model

Under the Byzantine security model, we first prove that Calyx realizes *balance security*, *liveness*. The general proof idea is that, in order to break these security properties, the adversary should be able to break the underlying assumptions, either the ideal underlying ledger or the ideal cryptographic tools that we are using in the system. We also prove Calyx realizes L2 transaction privacy through the L2-IND game.

Theorem 1 (Balance security)**.** *Calyx realizes balance security under the Byzantine security model.*

Theorem 2 (Liveness)**.** *As long as there exists one honest operator and one honest verifier, the* liveness *of Calyx is guaranteed.*

Theorem 3 (L2 transaction privacy)**.** *The Calyx protocol is an L2 transaction privacy-preserving optimistic-Rollup protocol through realizing L2 transaction indistinguishability.*

6.2 Rational security model

As for the analysis under the rational security model, we quantify the adversary's potential benefits from attempting to break *balance security*, *liveness*, and *privacy*, together with the corresponding costs of carrying out such attacks. By showing that the expected benefits of such attacks are strictly lower than those of honestly following the Calyx protocol, we obtain the following result:

Theorem 4 (Rational security & privacy)**.** *The Calyx protocols realize rational security and privacy under a rational security model if each operator deposits more than* $\frac{TVL}{|V|} + c_f$ *stake on-chain.*

7 Implementation and Evaluation

To demonstrate the feasibility of our protocol Calyx, we implemented the protocol's smart contract in Solidity v0.8.28, with Zokrates [3] and deployed it on an Ethereum testnet. We assume a gas base price of 8.66×10^{-9} ETH and a BLOB gas base price of 3.52×10^{-9} ETH, which are average values taken over the previous year. In the same way, we set the exchange rate between USD and ETH at 2690 : 1. The gas costs for each phase are shown in Table 2.

Deploying all the smart contracts introduced in Section 5 costs approximately 0.43265 ETH. This deployment is performed only once and supports long-term protocol operation. The relatively high cost arises from pre-listing all fraud-proof logic, which helps to significantly reduce the cost of potential L1 interactions. Moreover, since gas fees fluctuate, deployment can be scheduled when the base gas price is below average, potentially lowering the setup cost substantially. For a client to join the protocol, it must send a peg-in transaction to the smart contract on L1 and create a mint transaction, which the operator later publishes to L1. The combined cost of these two steps is 0.00216 ETH. The off-chain parts of protocol joining and transaction preparation do not require publishing to the L1 blockchain and therefore incur zero gas cost. We then evaluate the cost of publishing a batch to the L1 blockchain. The batch size is constrained by the BLOB size, and under our construction, a batch can include up to 269 mint transactions, 167 burn transactions, or 86 transfer transactions. The per-transaction publishing cost is 0.00002 ETH, which also reflects the execution cost for one transaction in the optimistic case, since no fraud-proof transactions need to be published. In the worst-case analysis, we consider the cost of publishing a fraud-proof under rule 1e, with the maximum of 8 inputs and outputs, whose cost is 0.2417 ETH. Finally, for a client to leave the protocol and peg out with its coins, they must send a burn transaction, which the operator publishes, followed by a transaction to retrieve the corresponding tokens on L1. This process costs 0.00189 ETH in total. In conclusion, since the highest gas costs arise only in pessimistic situations, and given that Calyx achieves L2 transaction privacy simultaneously, the overall gas overhead is acceptable.

	Deployment	Join	Transaction execution		Leave
			Publish per TX	Fraud-proof	
Gas	49959175	250455	2339	$\leq$ 2791028	219315
ETH	0.43265	0.00216	0.00002	$\leq$ 0.02417	0.00189
USD	1163.84	5.83	0.06	$\leq$ 65.02	5.11

Table 2: Gas fee evaluation for each phase of Calyx protocol. For the fraud-proof verification cost we analyze the worst situation of rule 1e.

Furthermore, we conduct an asymptotic comparison with two other Rollup protocols: the SNARK–aggregation–based zk-Rollup protocol, such as Aztec, and the Arbitrum optimistic-Rollup protocol. Assuming the protocol executes n transactions in one batch, the comparison is shown in Table 3.

Since all three protocols realize *data availability*, all executed transactions are published in the BLOB space of the L1 blockchain, resulting in $O(n)$ on-chain cost. For non-BLOB L1 blockchain cost, Aztec requires the on-chain smart contract to verify the aggregated proof provided by the operator [2]. Although no formal security and performance analysis of their aggregation scheme is available, prior work on SNARK aggregation [19] shows that, without special techniques such as preprocessing, verification requires $O(\log n)$ computation. In the case of Arbitrum, the use of the bisection scheme [15] introduces $O(\log n)$ on-chain challenge interactions between the contract and the verifier in pessimistic scenarios. Our Calyx protocol, under optimistic execution, requires only constant $O(1)$ on-chain checkpointing, similar to Arbitrum. However, due to its one-step fraud-proof scheme, Calyx further reduces the verification complexity to constant $O(1)$ interaction time and proof size. Finally, regarding privacy guarantees, Arbitrum provides none, since all transactions are published on-chain in plaintext. In contrast, both Aztec and Calyx achieve transaction-level privacy by leveraging zk-SNARKs.

