Qurrency: a quantum-secure, private, and auditable platform for digital assets

Arka Rai Choudhuri, Sanjam Garg[1,2], Matthew Gregoire[3], Keewoo Lee[4], Mike Lodder[5], Hart Montgomery[6], Guru Vamsi Policharla[1], and Jim Zhang[7]

[1] University of California, Berkeley
[2] Exponential Science Foundation
[3] University of North Carolina at Chapel Hill
[4] Ethereum Foundation
[5] Lit Protocol
[6] Linux Foundation Decentralized Trust
[7] Kaleido, Inc.

Abstract. Central bank digital currencies (CBDCs) and other related digital asset platforms have the potential to revolutionize the financial world. While these platforms have been deployed in test environments by virtually all large financial institutions, including central banks, there are still several limitations of these systems that prevent widespread adoption. These include (i) privacy, (ii) security against quantum adversaries, and (iii) auditability. In this work, we undertake (to our knowledge) the first formal study of these systems.

While there have been many digital asset platforms implemented, we do not know of any formal model for a fundamentally UTXO-based digital asset platform/CBDC. Our first contribution is a formal modeling of a UTXO-based private digital asset system that meets our requirements listed above. This model is loosely based upon the open source software that we found came the closest to meeting our requirements, Linux Foundation Decentralized Trust (LFDT) Zeto. In the course of our formal modeling, we helped to improve the security of Zeto. We then provide an efficient construction of such a system, which we call Qurrency. Qurrency is an efficient UTXO-based privacy-preserving token system that includes an auditing mechanism and is secure against "harvest now, decrypt later" attacks, which is critically important for several central banks, including the Bank of Brazil. We implemented our construction to show that it is practically efficient and can be used on any EVM-based blockchain system with ease.

1 Introduction

Decentralized finance (DeFi) has been a revelation in blockchain technology, and it is not an exaggeration to suggest that it has been one of the primary reasons for the explosive growth of cryptocurrencies: there is currently approximately 100 billion USD currently locked in DeFi protocols [24], indicating a huge ecosystem. However, traditional finance (TradFi) and central banks have been slower to adopt blockchain technology. Previously, regulation was widely cited as the main barrier to adoption, particularly in the US [5], but today, even central banks including the European Central Bank, Swiss National Bank, Bank of Canada, Bank of England, Bank of Japan, and the US Federal Reserve are experimenting with blockchain technology [57,49].

Perhaps surprisingly, the biggest barriers to TradFi and central bank adoption of blockchain technology today are technical. While different financial institutions vary in terms of what exact requirements they need to adopt blockchain technology, there are currently several main functional barriers to widespread adoption, including privacy, auditing, quantum security, interoperability, and regulatory compliance. A blockchain-based CBDC would ideally address these requirements in the UTXO model, which allows for much greater parallelism than in account-based token systems. In fact, in a very recent invited talk at the Real World Cryptography conference entitled "What would it take to operationalize UTXO-based settlement for central bank digital currency?" [63], Silvio Petriconi of the Bundesbank highlighted all of these concerns as major barriers to CBDC adoption. To our knowledge, no system available today successfully solves all of these problems simultaneously, which we explain in more detail below, as well as some other technical requirements for practical adoption of blockchain for these financial systems.

Privacy The lack of privacy in most CBDC deployments and proofs of concept has been cited as probably the largest bottleneck to further CBDC deployments, particularly in developed economies. In fact, the EU Parliament passed a regulation that explicitly required strong privacy guarantees in any CBDC

deployment [69], and the European Central Bank (ECB) has put in a considerable amount of work investigating privacy [22]. In the US, the House of Representatives passed a bill (H.R. 5403) banning retail CBDCs without explicit approval from congress with the primary stated motivation being protecting citizens' financial privacy [70], and many advocacy groups have issued statements about CBDC privacy concerns [28,3]. Large commercial financial firms, such as JP Morgan [64,51] and DTCC [66], have also frequently mentioned privacy in their documentation on and product launches of digital assets.

Perhaps the clearest articulation of the need for privacy on blockchains has come from the Banco Central do Brasil (BCB), which is piloting the Drex CBDC project. Privacy has been imperative to Drex from the beginning [13], but the BCB has, at multiple times, cited privacy as "challenging" [44] or still an obstacle [38] to a real-world CBDC deployment. The latest phase of development on the Drex was explicitly focused on privacy [31]. Most recently, the BCB has decided not to use blockchain in the upcoming Drex pilot (instead leaving it for a future v2 system) [68], and the lack of good privacy solutions in conjunction with other desired features, including auditing and others, was mentioned as one of the most important reasons for this delay.

Auditing Unlike cryptocurrencies, which are still mostly unregulated or loosely regulated, there are many legal restrictions that would apply to CBDCs or digital asset platforms where securities are traded. For instance, due to the FFIEC Bank Secrecy Act [30,58], banks that have a presence in the US must (securely) keep a record of any checks and deposits over 100 USD and any international payments for more than $10,000$ USD for five years for anti-money laundering and anti-terrorism purposes, and these restrictions would almost certainly apply to CBDCs. The Sarbanes-Oxley Act (H.R. 3763) imposes even more restrictions on data retention for accounting purposes [60], and securities laws can be even more complicated. These legal regulations seem to necessitate that at least some data related to transactions in these systems be stored encrypted on chain (rather than just committed to, for instance).

Both central and commercial banks have taken note of this and incorporated these requirements into their designs for CBDCs and digital assets. According to the US Federal Reserve [59], "Any CBDC would need to strike an appropriate balance between safeguarding the privacy rights of consumers and affording the transparency necessary to deter criminal activity." In addition, "Financial institutions in the United States are subject to robust rules that are designed to combat money laundering and the financing of terrorism. A CBDC would need to be designed to comply with these rules." Other jurisdictions agree. The Bank of Japan (BoJ) and the ECB had a similar discussion on balancing confidentiality and auditability in their report on Project Stella [4]. The Drex is being designed in a way that allows a "high degree of auditability, traceability, and transparency, ensuring proper oversight and regulation by the CBB" [20]. Deloitte [25] has pointed out that exact audit requirements will be varied between systems, so any practically reusable system design will need some flexibility to serve all use cases.

Quantum Security Security against quantum computers is a well-known issue for anyone involved in cryptography or computer security. Banks and other financial institutions have certainly taken note when designing systems or system requirements for CBDCs and digital tokenization platforms. The Drex project is explicitly targeting post-quantum (PQ) security [65], and as a result there has even been considerable work in the Brazilian community on building PQ variants of (permissioned) Ethereum [23]. Project Leap [18], a collaboration between the Banque du France, the Bundesbank, and the Bank of International Settlements (BIS) focuses on the need for post-quantum cryptography (PQC) in financial applications, including blockchain-related ones.