	BLOB L1 cost	non-BLOB L1 cost		Privacy-preserving
Aztec	$O(n)$	$O(\log n)$		✓
Arbitrum	$O(n)$	Optimistic	Pessimistic	×
		$O(1)$	$O(\log n)$	
Calyx	$O(n)$	Optimistic	Pessimistic	✓
		$O(1)$	$O(1)$	

Table 3: Asymptotic comparison for L1 costs and privacy-preserving property, assuming executing n transactions in one batch that is published on the L1 blockchain.

8 Extension

Apply to CBDC. Calyx is initially designed for privacy-preserving, scalable multi-token exchange on the Ethereum platform, but we believe the underlying idea can be applied to broader scenarios. For instance, Central Bank Digital Currencies (CBDCs) [23] have gained increasing attention in recent years as a convergence of blockchain systems and centralized payment infrastructures. Exploring how to achieve both privacy and efficiency in such a setting using the principles of Calyx represents an interesting research direction.

Transfer among different token types. In the current design of Calyx, each transaction requires that all inputs and outputs belong to a single token type, ensuring that no loss occurs due to dramatic exchange rate fluctuations in the Ethereum token market [17]. We believe Calyx can be extended to support transfers across different token types if a fixed globally agreed-upon exchange rate is encoded at the time the smart contract is deployed.

Extend to contract execution. The Calyx protocol is initially designed for transaction execution, and thus the fraud-proof rules and zk-SNARK circuits are tailored to provide security guarantees in this setting. If detailed security guarantees, fraud-proof schemes, and corresponding zk-SNARK circuits can be defined for specific smart contract executions, we believe the design of Calyx can be further extended to support contract execution scenarios.

9 Conclusion

In this paper, we propose the very first privacy-preserving multi-token optimistic-Rollup protocol, termed Calyx. Calyx leverages zk-SNARKs to protect L2 transaction privacy, combined with a carefully designed interaction model with the L1 blockchain to achieve state consistency across layers and to support efficient L1 fraud-proof interaction. Beyond presenting the protocol, we analyze its security under both the Byzantine and rational adversary models. We also formally define L2 transaction privacy for Rollup protocols and provide a corresponding analysis. Finally, we implement the on-chain components of Calyx, demonstrating its practicality.

Acknowledgement

This work was partially supported by the Vienna Science and Technology Fund (WWTF) through the project 10.47379/ICT22045; by the Austrian Science Fund (FWF) through the SpyCode SFB project F8510-N and F8512-N; by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 101141432-BlockSec).

References

[1] Coinmarketcap. https://coinmarketcap.com/ [Accessed: 2025-04-24 16:27].

[2] "the aztec protocol", 2018.

[3] Zokrates: A toolbox for zksnarks on ethereum. https://zokrates.github.io/, 2025. Accessed: 2025-08-24.

[4] Roman Storm Alexey Pertsev, Roman Semenov. Tornado cash privacy solution, 2019. https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf [Accessed: 2025-04-29 17:00].

[5] Kurt M Alonso et al. Zero to monero. *Zero to monero*, 2020.

[6] Astar Network Team. Ethereum layer 2 scaling technologies explained. https://astar.network/blog/ethereum-layer-2-scaling-technologies-explained-39982, 2023. Accessed: 2025-05-12.

[7] Alexandre Belling, Mary Maller, and Sarah Meiklejohn. Zeth: On integrating zerocash on ethereum. *arXiv preprint arXiv:1904.00905*, 2019.

[8] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[9] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery.

[10] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 423–443, Cham, 2020. Springer International Publishing.

[11] EScholar: Electronic Academic Papers for Scholars. Ethereum After EIP-4844: 82 % Lower Costs for Rollups, August 2025. Published August 13, 2025. Empirical findings on how EIP-4844 (blobs) reduced gas costs for both optimistic and ZK rollups, enabling direct comparison.

[12] Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. A security framework for distributed ledgers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1043–1064, 2021.

[13] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 201–226. Springer, 2020.

[14] Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, version 2024.5.1-112-gcf7a5c [nu6], 2024. https://zips.z.cash/protocol/protocol.pdf [Accessed: 2025-04-28 10:00].

[15] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, 2018.

[16] Matter Labs. zksync: Trustless scaling and privacy solution for ethereum. https://zksync.io/, 2020. Accessed: 2025-05-13.

[17] OKX Learn. Neiroeth and cex tokens: Why wintermute's bold move amid a 97% price crash matters. OKX Learn, August 2025. Accessed: 2025-08-24.

[18] Lee Bousfield et al. Arbitrum nitro: A second-generation optimistic rollup, 2022. https://github.com/OffchainLabs/nitro/blob/master/docs/Nitro-whitepaper.pdf [Accessed: 2024-11-25 14:00].

[19] Xun Liu, Shang Gao, Tianyu Zheng, and Bin Xiao. Snarkfold: Efficient snark proof aggregation from split incrementally verifiable computation. IACR ePrint Archive 2023/1946, December 2023. https://eprint.iacr.org/2023/1946.

[20] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[21] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, 2013.

[22] Shen Noether, Adam Mackenzie, et al. Ring confidential transactions. *Ledger*, 1:1–18, 2016.

[23] Peterson K Ozili. Cbdc, fintech and cryptocurrency for financial inclusion and financial stability. *Digital Policy, Regulation and Governance*, 25(1):40–57, 2023.

[24] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1510–1532, 2023.

[25] Optimism PBC. Optimism: Scalable ethereum infrastructure. https://www.optimism.io/, 2021. Accessed: 2025-05-13.

[26] Polygon Labs. What EIP-4844 will mean for polygon. Blog post, 2024. Announces Feijoa upgrade enabling blob support for zkEVM.

[27] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. https://lightning.network/lightning-network-paper.pdf [Accessed: 2025-04-31 17:00].

[28] Antoine Rondelet. Zecale: Reconciling privacy and scalability on ethereum. *arXiv preprint arXiv:2008.05958*, 2020.

[29] Scroll Team. Blobs are here: Scroll's bernoulli upgrade. Blog post, 2024. Describes Scroll's use of EIP-4844 blobs for L1 data availability.

[30] Adrià TORRALBA-AGELL, Ghazaleh Keshavarzkalhori, Cristina Pérez Solà, David Megías, Jordi HERRERA-JOANCOMARTÍ, et al. Unmasking the illusion: The shortcomings of "zero-knowledge" rollups in achieving privacy. *XVIII Reunión Española sobre Criptología y Seguridad de la Información: XVIII RECSI, León 23-25 octubre 2024*, pages 361–366.

[31] Christof Ferreira Torres, Ramiro Camino, et al. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1343–1359, 2021.

[32] Friedhelm Victor. Address clustering heuristics for ethereum. In *Financial Cryptography and Data Security Workshops (FCW)*, 2020.

[33] Visa Inc. https://usa.visa.com/solutions/crypto/deep-dive-on-solana.html [Accessed: 2024-12-09 14:00].

[34] Fabian Vogelsteller and Vitalik Buterin. Erc-20: Token standard. Ethereum Improvement Proposals, Nov 2015. https://eips.ethereum.org/EIPS/eip-20 [Accessed: 2024-12-21 15:00].

[35] Anton Wahrstätter, Jens Ernstberger, Aviv Yaish, Liyi Zhou, Kaihua Qin, Taro Tsuchiya, Sebastian Steinhorst, Davor Svetinovic, Nicolas Christin, Mikolaj Barczentewicz, et al. Blockchain censorship. In *Proceedings of the ACM Web Conference 2024*, pages 1632–1643, 2024.

A Proof for the Byzantine security model

A.1 Balance security

The main idea for proving balance security in Calyx is to show that the adversary cannot affect the correctness of each participant's L2 and L1 states. In the following, we demonstrate that across the three phases of Calyx —protocol joining, L2 transaction execution, and protocol leaving—the adversary is unable to influence honest participants.

Protocol joining. The security goal for this first step can be concluded as following Lemma:

Lemma 1 (Correct joining)**.** *For any deposit on the L1 blockchain, a corresponding Calyx L2 state is initialized through a mint transaction.*

Proof. Assume the lemma does not hold. Then the violation must occur in one of the following three cases:

1. A mint transaction occurs on L_2 without a corresponding *LOCK* event on L_1. In such a case, This misbehavior can be challenged by V using rule 1b, which checks the revealed output values of the mint transaction against the claimed bridge nonce. Since the bridge also records the block number at which the nonce was created, the Judge can verify the "happened earlier" relation.

2. A *LOCK* event is claimed twice with two different mint transactions. In such a case, as mint nonces are treated as nullifiers, rules 1g and 1h prevent a mint nonce from being used more than once in mint transactions.

3. A mint transaction occurs on L_2 with a corresponding *LOCK* event on L_1, but is challenged and invalidated by a malicious V. The malicious V can only challenge through rules 1b, 1g, and 1h. If V is able to successfully challenge through rule 1b, then the *LOCK* event triggered by a committed deposit transaction would be reverted, which contradicts the assumption of an ideal underlying L1 blockchain. If V is able to successfully challenge through rules 1g or 1h, then the adversary must

be able to create a hash collision without knowing the secret input of the honest joining client, which contradicts the assumption of an ideal hash function.

Conclusively, in Calyx the correctness of protocol joining cannot be broken by the adversary. □

L2 transfer transaction execution. According to Calyx as introduced in 5, a regular L2 transfer transaction is considered executed only after it is published on the L1 blockchain and no fraud-proof is submitted by a verifier within a specified time window. We state the security property of L2 transfer transaction execution in Calyx with the following lemma:

Lemma 2 (Correct L2 transfer transaction execution)**.** *No honest participants will lose coins during the execution of L2 transfer transactions in Calyx.*

Proof. We prove the lemma by enumerating the possible violations and showing that each contradicts our underlying assumptions.