There has been quite a bit of work and analysis done on the importance of PQC for blockchain-based applications [42], including a report from the World Economic Forum (WEF) and Accenture about the importance of PQC for CBDCs [54]. The biggest issue is, as with most deployments of cryptography in general, the so-called "harvest now, decrypt later" problem. In other words, any encryption or commitments on chain that aren't perfectly (i.e. information-theoretically) hiding need to be quantum secure, because otherwise an adversary could see them, record them, and attack them later when quantum computers become powerful enough. Blockchains that don't store encrypted data typically take a much more relaxed approach to the quantum threat; for instance, the Ethereum roadmap [32] calls for switching signatures and commitments to post-quantum variants but does not yet have a firm timeline.

Programmability Programmability to support multi-step flows and atomic settlement, which is necessary for interoperability between different tokens and platforms, is an important requirement in financial service use cases. It's important that the assets committed for the final settlement are not spent elsewhere while the overall business transaction is still pending. In token implementations without privacy support

this can be easily achieved by first transferring the committed assets to an escrow account managed by a smart contract, which has logic to settle or to refund, such as using a Hash Timed Lock Contract (HTLC) with ERC20 (for fungible assets) or ERC721 (for non-fungible assets) token standards. But such techniques are not possible with privacy enabled tokens because smart contracts cannot generate zero-knowledge proofs of ownership, and thus they cannot own assets. An alternative mechanism using locking and unlocking is necessary to achieve this function, which is described in this paper.

KYC KYC (Know Your Customers) is another prevalent requirement in financial services use cases for regulatory compliance. To achieve this in a privacy enabled token implementation, a transaction must prove that both the sender and the receiver are on the approved participant registry, without divulging the identities of the parties.

Anonymity Finally, we also found that anonymity is not a universally supported feature in existing privacy tokens. For instance, "vanilla" Zether (i.e. without the extension described in [16] Appendix D) and the systems built upon it are not anonymous. In use cases that involve competing parties, such as commercial banks in a wholesale CBDC network, it is critical that the transactions do not give away the identities of the counterparties.

1.1 Our Contributions

In this work, we provide the first formal model of a privacy-preserving, auditable UTXO-based system for CBDCs, digital assets, tokenized deposits, and stablecoins that addresses all of the necessary practical problems (like interoperability and KYC) for these systems to be useful in the real world. We then build the first such system that achieves strong privacy guarantees, is secure against quantum "harvest now, decrypt later" attacks, and offers strong auditability functionalities. We call this system, which can be implemented as a smart contract or series of smart contracts on (essentially) any EVM-based chain "Qurrency." We also formally prove Qurrency secure in our new model. In addition, we implemented Qurrency in the Zeto framework via a new token implementation and used this implementation to demonstrate that Qurrency has excellent performance.

Formal Modeling Our first contribution is a formal model for a UTXO-based privacy-preserving token system that supports auditability, is "quantum harvest-now, decrypt later" secure, and includes all of the practical features necessary for a real-world implementation of a CBDC or digital asset platform. Our definitions and modeling take inspiration from Zerocash [9], Zether [16] and Hawk [47] and perhaps most importantly, the Zeto [15] codebase itself.

Like Zether, we model a smart contract that can be built on any EVM chain. We formally define an ideal functionality (inspired by Hawk) that captures properties like balance (inspired by Zerocash), auditability, and quantum privacy. Our model also includes locking functionalities, which is critical for asset swaps and any kind of blockchain interoperability, which is something that we have not seen in other models as well. We capture all of the practical functionalities in the Zeto codebase that are essential for deployments but often overlooked by theoretical analysis.

Qurrency Design We designed and built Qurrency, a secure UTXO-based privacy-preserving token system that can be used to build CBDCs and other digital asset platforms. We emphasize that Qurrency enables audits and offers secure *privacy* for users against quantum attacks. At an extremely high level, Qurrency utilizes a "Groth16" proof system [36] to build a privacy-preserving token system without auditability, but with *quantum privacy*. Qurrency then requires each transaction to be encrypted using ML-KEM [43], and then includes in its proof system an additional constraint that the plaintext of the ML-KEM ciphertext matches the inputs to the Groth16-based token system.

The core idea of using a ciphertext and a proof of equivalence with a privacy-preserving token system is not a novel idea. In fact, setting aside concerns about efficiency and compliance with standards, it is possible to achieve such a construction in a straightforward manner using post-quantum encryption and quantum-secure, general-purpose zero-knowledge proof (ZKP) systems, such as STARKs [8]. However, building all of this efficiently for a token application with our requirements has (to our knowledge) never been done (and much less with a formal model and security proof). In order to achieve practical efficiency and compliance with the standards, we needed a number of new innovations, including the following:

- Lattice-based public key encryption schemes, including the NIST-standardized ML-KEM, typically satisfy only a relaxed notion of correctness, allowing for a negligible but non-zero probability of decryption failure. This stands in contrast to most pre-quantum schemes, which guarantee perfect correctness. While this relaxation is intended to optimize performance, it raises questions about the soundness of our proof for ciphertext well-formedness. In particular, standard security models do not typically consider adversaries who intentionally manipulate encryption randomness to generate ciphertexts that are undecryptable.
  Our observation is that ML-KEM already satisfies this stronger notion of *correctness against adversarial encryptor*. This stems from its use of a cryptographic hash function (SHAKE256) to derive encryption randomness from a short random seed. While the standard models SHAKE256 as a pseudorandom function, sufficient for its intended security guarantees, modeling it as a random oracle yields correctness against adversarial encryptors for free. This approach allows us to avoid modifying the parameters of ML-KEM, which would both deviate from the standard and introduce significant inefficiency. See Remark 6 for a detailed discussion.

- Post-Quantum ciphertext are typically much larger than their classical counterparts. The statement i.e. the ML-KEM ciphertext is at least 768 bytes. If one were to naively, pass this as input to the Groth16 proof system, it would require just as many elliptic curve multiplication and addition operations. Each of these costs 6000, and 150 gas respectively,[8] amounting to about 4.7 million gas! When compared to the base line of Groth16 verification, which is about 300,000 gas, this naive solution has an overhead of $15\times$. To minimize this, we leverage the fact that hashing is a much cheaper operation on the EVM. Instead of passing the entire ciphertext as input, we pass a short digest of it as input to the proof system – which now only requires two elliptic curve multiplication and one addition.

- Finally, our implementation is designed to be compatible with the FIPS 203 [43] specification. This results in several direct benefits in terms of usability of the system. First, an auditor can now use *any* ML-KEM implementation, based on compatibility with their existing infrastructure/any internal requirements they have. Furthermore, the auditor's secret key is highly sensitive and can be used to break the privacy of all transactions on chain, and hence Key Management by the auditor is a crucial aspect. Because, ML-KEM is a standardized algorithm it is relatively easy to adopt best practices for Key Management such as only storing it in a secure enclave and/or distributing it amongst multiple non-colluding servers. This is in sharp contrast to an esoteric encryption scheme that is designed to be zk-friendly but hardware-vendors are unlikely to build support for it.

Qurrency Performance We implemented Qurrency as a solidity smart contract and analyzed its performance. Qurrency is extremely competitive relative to other existing, practically used systems. In Section 4, we compare against the baseline of Zeto, a privacy preserving payment system that does not support auditability. Our evaluation reveals that we only incur a moderate 16.4% overhead in gas costs while the proof generation time is under one minute for a transfer involving two input UTXOs and two output UTXOs. We discuss the performance of Qurrency in depth in Section 4.