Since Calyx adopts the UTXO model for L2 transactions, the only way an honest client could lose a coin is if the coin were spent without the client's permission. For this to happen, either the adversary would need to forge a valid signature or learn the hidden secret from the zk-SNARK proof. Both cases contradict our assumptions of an ideal signature scheme and an ideal zk-SNARK scheme. □

Protocol leaving. As the final procedure, our protocol must guarantee that each honest client can leave the protocol with its corresponding L2 state. The adversary may attempt to influence this procedure in two ways. First, the adversary could try to disrupt the execution of a client's leave request. Second, the adversary could attempt to execute invalid transactions, thereby affecting the token storage in the Calyx contract and indirectly impacting the success of an honest client's departure. In the following, we analyze these two situations separately.

For the first possible influence, we have the following Lemma:

Lemma 3 (Correct execution of leaving request)**.** *In Calyx, the leaving request from the honest client will not be invalidated once published on the L1 blockchain.*

Proof. First, the adversary could attempt to disrupt the procedure by publishing an invalid burn transaction. However, a burn transaction must include a revealed input that is checked by rule 1c. In addition, the L1 beneficiary address is verified by rule 3a. Therefore, as long as an honest verifier exists, any such violation will be detected and challenged.

Second, the adversary could attempt to influence the procedure by corrupting the verifier and trying to invalidate a valid burn transaction with a fraudulent challenge. In this case, the verifier can only invoke rules 3a and 3b. Once the burn transaction has been published on the L1 blockchain, the checks in these rules will succeed unless the consistency of the underlying L1 blockchain is broken, which contradicts our assumption.

Finally, without a correct burn transaction included in the BLOB and the corresponding information, such as the beneficiary address, committed to the smart contract, no party is able to retrieve coins from the contract. □

For the second possible influence, we have the following Lemma:

Lemma 4 (Correct token value change)**.** *In Calyx, the adversary cannot alter the total deposited token amount through invalid transactions.*

Proof. The adversary could attempt to change the total deposited token amount in the following ways: (1) double-spending tokens; (2) creating transactions where the output value does not match the input value; (3) generating multiple mint or burn transactions.

For the first case, rule 1d ensures that the input zk-SNARK proof is correct. A correct input zk-SNARK proof guarantees that the deterministic serial number (nullifier) is revealed for each input. All nullifiers ever revealed will appear in the nullifier tree committed to by all *following* transaction brackets. Thus, rule 1g prevents the reuse of nullifiers across different transaction brackets, while rule 1h prevents double spending within the same bracket.

For the second case, rule 1e ensures that every zk-SNARK proof in transfer transactions is correct, and a correct zk-SNARK proof guarantees that each transfer transaction balances. Furthermore, every input to a transfer transaction must be correctly derived from a coin output, which is also validated by the input zk-SNARK proof, whose correctness is guaranteed by rule 1d. Therefore, as long as there exists one honest verifier, any imbalance can be detected and disputed.

For the third case, if a *LOCK* event is claimed twice, mint nonces are treated as nullifiers, and rules 1g and 1h prevent the reuse of mint nonces in multiple mint transactions. If a nullifier is revealed in a transaction bracket txb, then it will appear in all nullifier trees of later brackets txb' with $txb < txb'$. Similarly, if a burn transaction were to be claimed twice, this would fail since the bridge contract on L1 clears the reference to that specific burn transaction once claimed. Therefore, as long as at least one honest verifier exists, such violations will be detected and successfully disputed. □

Based on the conclusions above, we can have the following result:

Theorem 1 (Balance security)**.** *Calyx realizes balance security under the Byzantine security model.*

A.2 Liveness

Since the execution of a transaction in Calyx requires two steps: the operator publishing transaction batches on the L1 blockchain and the subsequent verification by a verifier, we prove that liveness is guaranteed in Calyx under the following assumptions:

Theorem 2 (Liveness)**.** *As long as there exists one honest operator and one honest verifier, the* liveness *of Calyx is guaranteed.*

Proof. First, as long as there exists at least one honest operator in Calyx, the transactions of honest clients will eventually be published on the L1 blockchain once they are delivered to the operator under our asynchronous communication model. A round-robin operator election can further ensure that the honest operator has the opportunity to publish batches onchain. Any batch published by a dishonest operator will be disputed by the honest verifier and therefore will not affect the execution of transactions from honest clients.

Then, the only way for the adversary to influence liveness by corrupting the verifier is to provide a valid fraud-proof π against correct behavior from honest clients and honest operators. After a deposit transaction, transaction batch, or checkpoint has been committed on the L1 blockchain, if the verifier were able to generate a proof of non-existence of such actions, it would require reverting the underlying L1 blockchain, which contradicts the assumption of an ideal blockchain. Another possible way to construct a fraud-proof would be to either create a collision in the hash function, compute a preimage of a hash value or an encrypted message, or generate a valid zk-SNARK proof for an invalid input. All of these contradict our assumption of ideal cryptographic tools. Thus, a verifier cannot influence liveness. □

A.3 Privacy

To prove the privacy of Calyx, we need to prove that there exists no PPT adversary $\mathcal{A}$ that can win the L2-IND game with non-negligible probability.