1.2 Related Practical Work

There are a number of works that have seen practical adoption or experimentation in the digital asset and CBDC spaces; we briefly mention some of those here. In particular, in this section we only consider works that have strong privacy guarantees and can be run on an EVM-compatible chain with minimal changes (so we don't mention privacy-preserving permissioned chains here, for instance). For a full and thorough discussion of all related work (including mostly theoretical works), please see Appendix A.

Zether Zether [16,26] is one of the most well-known EVM-based privacy-preserving platforms and has seen public experimentation in the DREX project (in particular the implementation called Anonymous Zether, which is based on [26]). However, because Zether is account based, it needs to settle all state transfers affecting an account balance to a single encrypted value in the smart contract. This means for a spending transaction to be valid, the sender must use the latest state that is consistent with what the smart contract will use when verifying the proof. This problem exists for any account based privacy tokens that rely on ZKP for transaction validity checks.

[8] `https://www.evm.codes/precompiled`

System	UTXO	EVM	Auditable	Quantum Privacy	Locking	Verifier Gas Cost
Anon. Zether	✗	✓	✗	✗	✓	7188k
Zerocash	✓	✗	✗	✓	✗	N/A[12]
Zeto	✓	✓	✓	✗	✓	1725k
This Work	✓	✓	✓	✓	✓	2009k

Table 1: Comparison against Related Work

To prevent a transaction sent by other users that has incoming transfers to the sender account from modifying the account state, thus preventing the spending transaction from being successfully verified, Zether employs a "spending window" called epochs. Within a spending window, new state transfers are put inside a pending queue, such that account states are not updated. This allows spending transactions to be successfully verified. Then the pending state transfers are applied at the beginning of the next epoch. This solves the problems on spending, but it also limits throughput to be one spending transaction per account per epoch. We don't know of any account-based systems that solve this (seemingly inherent) problem, and this is the rationale for the core reason most practical privacy tokens are based on a UTXO design, including ours: unlike in the account model, UTXO transactions can be fully parallelized, because each UTXO regardless of their ownership can be processed independently.

Zerocash Zerocash [9] is limited due to its lack of smart contract support. The system is designed to support transfers of the zerocoins only, similar to Bitcoin. A regulated digital asset system, such as CBDCs, tokenized deposits or regulated stablecoins, requires programmability in order to support regulatory compliances and different use cases such as Delivery-vs-Payment, Foreign Exchange, etc. The token transfer transactions are subject to various policies enforcement depending on the use case. All this requires the token to be built on a smart contract platform like Ethereum rather than a coins-only system like Zerocash.

Zeto The system which we found to be closest to practically ready for building CBDCs and digital asset platforms on EVM-based blockchains is LFDT Zeto [15]. Zeto is a popular UTXO-based privacy-preserving token toolkit and, when put into the Linux Foundation Decentralized Trust (LFDT) Paladin [14] platform, can be used to quickly and easily build privacy-preserving tokens on any (unmodified) EVM-based chain. It includes multiple implementations that incorporate different policy enforcements including simple mass conservation, and regulatory compliances like privacy-preserving KYC, and non-repudiation.

However, even Zeto had some drawbacks that hindered its adoption. Prior to the enhancements described in this work, it did not provide quantum privacy, and there were no formal models or security proofs for most tokens in the Zeto system.

2 Payment Mechanism

In this section, we describe the various components of a payment mechanism. This includes the setup, user algorithm, smart contract and the correctness and security properties. We build on the payment mechanism proposed in Zether [16], with a few notable differences. First, we consider a UTXO based model in contrast to the account based model consider in Zether - this is reflected most notably in the user algorithms. Next, we require the payment mechanism to be auditable, which necessitates a new security definition along with new auditor specific algorithms. We now present the appropriate payment mechanism by providing a brief overview before describing the various component.

On Ethereum, a payment mechanism is implemented by a smart contract (SC) that maintains a state st. There is an initial setup, and a specified set of user algorithms that let users interact with SC.

As in Zether [16], we will allow UTXOs to be locked to a specified Ethereum address addr such that only transactions originating from addr can spend the corresponding UTXOs. Here a transaction contains some data and ETH, and is signed by the person generating the transaction. These transactions are sent from one Ethereum address to another such that if the destination address has code, the code is executed on the data contained within the transaction.

We next describe the various components of a payment mechanism.

[12] Zerocash is a separate L1.

2.1 Components

Setup. In the setup phase the smart contract (SC) is deployed with public parameters that include an auditor (public) key. These parameters are generated off-chain and then hard coded into the smart contract.

User Algorithms. We denote below the algorithms that allow users to interact with SC. These transactions will further be signed by the user, and this will be used by SC to identify the sender of the transaction. However, for ease of notation we assume that the transactions are appropriately signed before they are sent to SC. Some of these algorithms will also take as input the state st_h of SC at block height h.

1. $\mathsf{pk}, \mathsf{sk} \leftarrow \texttt{CreateKey}(1^\lambda)$: Used to create key pairs. On input the security parameter (in unary), generates a public key pk and secret key sk pair.
2. $\mathsf{tx}_{\mathsf{deposit}} \leftarrow \texttt{CreateDepositTx}(v, \{v_i, \mathsf{pk}_i\}_{i\in[n]})$: Used to deposit funds into one or more UTXOs. On input a value v and the partitioning of v into a set of values $\{v_i, \mathsf{pk}_i\}_i$ (i.e. $\sum_i v_i = v$) , the algorithm creates a transaction that includes n output UTXOs each containing the value v_i, and owned by corresponding public key pk_i.
3. $\mathsf{tx}_{\mathsf{withdraw}} \leftarrow \texttt{CreateWithdrawTx}(\{\mathsf{utxo}_i, v_i, \rho_i\}_{i\in[n]}, v, \mathsf{pk}', \mathsf{rt})$: Used to withdraw funds from one or more UTXOs. On input the UTXOs $\{\mathsf{utxo}_i,\}_{i\in[n]}$ and the corresponding owner information for each UTXO $\{v_i, \rho_i\}_{i\in[n]}$, value v and output public key pk', root rt and state st the algorithm generates a transaction that returns amount v to msg.sender, and creates a new UTXO owned by pk' containing the remaining value.
4. $\mathsf{tx}_{\mathsf{transfer}} \leftarrow \texttt{CreateTransferTx}(\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i\}_{i\in[n]}, \{v'_i, \mathsf{pk}'_i\}_{i\in[n']}, \mathsf{rt}, \mathsf{st})$: Used to transfer amount between UTXOs. On input UTXOs $\{\mathsf{utxo}_i\}_{i\in[n]}$ along with owner information $\{v_i, \rho_i, \mathsf{sk}_i\}_{i\in[n]}$, and recipient information $\{v'_i, \mathsf{pk}'_i\}_{i\in[n']}$, root rt and state st the algorithm generates a transaction that includes output UTXOs for the specified values and public keys.
5. $\mathsf{tx}_{\mathsf{lock}} \leftarrow \texttt{CreateLockTx}(\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i\}_{i\in[n]}, \{v'_i, \mathsf{pk}'_i\}_{i\in[n']}, \{\widetilde{v}_i, \widetilde{\mathsf{pk}}'_i\}_{i\in[\widetilde{n}]}, \mathsf{delegate}, \mathsf{rt}, \mathsf{st})$: Used to consume UTXOs to create both unlocked and locked UTXOs. On input UTXOs $\{\mathsf{utxo}_i\}_{i\in[n]}$ along with owner information $\{v_i, \rho_i, \mathsf{sk}_i\}_{i\in[n]}$, and locked and unlocked recipient information $\{v'_i, \mathsf{pk}'_i\}_{i\in[n']}$ and $\{\widetilde{v}_i, \widetilde{\mathsf{pk}}'_i\}_{i\in[\widetilde{n}]}$ respectively, delegate delegate, root rt and state st the algorithm generates a transaction that includes locked and unlocked UTXOs for the specified values and public keys, with the locked UTXOs delegated to delegate. Locking a transaction $\widetilde{\mathsf{utxo}}$ ensures that only a transaction sent to the smart contract by delegate will be able to spend $\widetilde{\mathsf{utxo}}$.
6. $\mathsf{tx}_{\mathsf{transferLocked}} \leftarrow \texttt{CreateTransferLockedTx}(\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i\}_{i\in[n]}, \{v'_i, \mathsf{pk}'_i\}_{i\in[n']}, \mathsf{rt}, \mathsf{st})$: Used to transfer amount from locked UTXOs to unlocked UTXOs. On input locked UTXOs $\{\mathsf{utxo}_i\}_{i\in[n]}$ along with owner information $\{v_i, \rho_i, \mathsf{sk}_i\}_{i\in[n]}$, and recipient information $\{v'_i, \mathsf{pk}'_i\}_{i\in[n']}$, root rt and state st the algorithm generates a transaction that includes output UTXOs for the specified values and public keys.
7. $\mathsf{tx}_{\mathsf{delegateLock}} \leftarrow \texttt{CreateDelegateLockTx}(\{\mathsf{utxo}_i\}_{i\in[n]}, \mathsf{delegate})$. Used to change the delegate of locked UTXOs. On input locked UTXOs $\{\mathsf{utxo}_i\}_{i\in[n]}$ and delegate delegate the algorithm produces a transaction that changes the delegate of the UTXOs to delegate. Note that this algorithm only produces a valid transaction if it is run by the user to whom $\{\mathsf{utxo}_i\}_{i\in[n]}$ are currently delegated.
8. $b = \mathsf{ReadValue}(\mathsf{st}, \mathsf{utxo}, \rho, \mathsf{pk}, v)$: Used to check if the UTXO matches the inputs provided. On input the current state of SC, the UTXO and auxiliary information, checks if the UTXO is consistent with the auxiliary information.
9. $\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']} \leftarrow \texttt{Audit}(\mathsf{ask}, \mathsf{tx})$: Used to audit the transaction. On input the audit secret key ask and a transaction tx, outputs information about all UTXOs that were both consumed and created during the transaction.
10. $b = \texttt{NullifierCheck}(\mathsf{st}, \mathsf{utxo}, \mathsf{null}, v, \rho, \mathsf{pk}, \mathsf{sk})$: Used to check if the nullifier matches the input provided. On input the current state of SC, the UTXO and the owner information, checks if the UTXO is consistent
11. $b = \mathsf{checkSpend}(\mathsf{st}, \mathsf{utxo}, \mathsf{pk}, \mathsf{sk}, \rho, v)$: Used to check if UTXO has already been spent. On input the state st, UTXO utxo, public key pk, secret key sk, randomness ρ and value v, Outputs 1 if UTXO has already been spent in st.
12. $b = \mathsf{checkUnlock}(\mathsf{st}, \mathsf{utxo})$: Used to check if UTXO is unlocked. On input the state st and UTXO utxo, output 1 if the UTXO is unlocked in st.

Remark 1. We note that, while we present auditing as a single algorithm, one can envision a system where auditing is performed in a decentralized manner to distribute trust. However, this is outside the scope of this work.

Receiving payments. We note that a created UTXO is destined for the owner of the public key used in the generation of the UTXO. However, the transactions do not contain the public key, and a user may not be aware of UTXOs that are sent to them. There are a few ways to deal with this concern, and we discuss a couple of them here. The first is to have some out-of-band mechanism where the person creating the UTXO (sender) informs the owner of the public key (receiver) of the value and salt used in its creation – the user can check this using the ReadValue algorithm. An alternate approach is to have the sender encrypt this information with respect to a receiver's public key[13] and add it to the transaction. Importantly there are no consistency proofs generated for these ciphertexts, the receiver simply decrypts this ciphertext and uses the decrypted information to see if the UTXO can be spent by them. For simplicity we do not make an explicit choice in our description, instead simply saying that the receiver has a mechanism to receive payments. One could require a stronger availability property where in the aforementioned alternative approach, the ciphertext are additionally proved to be consistent with the UTXO. However we do note that if we want quantum privacy, any mechanism that provides a ciphertext must use a post-quantum encryption scheme.

Smart Contract. The internal state of SC is modified by functions provided by the smart contract – `Deposit`, `Withdraw`, `Lock`, `Transfer`, `TransferLocked`, `DelegateLock`. These functions take in as input the corresponding transaction generated via the user algorithm, and either output 0 to indicate that the function failed to process the transaction, or 1 to indicate success and further update the state accordingly.

2.2 Properties of a Payment Mechanism

We refer the reader to Appendix G that describes the ideal functionality $\mathcal{F}_{\text{pq-aud-priv-cash}}$ that formally models and captures the various properties of a Payment Mechanism. Here, we provide a brief informal description of the functionality and describe how the properties are captured by the model.

Ideal Functionality. The ideal functionality interacts with parties enabling a set of functionalities. First, it allows a party to deposit a value v, upon which it generates a coin by sampling a fresh random handle c and returning it to the party. The functionality records the coin together with its value and owner.

To execute a transaction, a party submits a list of coins to spend $c_1, \ldots, c_n$, a list of output values $v'_1, \ldots, v'_{n'}$, and the recipient of each output coin. The functionality retrieves the values of the input coins, deletes them, and creates new coins $c'_1, \ldots, c'_{n'}$ by sampling fresh random handles. If a coin c'_j is assigned to party $\mathcal{P}$, the functionality delivers to $\mathcal{P}$ both c'_j and its associated value v'_j.

The checks performed by the ideal functionality ensure the following properties:

Completeness: Every valid transaction is processed, consuming the input coins and creating the specified output coins.

Balance: The sum of the input coin values equals the sum of the output coin values.

Auditability: The functionality reports the details of every transaction to an auditor.

Privacy: Each coin is represented by a fresh random handle known only to its owner (and the transaction creator), hiding transaction linkage from all other parties.

Soundness: Only the owner of a coin, who holds its random handle c, may spend it.