A.3.1 Game and oracle definition

First of all, the oracles used in the game are defined as follows:

Ledger Oracle: A ledger oracle $\mathcal{L}_1$ allows for two types of requests:

- $WriteOnL1(tx)$ takes a valid transaction tx and append to the transaction list it maintained.
- $ReadOnL1()$ returns all the committed L1 blockchain transaction list and states.

Furthermore, since the transaction list of Calyx is also published on the L1 blockchain, here we similarly define a L2 ledger oracle $\mathcal{L}_2$ that is connected to $\mathcal{L}_1$ and also takes two types of requests:

- $CommitBatch(batch)$ takes an input batch and appends to the L1 transaction list first, and then to the L2 transaction list if no fraud-proof response is received by interacting with the verifier oracle $\mathcal{V}$.
- $ReadOnL2()$ returns all the committed L2 transaction list and states.

Client Oracle: A client oracle $\mathcal{H}$ needs access to both the L1 and the L2 ledger oracle and models the behaviour of clients and allows for the following queries:

- $CreateAddress(pp)$ creates a new L1 address id_{L1} (generated from pk_{sig}) and corresponding key pair $pk = (pk_{sig}, pk_{enc}, pk_{coin})$ and $sk = (sk_{sig}, sk_{enc}, sk_{coin})$ based on public parameter pp. The mapping of the L1 address and the key pair is stored in the oracle. Noted that the key pair can also be generated separately.
- $PegIn(id_{L1}, a)$ takes the L1 address id_{L1}, asset a, the encoded contract address B and prepare a pegin transaction tx_{pegin} and send to $\mathcal{L}_1$ through request $WriteOnL1(tx_{pegin})$. Then, it creates a fresh nonce n and records the mapping. And finally returns (n, tx_{pegin}).
- $Mint(id_{L1}, a, n)$ checks on the L1 oracle through $ReadOnL1()$ request whether n is a bridge nonce that was awarded for locking a assets and checks that no mint transaction for that bridge nonce was created by this oracle. If a check fails, it returns $\perp$. Then, it calls $CreateAddress$ to create a new coin keypair (pk_{coin}, sk_{coin}). Based on (pp, n, a, pk_{coin}), use the hash and commitment scheme, generate and return a new mint transaction tx_{mint} containing coin c. Coins created by this oracle are added to the coins array $COINS$, and their index in this array is denoted by $cid(c)$. Finally, creates a transaction bracket txb for mint transaction and returns $(cid(c), txb)$.
- $RevealValues(cid, sk)$ takes a coin index cid, and checks whether sk proves the ownership of the coin with the decryption and commitment scheme. If check passes, the oracle returns the private values t, v, g of $COINS[cid]$. Otherwise, it returns the random value attached to the coin's transaction.
- $Transfer$ takes a list of coin commitments $(cid_0, cid_1, ..., cid_k)$ and a list of intended outputs $((pk_0, t_0, v_0, g_0), (pk_1, t_1, v_1, g_1), ..., (pk_n, t_n, v_n, g_n))$.
If any of the referenced coins do not exist through $ReadOnL2()$ check, a transfer or burn transaction has already been created for a coin, or the query cannot be fulfilled with a correct transfer transaction, the oracle returns $\perp$. It attempts to create a transfer transaction tx that spends all input coins and creates the requested outputs. All output coins created are added to the

COINS array and their indices are returned alongside *txb*, which is a transaction bracket that contains *tx*.

- $Burn(cid, id_{L1}, sk)$ takes a coin index *cid* and a L1 address id_{L1}. If a coin c exists at this position for which no transfer or burn transaction has yet been created, a bracket containing a burn transaction that burns this coin and specifies id_{L1} as beneficiary address is created and returned.

- $PegOut(cid, id_{L1})$ takes a coin index *cid* and a L1 address. It creates and returns a L1 transaction that claims these assets if *cid* is a coin for which a burn transaction is already placed on L2, which is checked through request $ReadOnL2()$. Only one such transaction can be created for the same burn transaction.

Note that this oracle can also be queried with a list of mint, transfer or burn transactions, in which case a single transaction bracket is returned that contains all created transactions. If any transaction fails to be created, then no bracket is returned at all. Further note that $\mathcal{H}$ will automatically run the coin reception algorithm *Receive* with all its maintained identities.

Operator Oracle: An operator oracle $\mathcal{S}$ needs access to both the L1 and the L2 ledger oracle and models the behaviour of a live, honest operator that can be queried to create batches from transaction brackets.