While this description captures the core behavior of the ideal functionality, our formal model also accounts for additional features such as coin locking where only a specified party (except for the owner) can spend the coin, as well as capturing situations where an adversarial party may creates coins intended for honest parties, but never send them information necessary to spend them.

[13] For security, each user would have a pair of public keys, where one of the keys would be used exclusively for this purpose.

3 Qurrency Payment Mechanism

We now present the Qurrency payment mechanism. As outlined in Section 2, the description involves the setup algorithm, the smart contract, and the user algorithms, which we describe below. The proofs of the payment mechanism properties are provided in Appendix H.

We briefly describe how we are able to achieve auditability for our payment mechanism. For each transaction (except for delegating lock), we add a ciphertext encrypting all the private information required to generate the transaction. However, to ensure auditability, one needs to include a proof demonstrating that this ciphertext is computed in an honest manner.

3.1 Setup

The setup initializes the non-interactive zero knowledge (NIZK) parameters, intializes the data structures. It allows the auditor to input an auditor public key. It then deploys the smart contract with these parameters. We describe the data structures and NIZK relations below. See the formal specification in Algorithm 27.

Algorithm 1 `Setup`

Input: Security parameter λ (in unary) and auditor public key apk

1: $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^\lambda$.
2: $\mathsf{NIZK.pp} \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$.
3: Initialize $\mathsf{CommTree} \leftarrow \mathsf{SMT.Setup}(1^\lambda)$, $\mathsf{LockedCommTree} \leftarrow \mathsf{SMT.Setup}(1^\lambda)$ and table $\mathsf{Nullifiers} : \{0,1\}^\lambda \mapsto \{0,1\}$.
4: Deploy a smart contract with parameters H, $\mathsf{NIZK.pp}$, $\mathsf{CommTree}$, $\mathsf{LockedCommTree}$, $\mathsf{Nullifiers}$, apk.

Data Structures Two Sparse Merkle Trees (SMT) (Appendix C.3), CommTree and LockedCommTree, are maintained by the smart contract. These trees keep track of the unlocked and locked UTXOs respectively. While CommTree will be an append only tree, LockedCommTree which in addition to appends, will also allow changing the value of leaf nodes in the tree. But neither of these trees will allow for nodes to be deleted. Since there is no public mechanism to mark a UTXO as spent, a table Nullifiers is maintained to track this status via a (computationally) injective mapping between each nullifier and its corresponding UTXO. This mapping is known only to the UTXO's owner.

NIZK Relations We define a base language $\mathcal{L}$ in Figure 1, such that all other languages can be cast as a special case of this relation. At a high level, $\mathcal{L}$ ensures that the transaction is well formed. In particular, the statement is specified by input and output values v_{in} and v_{out}, ciphertext ct, a Sparse Merkle Tree (SMT) root rt, a delegate $\mathsf{delegate}$, input nullifiers $\{\mathsf{null}_i\}_i$, output UTXOs $\{\mathsf{utxo}'_i\}_i$, and locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_i$). We note that some parts of the statement are optional: the root and delegate and may be denoted as as $\perp$, and an empty list for the nullifier, or either list of output UTXOs may be denoted as ϕ. The language checks the following, given as witness: the encryption randomness R; information regarding the input UTXOs $\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i\}_i$; information regarding the output UTXOs $\{v'_i, \rho'_i, \mathsf{pk}'_i\}_i$; and information regarding the locked UTXOs $\{\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}}_i\}_i$:

Well-formedness of output UTXOs: The output and locked output UTXOs are computed correctly as $\mathsf{utxo}'_i = \mathsf{H}(v'_i, \rho'_i, \mathsf{pk}'_i)$ and $\widetilde{\mathsf{utxo}}_i = \mathsf{H}(\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}}_i)$, respectively. Further, the values v'_i and $\widetilde{v}_i$ should be positive, and within the specified range.

Ownership of input UTXO: First, it checks whether the information regarding the input UTXOs is consistent, i.e., that $\mathsf{utxo}_i = \mathsf{H}(v_i, \rho_i, \mathsf{pk}_i)$ and that $(\mathsf{sk}_i, \mathsf{pk}_i)$ form a valid secret-key/public-key pair. Next, it verifies that the input UTXO corresponds to the nullifier in the statement, i.e., $\mathsf{null}_i = \mathsf{H}(v_i, \rho_i, \mathsf{sk}_i)$. Finally, it checks whether utxo_i exists in the Sparse Merkle Tree rooted at rt—this is verified using the Merkle proof τ_i. Depending on whether the input transactions are locked or unlocked (i.e., depending on whether $\mathsf{delegate} = \perp$), the delegate $\mathsf{delegate}$ is used to perform the check in the SMT.

Value Transfer: The values exchanged are conserved, i.e. $v_{\mathsf{in}} + \sum_i v_i = v_{\mathsf{out}} + \sum_i v'_i + \sum_i \widetilde{v}_i$.

Well-formedness of ciphertexts: The ciphertext using randomness R encrypts the rest of the witness.

Language $\mathcal{L}$

Statement: $(v_{\mathsf{in}}, v_{\mathsf{out}}, \mathsf{ct}, \mathsf{rt}, \mathsf{delegate}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]})$

Witness:

- $m \coloneqq (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \{\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}}_i\}_{i\in[\widetilde{n}]})$
- R

Relation:

- $v_{\mathsf{in}} + \sum_{i\in[n]} v_i = v_{\mathsf{out}} + \sum_{i\in[n']} v'_i + \sum_{i\in[\widetilde{n}]} \widetilde{v}_i$
- $\mathsf{ct} = \mathsf{Enc}(\mathsf{apk}, m; R)$
- For all $i \in [n]$:
  1. $(\mathsf{pk}_i, \mathsf{sk}_i) \in \mathsf{keyGen}$
  2. $\mathsf{utxo}_i = \mathsf{H}(v_i, \rho_i, \mathsf{pk}_i) \wedge \mathsf{null}_i = \mathsf{H}(v_i, \rho_i, \mathsf{sk}_i)$
  3. If $\mathsf{delegate} = \perp$:
     - $\mathsf{SMT.Verify}(\mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i), \tau_i) = 1$

     Else:
     - $\mathsf{SMT.Verify}(\mathsf{rt}, (\mathsf{utxo}_i, \mathsf{delegate}), \tau_i) = 1$
- For all $i \in [n']$:
  1. $v'_i \in [0, \mathsf{maxValue}]$
  2. $\mathsf{utxo}'_i = \mathsf{H}(v'_i, \rho'_i, \mathsf{pk}'_i)$
- For all $i \in [\widetilde{n}]$:
  1. $\widetilde{v}_i \in [0, \mathsf{maxValue}]$
  2. $\widetilde{\mathsf{utxo}}_i = \mathsf{H}(\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}}_i)$

Fig. 1: Language $\mathcal{L}$

Remark 2. We assume here that it is efficiently checkable that a pair $(\mathsf{sk}, \mathsf{pk})$ are the output of the key generation algorithm. In practice, we consider schemes where pk can be derived deterministically given sk.

The language $\mathcal{L}$ is defined with respect to the audit public key apk, and thus is specified after the apk is sampled. The languages for the various transactions, cast as special instances of $\mathcal{L}$, are specified in Appendix D

3.2 Smart Contract

We now describe the smart contract underlying Qurrency. In describing the smart contract, we use Solidity syntax in several places and provide an explanation of the syntax below. The following provides a high-level overview of the Qurrency smart contract. We defer the reader to the formal specification for further details in Appendix E. Each of the smart contracts first validates the inputs to the transaction, verifies proofs, and then potentially updates the state.

`Validate`: This is a helper function that on input nullifier $\{\mathsf{null}_i\}_i$, output UTXOs $\{\mathsf{utxo}'_i\}_i$ and locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_i$ validates the proposed transaction by making sure that the output UTXOs are not being reused, and the input nullifiers have not already been marked as spent.

`UpdateState`: This is a helper function that on input nullifier $\{\mathsf{null}_i\}_i$, output UTXOs $\{\mathsf{utxo}'_i\}_i$, locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_i$ and delegate $\mathsf{delegate}$ updates the internal state to mark the nullifiers as spent and adds $\{\mathsf{utxo}'_i\}_i$ to $\mathsf{CommTree}$ and $\{\widetilde{\mathsf{utxo}}_i\}_i$ to $\mathsf{LockedCommTree}$.

`Deposit`: Allows the user to deposit value and split into multiple UTXOs. On input amount v, output UTXOs $\{\mathsf{utxo}'_i\}_i$, proof π and ciphertext ct, `Deposit` uses `Validate` to check the validity of the transaction, then it verifies the proof π and then updates the state by calling `UpdateState`.

`Withdraw`: Allows a user to withdraw value from the UTXOs. On input the nullifiers $\{\mathsf{null}_i\}_i$, value v, output UTXO utxo', proof π, root rt and ciphertext ct, `Withdraw` uses `Validate` to check the validity of the transaction, then it verifies the proof π, it then checks that rt was among the list of roots for $\mathsf{CommTree}$ and then updates the state by calling `UpdateState`. Checking if rt was among the list of prior roots of $\mathsf{CommTree}$ is important as the rt may have otherwise changed since the transaction was created. This prevents the transaction from simply being rejected if this were the case.

`Transfer`: Allows users to transfer value between UTXOs. On input the nullifiers $\{\mathsf{null}_i\}_i$, value v, output UTXOs $\{\mathsf{utxo}'_i\}_i$, proof π, root rt and ciphertext ct, `Transfer` uses `Validate` to check the validity of the transaction, then it verifies the proof π, it then checks that rt was among the list of roots for CommTree and then updates the state by calling `UpdateState`.

`TransferLocked`: Allows users to transfer value between UTXOs where the input UTXOs are locked. On input the nullifiers $\{\mathsf{null}_i\}_i$, value v, output UTXOs $\{\mathsf{utxo}'_i\}_i$, proof π, root rt and ciphertext ct, `TransferLocked` uses `Validate` to check the validity of the transaction, then it verifies the proof π, it then checks that rt was among the list of roots for LockedCommTree and then updates the state by calling `UpdateState`. Note that here the NIZK verification uses msg.sender (sender of the message) to ensure that the person sending the transaction is allowed to spend the input UTXOs corresponding to the nullifiers.

`Lock`: Allows users to spend UTXOs to create output UTXOs, some of which are locked to a specified delegate. On input the nullifiers $\{\mathsf{null}_i\}_i$, value v, output UTXOs $\{\mathsf{utxo}'_i\}_i$, locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_i$ proof π, root rt, delegate $\mathsf{delegate}$ and ciphertext ct, `Lock` uses `Validate` to check the validity of the transaction, then it verifies the proof π, it then checks that rt was among the list of roots for CommTree and then updates the state by calling `UpdateState` with $\mathsf{delegate}$ to make sure $\{\widetilde{\mathsf{utxo}}_i\}_i$ are stored in LockedCommTree with delegate $\mathsf{delegate}$.

`DelegateLock`: Allows users to delegate the lock of one or more of their locked UTXOs to a different delegate. On input locked UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_i$ and delegate $\mathsf{delegate}$, `DelegateLock` checks if the sender is the owner of the locked UTXOs, and if so updates the corresponding leaves in LockedCommTree to change the delegate to $\mathsf{delegate}$.

3.3 User Algorithms

Here, we describe the user algorithms for the Qurrency payment mechanism.

User Algorithm: `CreateKey`

1: INPUTS: 1^λ
2: OUTPUTS: $\mathsf{sk} \in \mathbb{Z}_p, \mathsf{pk} \in \mathbb{G}$
3: $\mathsf{sk} \leftarrow\!\$\ \mathbb{Z}_p$ and $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$

User Algorithm: `CreateDepositTx`

1: INPUTS: amount v in ETH, recipient amounts and owners $\{v'_i, \mathsf{pk}'_i\}_{i\in[n']}$
2: OUTPUTS: $\mathsf{tx}_{\mathsf{deposit}} = (v, \{\mathsf{utxo}'_i\}_{i\in[n]}, \mathsf{ct}, \pi_{\mathsf{deposit}})$, salts $\{\rho'_i\}_{i\in[n']}$
3: sample salts $\{\rho'_i \leftarrow\!\$\ \mathbb{Z}_q\}_{i\in[n']}$
4: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
5: set $m = (\phi, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \phi)$
6: sample $R \leftarrow\!\$\ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
7: prove $\pi_{\mathsf{deposit}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{deposit}})$:
   – sta : $(\mathsf{ct}, v, \{\mathsf{utxo}'_i\}_{i\in[n']})$
   – wit : (R, m)

User Algorithm: `CreateTransferTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, recipient amounts and owners $\{(v'_i, \mathsf{pk}'_i)\}_{i\in[n']}$, root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{transfer}} = (\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \mathsf{rt}, \mathsf{ct}, \pi_{\mathsf{transfer}})$, salts $\{\rho'_i\}_{i\in[n']}$
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{CommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i)\}_{i\in[n]}$
4: sample salts $\{\rho'_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[n']}$
5: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
6: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \phi)$
7: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
8: prove $\pi_{\mathsf{transfer}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{transfer}})$:
- $\mathsf{sta} : (\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']})$
- $\mathsf{wit} : (R, m)$

User Algorithm: `CreateWithdrawTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, withdrawal amount v in ETH, owner of balance amount pk', root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{withdraw}} = (v, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}', \mathsf{rt}, \mathsf{ct}, \pi_{\mathsf{withdraw}})$, salt ρ'
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{CommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i)\}_{i\in[n]}$
4: if $\sum_{i\in[n]} v_i > v$:
- compute $v' = \sum_{i\in[n]} v_i - v$
- sample $\rho' \leftarrow\$ \mathbb{Z}_q$
- compute $\mathsf{utxo}' \leftarrow H(v', \rho', \mathsf{pk}')\}_{i\in[n]}$

5: else $\mathsf{utxo}' \leftarrow \perp$, $v' \leftarrow \perp$ and $\rho' \leftarrow \perp$.
6: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, (v', \rho', \mathsf{pk}'), \phi)$
7: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
8: prove $\pi_{\mathsf{withdraw}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{withdraw}})$:
- $\mathsf{sta} : (\mathsf{ct}, v, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}')$
- $\mathsf{wit} : (R, m)$