It allows for the following queries:

- $AddBracket(txb)$ takes a bracket from clients and attempts to add it to its mempool by updating $M' = M + txb$, and checks that the resulting blob serialization does not exceed the maximum BLOB size. If it doesn't exceed the size, then it sets $M := M'$, i.e. it updates its mempool. Otherwise, it creates a new L1 identity (pk, sk) using $CreateAddress(pp)$, takes the brackets from its mempool M, and creates a new valid batch b. This batch is then sent to $\mathcal{L}_2$ through $CommitBatch$ request, and the mempool is reset: $M := ()$.

Verifier Oracle: The verifier oracle $\mathcal{V}$ takes the following request:

- $CommitBatch(batch)$ takes an appended *batch* and will return a corresponding fraud-proof if there exists any invalid transaction there, otherwise outputs nothing.

Then we can formally define the L2-IND game as follows:

Definition 5. (L2-IND game) The L2-IND game is a round-based game played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ that proceeds as follows: First is the setup phase, $\mathcal{C}$ publicly creates $pp \leftarrow Setup(1^{\eta})$ with security parameter η, and then it creates the following:

- Two L1 ledger oracles: $L_{1;0}$ and $L_{1;1}$.

- Two L2 ledger oracles: $L_{2;0}$ and $L_{2;1}$.

- Two client oracles: $\mathcal{H}_0$ and $\mathcal{H}_1$.

- Two operator oracles: $\mathcal{S}_0$ and $\mathcal{S}_1$.

- Two verifier oracles: $\mathcal{V}_0$ and $\mathcal{V}_1$.

Furthermore, $\mathcal{C}$ privately samples a random bit $b \leftarrow_\$ \{0,1\}$. Then in each round, $\mathcal{A}$ sends to $\mathcal{C}$ a pair of the same type of queries (Q_0, Q_1). $\mathcal{C}$ first checks if Q_0 and Q_1 are publicly consistent, which is defined in Definition 6. If they are not, then $\mathcal{C}$ rejects this query and asks for the next one. Else, depending on the type, the following happens:

- If the query type is $CommitBatch$, then $\mathcal{C}$ forwards Q_0 to $L_{2;0}$ and Q_1 to $L_{2;1}$. Then, $\mathcal{C}$ asks $\mathcal{V}_0$ and $\mathcal{V}_1$ if the appended batches are correct. If at least one of them outputs a fraud-proof, $\mathcal{C}$ outputs 0 (indicating that the Adversary lost the game)

- If the query type is $WriteOnL1$, *PegIn* and *PegOut*, then $\mathcal{C}$ forwards Q_0 to $L_{1;0}$ and Q_1 to $L_{1;1}$. If either ledger rejects the transaction, $\mathcal{C}$ outputs 0.

- The Adversary may end the game by sending a guess $b_{\mathcal{A}}$. $\mathcal{C}$ outputs 1 iff $b_{\mathcal{A}} = b$, else 0.

- Otherwise, the query is a query to the client oracle. $\mathcal{C}$ forwards Q_0 to $\mathcal{H}_b$, which results in some information output like a newly created public key and newly created coin indices, an L1 transaction tx_{L1}, or an L2 transaction *txb*. $\mathcal{C}$ forwards any L1 transaction to $L_{1;b}$. $\mathcal{C}$ forwards any L2 transaction bracket *txb* to $\mathcal{S}_b$. Any information output is returned to $\mathcal{A}$.

  $\mathcal{C}$ performs the same procedure with Q_1, but instead of forwarding it to $\mathcal{H}_b$, it forwards it to the other honest client oracle $\mathcal{H}_{1-b}$. Any L1 transaction is forwarded to $L_{1;1-b}$, any L2 transaction bracket to $\mathcal{S}_{1-b}$.

The Adversary may also read the state of any ledger oracle at any time. Finally $\mathcal{A}$ sends the *guess* request with $b_{\mathcal{A}}$ to $\mathcal{C}$ and is considered to win the game if $b_{\mathcal{A}} = b$ with non-negligible probability greater than $\frac{1}{2}$.

Definition 6. (Public Consistency) Given two queries Q_0, Q_1 of the same type we call them **publicly consistent** according to the following rules:

1. All $CreateAddress$ queries are publicly consistent, but we require that both oracles create the same address.

2. If the type is $WriteOnL1, CommitBatch, PegIn_{L1}, Mint$, *Burn* or $PegOut_{L1}$, these queries are publicly consistent if and only if all public values are equal in both queries.

3. If the type is $Transfer$, the following rules determine if the queries are publicly consistent:

(a) The size and total transaction value of both queries must be equal, i.e., the number of inputs and outputs of Q_0 must match the number of inputs and outputs of Q_1.

(b) Both queries Q_0, Q_1 have the same results in the validity checking conducted by the oracle.

(c) The public information (e.g., the fee) on both queries must be equal.