User Algorithm: `CreateLockTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, recipient amounts and owners $\{(v'_i, \mathsf{pk}'_i)\}_{i\in[n']}$, locked recipient amounts and owners $\{(\widetilde{v}_i, \widetilde{\mathsf{pk}}_i)\}_{i\in[n']}$, delegate address $\mathsf{ethAddr}$ root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{lock}} = (\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}, \mathsf{rt}, \mathsf{ct}, \mathsf{ethAddr}, \pi_{\mathsf{lock}})$, salts $\{\rho'_i\}_{i\in[n']}$ and $\{\widetilde{\rho}_i\}_{i\in[\widetilde{n}]}$
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{CommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i)\}_{i\in[n]}$
4: sample salts $\{\rho'_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[n']}$ and $\{\widetilde{\rho}_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[\widetilde{n}]}$
5: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
6: compute $\{\widetilde{\mathsf{utxo}}_i \leftarrow H(\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}}_i)\}_{i\in[\widetilde{n}]}$
7: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \{\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}}_i\}_{i\in[\widetilde{n}]})$
8: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
9: prove $\pi_{\mathsf{lock}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{lock}})$:
- $\mathsf{sta} : (\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]})$
- $\mathsf{wit} : (R, m)$

3.4 Instantiation

We instantiate Qurrency with Groth16 [36] for NIZK, Poseidon [35] for cryptographic hash function, and *IND-CPA* version of ML-KEM [43,11] (Appendix B) for underlying PKE. Refer to Appendix B.1 for discussions on the choice of ML-KEM and a justification of why the IND-CPA version of ML-KEM suffices when instantiating Qurrency.

User Algorithm: `CreateTransferLockedTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, recipient amounts and owners $\{(v'_i, \mathsf{pk}'_i)\}_{i\in[n']}$, root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{transferLocked}} = (\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \mathsf{rt}, \mathsf{ct}, \pi_{\mathsf{lock}})$, salts $\{\rho'_i\}_{i\in[n']}$
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{LockedCommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{msg.sender})\}_{i\in[n]}$
4: sample salts $\{\rho'_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[n']}$
5: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
6: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \phi)$
7: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
8: prove $\pi_{\mathsf{transfer}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{transferLocked}})$:
   – sta : $(\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']})$
   – wit : (R, m)

User Algorithm: `CreateDelegateLockTx`

1: INPUTS: input UTXO $\{(\mathsf{utxo}_i)\}_{i\in[n]}$, delegate ethAddr, state st of SC
2: OUTPUT: $\mathsf{tx}_{\mathsf{delegateLock}} = (v, \{\mathsf{utxo}\}_{i\in[n]}, \mathsf{delegate})$

User Algorithm: `ReadValue`

1: INPUTS: UTXO utxo, salt ρ, amount v in ETH, public key pk
2: OUTPUT: $b = 1$ if $\mathsf{utxo} = H(v, \rho, \mathsf{pk})$, else output $b = 0$.

User Algorithm: `Audit`

1: INPUTS: auditor secret key ask, transaction tx
2: OUTPUT: $\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']} = \mathsf{Dec}(\mathsf{ask}, \mathsf{ct})$ where ct is contained in tx.

User Algorithm: `NullifierCheck`

1: INPUTS: UTXO utxo, salt ρ, amount v in ETH, public key pk, secret key sk, nullifier null
2: OUTPUT: $b = 1$ if $\mathsf{utxo} = H(v, \rho, \mathsf{pk})$ AND $\mathsf{null} = H(v, \rho, \mathsf{sk})$, else output $b = 0$.

User Algorithm: `checkSpend`

1: INPUTS: UTXO utxo, salt ρ, amount v in ETH, public key pk, secret key sk, nullifier null, state st of SC
2: OUTPUT: $b = 1$ if $\mathsf{utxo} = H(v, \rho, \mathsf{pk})$ AND $\mathsf{null} = H(v, \rho, \mathsf{sk})$, else output $b = 0$.

User Algorithm: `checkUnlock`

1: INPUTS: UTXO utxo, state st of SC
2: OUTPUT: $b = 1$ if $\mathsf{inTree}(\mathsf{CommTree}, (\mathsf{utxo}, \mathsf{utxo})) = 1$, else output $b = 0$.

4 Implementation and Evaluation

We implemented the Qurrency payment mechanism using Zeto [15] as a starting point and extended it to incorporate auditability. All circuits required for the zero-knowledge proof are written in circom [7], a domain-specific language for implementing zero-knowledge proofs. We also implement the Ethereum smart contract (described in Section 3) for the Qurrency payment mechanism to obtain estimates on gas costs incurred by various operations. Our implementation was guided by two main requirements:

1. All ciphertexts generated in the scheme must be compliant with the FIPS 203 ML-KEM standard [43]. We make this choice despite the fact that ML-KEM is not "zk-friendly" for a few important reasons:
   - Our implementation is compatible with *any* ML-KEM implementation. In particular, this means that an auditor can now use *any* ML-KEM implementation, based on convenience/any internal requirements they have.
   - Given that the auditor's secret key can be used to break the privacy of all transactions on chain, key management is a critical aspect. From a best practices perspective, the key should be stored in a secure enclave and/or distributed amongst multiple servers such that an adversary corrupting some threshold number of them will not able to break privacy of transactions. If we design a new encryption scheme, then it is unlikely that hardware-vendors will build support for it. It also allows auditors to directly use any solutions for threshold decryption of ML-KEM ciphertexts.[14]
2. **Minimizing gas costs paid per transaction.** We highlight that the Qurrency system only introduces a minimal increase in gas costs (+16.4%) compared to Zeto without auditability.

	#(constraints)	**Prover Time** (Witness+Proof)	**Gas Cost**
Zeto	41150	0.7 + 0.75 sec	1.7 m
Qurrency	1676861	14.6 + 6.5 sec	2 m

Table 2: Comparison Against Prior Work.

In Table 2 we compare the performance of a Qurrency transfer involving two input UTXOs and two output UTXOs against the baseline of a Zeto transfer which does not provide auditability. As expected, we incur a moderate overhead (+16.4%) in gas cost which is mainly due to the storage cost of the large ML-KEM ciphertext (768 bytes). As expected, the number of constraints and the prover time is larger but this is expected as Zeto was designed to prove zk-friendly statements involving the Poseidon hash function and Elliptic Curve operations over the Babyjubjub curve [15] which forms a cycle with the BN254 curve used in the Groth16 proof system. However, Qurrency proves an inherently expensive ML-KEM encryption circuit as it aims to be compliant with standardized post-quantum encryption schemes. We emphasize that this is a proof of concept and that there is room for optimization in the number of constraints, prover times, and gas costs.

Qurrency is integrated within the Zeto repository maintained by Linux Foundation Decentralized Trust. Our full implementation and benchmarks are available at the following GitHub repository: `https://github.com/LFDT-Paladin/zeto`.