(d) Both queries must also be equivalent with respect to the information available to $\mathcal{A}$, which means

i. For every output of Q_0 that specifies a recipient address which is controlled by the adversary (i.e., it was not created by *CreateAddress*), then the assets of this output must match in both queries. The same applies to Q_1, respectively.

ii. If an input i_j of Q_0 references (in the ledger $L_{2;b}$) an output contained in a transaction that was created by the Adversary (i.e., it was appended to the ledger by means of a *CommitBatch* query), then the corresponding input i'_j in Q_1 must reference (in the ledger $L_{2;1-b}$) a coin commitment that also appears in a transaction posted via a *CommitBatch* query. Furthermore, the assets of these inputs i_j and i'_j must be equal in both queries. The same applies to Q_1, respectively.

Recall that the adversary is allowed to send a list of the above queries that request L2 transactions, in which case all resulting transactions are packed inside a single bracket. In this case, these lists of queries are publicly consistent iff the number of queries is the same in both lists, and every corresponding pair of queries is publicly consistent.

Finally, we can prove the *L2 transaction privacy* of Calyx by conducting the L2-IND experiment.

Theorem 3 (L2 transaction privacy)**.** *The Calyx protocol is an L2 transaction privacy-preserving optimistic-Rollup protocol through realizing L2 transaction indistinguishability.*

Proof. We assume that there is an adversary A running in probabilistic polynomial time that has a non-negligible advantage p_A in the L2-IND game. Using a series of modifications for which we prove are at most distinguishable with negligible probability from the real L2-IND game we arrive at a simulation $\mathcal{G}_S$. No oracle answer from the simulation $\mathcal{G}_S$ depends on the bit b, thus the advantage any adversary A' can have in $\mathcal{G}_S$ is precisely 0. This results in a contradiction, thus completing the proof that no adversary A can exist.

The behaviour of the verifier $\mathcal{C}$ in the simulation $\mathcal{G}_S$ differs from its behaviour in the L2-IND game as follows: When running the $Setup$ algorithm to create the public parameters pp, $\mathcal{C}$ stores the trapdoor information $trap_{input}, trap_{tx}$ for the input and transaction zk-SNARK proof schemes. $\mathcal{C}$ will use that information to be able to provide zk-SNARK proofs for any witness/statement pair (x, w) that will be accepted by $\mathcal{V}_0$ and $\mathcal{V}_1$, regardless whether (x, w) are in the relation or not. When $\mathcal{C}$ receives a publicly consistent query pair Q_0, Q_1 that it would normally forward to $\mathcal{H}_b, \mathcal{H}_{1-b}$, it instead forwards it to $\mathcal{M}_0, \mathcal{M}_1$. Such an oracle $\mathcal{M}$ behaves as follows:

- If the query type is *CreateAddress*, it runs $(pk, sk) \leftarrow CreateAddress(pp)$ to obtain an address. Recall that $pk = (pk_{sig}, pk_{enc}, pk_{coin})$, where pk_{coin} is the public key used in the coin commitment and pk_{enc} is the public encryption key used to send private coin data to the receiver. Then, it randomly samples pk'_{coin} from the same distribution from which the coin public keys are sampled, and sets $pk' = (pk'_{coin}, pk_{enc})$. It stores sk and returns pk'.

- If the query type is $Transfer$, the way it calculates the inputs for the resulting transfer transaction differs from the real algorithm $Transfer$ in the following way:
  - Randomly sample a serial number sn' that is equal in length to real serial numbers and use sn' instead of the real sn.
  - Randomly sample a public key authorizer pk'_{auth} that is equal in length to real public key authorizers and use pk'_{auth} instead of the real pk_{auth}.
  - The remaining values are constructed as defined in the $Transfer$ algorithm.
  - The resulting input I' generated from sn', pk'_{auth} is not a valid input, thus an honest zk-SNARK prover cannot create a valid input proof for I'. However, since $\mathcal{M}$ is given access to $trap_{input}$, it is able to forge a valid zk-SNARK proof π'_I that will be accepted by the Verifier.

  Also, the way $\mathcal{M}$ calculates the outputs for the resulting transfer transaction differs from the real $Transfer$ algorithm:
  - If the query requests that the coin is sent to an address $pk =: (pk_{sig}, pk_{enc}, pk_{coin})$ maintained by $\mathcal{M}$, a random coin commitment value cm' is generated. Furthermore, the encrypted extra data d_{enc} is obtained by encrypting random data of the correct length under pk_{enc}.
  - If the query requests that the coin is sent to an address not maintained by $\mathcal{H}$ (i.e., it is under control of the Adversary), $\mathcal{M}$ correctly constructs the output as specified in the $Transfer$ algorithm.
  - The remaining values are constructed as defined in the $Transfer$ algorithm.

Clearly, the resulting transaction tx' generated from $\mathcal{H}$ is not a valid transaction. However, since $\mathcal{M}$ is given access to $trap_{tx}$, it is able to forge a valid zk-SNARK proof π that will be accepted by the Verifier.

Specifically note that in this simulation, no query response given to the $\mathcal{A}$ depends on bit b. Thus, any advantage the Adversary can have in this simulation is 0. We proceed in showing that the Adversary can distinguish such a simulation S from a real execution of the game at most with negligible advantage by providing a series of modifications. Denote with $\mathcal{G}_{real}$ the real game, and $\mathcal{G}_S$ the simulation S.

From $\mathcal{G}_{real}$ to $\mathcal{G}_1$. This modification consists of faking all zk-SNARK proofs using the trapdoor information obtained from the *Setup* algorithm. This modification does not result in a different distribution of the zk-SNARK proofs, since we assumed that the zk-SNARK scheme used is zero-knowledge.

From $\mathcal{G}_1$ to $\mathcal{G}_2$. This modification consists of replacing the encrypted data in outputs that are maintained by an honest client with a pure random value. The adversary does not have access to the honest client's secret key, and since by our assumption on the encryption scheme used, the advantage that the adversary has in distinguishing ciphertexts from random is at most negligible.

From $\mathcal{G}_2$ to $\mathcal{G}_3$. This modification consists of replacing hash digests (coin public keys, serial numbers, public key authorizers) with random values, as defined in the above definition of $\mathcal{G}_S$. Since we assume that our hash function is a PRF, the advantage that the adversary has in distinguishing hash digests from random values is at most negligible.

From $\mathcal{G}_3$ to $\mathcal{G}_S$. This modification consists of replacing some coin commitments with random values, as defined in the above definition of $\mathcal{G}_S$. Note that coin outputs already no longer depend on the bit b, since any two outputs created in a pair of mint queries must commit to the same public values in order for the queries to be publicly consistent, to the random coin public key (introduced by $\mathcal{G}_3$) and random coin identifier. Since our construction uses the hash function as a hiding commitment scheme (by including random values not known to the adversary in the pre-image), the advantage that the adversary has in distinguishing commitments from random values is at most negligible. □

B Proof for the Rational security model

B.1 Balance security

Lemma 5. *A rational verifier will dispute all the invalid batches published on the L1 blockchain.*

Proof. If an invalid batch is added on chain, every verifier V has two choices of strategies:

- V disputes the batch. In this case, V will get the operator's stake as a reward. We assume that the cost of submitting the fraud-proof is far lower than the stake of the operator.
- V doesn't dispute the batch. In this case, the verifier receives no reward from Calyx. However, this verifier might have been bribed by the rational misbehaved entity $\mathcal{A}$ not to dispute the batch. By posting an invalid batch, $\mathcal{A}$ can create (incorrect) burn data on L1 that would allow $\mathcal{A}$ to withdraw up to the value TVL (all funds currently locked on the bridge from clients). Such an attack can be carried out by the following construction: $\mathcal{A}$ creates a smart contract sc that it specifies as the receiver of the stolen burned funds. sc is programmed so that it will correctly distribute the bribes to all verifiers. If the bribe b_k for a verifier k is greater than the reward for disputing, the verifier would not be incentivised to submit the fraud-proof.

  As long as the TVL is larger than the sum of all bribes required, this attack is more beneficial for $\mathcal{A}$. Here we assume s the minimal stake required to be an operator, c_f the worst-case cost to submit a fraud-proof for a verifier, TVL is the total value that clients locked in Calyx, and $|V| \geq 1$ the number of known verifiers in the system. A rational verifier among all $|V|$ will submit the fraud-proof if the following inequality holds:

  $$s - c_f > \frac{TVL}{|V|}$$

  Although Calyx is initially designed assuming anyone can become a verifier, in practice, there will be only a finite number of verifiers in the system, or only a finite verifiers who can react in a timely manner. Thus to protect the system is to increase the operator's stake as the TVL goes up, or increase the verifier amount accordingly.

□

B.2 Liveness

Lemma 6. *Calyx guarantees* liveness *for requests under rational model.*

Proof. As has been discussed before, the liveness could be influenced by the operator. Since the verifier is not able to generate a valid challenge to dispute valid transactions.

For a rational operator, there exist two choices of strategies:

- Publish the received valid transaction through batches. In this case, the operator is able to obtain the transaction fee provided by the client.

- Censor the received valid transactions. According to the rational model we have defined in Section 3, we do not consider that the participants have any external budgets and benefits for censoring a transaction for the whole challenge time period. By doing so, the operator will obtain a lower transaction fee compared to publishing all of them.

□

B.3 Privacy

Lemma 7. *Calyx guarantees* privacy *under rational model.*

Proof. We define the benefit of obtaining private information as b, the success probability as p, and the computational cost as c. Hence, the expected benefit for any participant attempting to retrieve private information is $p \times b - c$. In Calyx, privacy is ensured through secure cryptographic tools such as hash functions, zk-SNARKs, and commitment schemes. According to the security definitions of these primitives, the probability p of learning secret inputs is negligible. Therefore, as long as the security parameter is set appropriately, the expected benefit becomes negative, implying that no rational participant will attempt to compromise others' privacy. □

Conclusively, we can have the following result under the rational security model:

Theorem 4 (Rational security & privacy)**.** *The Calyx protocols realize rational security and privacy under a rational security model if each operator deposits more than* $\frac{TVL}{|V|} + c_f$ *stake on-chain.*