Acknowledgements

The work of second, fourth and seventh authors is supported in part by the AFOSR Award FA9550-24-1-0156, and research grants from the Bakar Fund, the Sui Foundation, the Stellar Development Foundation, Supra Inc., and Byzantine Research Inc. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these institutes.

We would like to thank our anonymous reviewers for their help in improving this paper.

[14] We note that ML-KEM can be thresholdized using general-purpose MPC, with concrete optimizations left for future work. Building on recent progress in thresholdizing ML-DSA using general-purpose MPC [10], we believe that practical performance is within reach.

[15] `https://eips.ethereum.org/EIPS/eip-2494`

References

1. Elli Androulaki, Marcus Brandenburger, Angelo De Caro, Kaoutar Elkhiyaoui, Alexandros Filios, Liran Funaro, Yacov Manevich, Senthilnathan Natarajan, and Manish Sethi. A framework for resilient, transparent, high-throughput, privacy-enabled central bank digital currencies. Cryptology ePrint Archive, Paper 2023/1717, 2023.
2. Elli Androulaki, Jan Camenisch, Angelo Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. Privacy-preserving auditable token payments in a permissioned blockchain system. In *AFT 20: 2nd ACM Conference on Advances in Financial Technologies*, pages 255–267, 10 2020.
3. Nicholas Anthony. Cbdc spells doom for financial privacy, 2024.
4. European Central Bank and Bank of Japan. Project stella: Balancing confidentiality and auditability in a distributed ledger environment, feb 2020.
5. Ashley Belanger. America's slow-moving, confused crypto regulation is driving industry out of us. Ars Technica, nov 2022.
6. Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111. Springer, Berlin, Heidelberg, May 1995.
7. Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 20(6):4733–4751, 2023.
8. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.
9. 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. IEEE Computer Society Press, May 2014.
10. Alexander Bienstock, Leo de Castro, Daniel Escudero, Antigoni Polychroniadou, and Akira Takahashi. Efficient, scalable threshold ML-DSA signatures: An MPC approach. Cryptology ePrint Archive, Paper 2025/1163, 2025.
11. Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy*, pages 353–367. IEEE Computer Society Press, April 2018.
12. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.
13. Banco Central Do Brasil. Piloto drex relatório fase 1, feb 2025.
14. Peter Broadhurst, Roger Brogan, David Echelberger, Alexander Herranz, John Hosie, Gabriel Indik, Enrique Lacal, Samuel May, Anna McAllister, Francesco Miliani, Hayeden Onelapahead, Andrew Richardson, Alex Shorsher, David Wertenteil, Matt Whitehead, Alex Wood, Chengxuan Xing, and Jim Zhang. Hyperledger paladin. = https://github.com/LF-Decentralized-Trust-labs/paladin, 2025.
15. Peter Broadhurst, Roger Brogan, Alexander Herranz, Enrique Lacal, Andrew Richardson, Chengxuan Xing, and Jim Zhang. Hyperledger zeto. = https://github.com/hyperledger-labs/zeto, 2025.
16. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 423–443. Springer, Cham, February 2020.
17. Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed E. Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via PVORM. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 701–717. ACM Press, October / November 2017.
18. BIS Innovation Hub Eurosystem Centre, Bank of France, and Deutsche Bundesbank. Project leap: Quantum-proofing the financial system, jun 2023.
19. Yu Chen, Xuecheng Ma, Cong Tang, and Man Ho Au. PGC: Decentralized confidential payment system with auditability. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part I*, volume 12308 of *LNCS*, pages 591–610. Springer, Cham, September 2020.
20. Paula da Cunha Duarte. Enhancing financial inclusion in developing countries: the potential of drex, the brazilian cbdc, oct 2024.
21. Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees - caching strategies and secure (non-)membership proofs. In Billy Bob Brumley and Juha Röning, editors, *Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings*, volume 10014 of *Lecture Notes in Computer Science*, pages 199–215, 2016.
22. Maarten G.A. Daman. Making the digital euro truly private, jun 2024.
23. Daniel de Haro Moraes, Joao Paulo Aragao Pereira, Bruno Estolano Grossi, Gustavo Mirapalheta, George Marcel Monteiro Arcuri Smetana, Wesley Rodrigues, Courtnay Nery Guimarães Jr., Bruno Domingues, Fábio Saito, and Marcos Simplício. Applying post-quantum cryptography algorithms to a DLT-based CBDC infrastructure: Comparative and feasibility analysis. Cryptology ePrint Archive, Paper 2024/1206, 2024.
24. DeFiLlama. Total value locked in defi, apr 2025.
25. Deloitte. Central bank digital currencies: The next disruptor, sep 2021.
26. Benjamin E. Diamond. Many-out-of-many proofs and applications to anonymous zether. In *2021 IEEE Symposium on Security and Privacy*, pages 1800–1817. IEEE Computer Society Press, May 2021.

27. Evan Duffield and Daniel Diaz. Dash: A payments-focused cryptocurrency. Github, dec 2024.
28. Tom Emmer. Congressman emmer announces stakeholder support for the anti-cbdc surveillance state act, mar 2025.
29. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 649–678. Springer, Cham, December 2019.
30. FFIEC. Bsa/aml manual, appendix p: Bsa record retention requirements, 2025.
31. Digital Pound Foundation. Brazil's drex cbdc pilot enters second phase with focus on privacy and interoperability, oct 2024.
32. Ethereum Foundation. Future-proofing ethereum, 2025.
33. Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In Jens Grossklags and Bart Preneel, editors, *FC 2016*, volume 9603 of *LNCS*, pages 81–98. Springer, Berlin, Heidelberg, February 2016.
34. Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. A security framework for distributed ledgers. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1043–1064. ACM Press, November 2021.
35. Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021.
36. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016.
37. Zhangshuang Guan, Zhiguo Wan, Yang Yang, Yan Zhou, and Butian Huang. Blockmaze: An efficient privacy-preserving account-model blockchain based on zk-snarks. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1446–1463, 2020.
38. Ian Hall. Brazil's cbdc strives to overcome data-privacy 'obstacle' as pilot tests evolve, jun 2024.
39. Jonathan Herzog, Moses Liskov, and Silvio Micali. Plaintext awareness via key registration. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 548–564. Springer, Berlin, Heidelberg, August 2003.
40. Daira-Emma Hopwood, Nathan Wilcox, Jack Grigg, Sean Bowe, and Kris Nuttycombe. Zip 316: Unified addresses and unified viewing keys.
41. HQC — pqc-hqc.org. `https://pqc-hqc.org/index.html`. [Accessed 21-04-2025].
42. Lars Hupel and Makan Rafiee. How does post-quantum cryptography affect central bank digital currency?, 2023.
43. National Institute of Standards Information Technology Laboratory and Technology. Fips 203: Module-lattice-basedkey-encapsulation mechanism standard. `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf`, aug 2024.
44. Ledger Insights. Brazil's drex wholesale cbdc pilot has been 'challenging', feb 2025.
45. Gweonho Jeong, Nuri Lee, Jihye Kim, and Hyunok Oh. Azeroth: Auditable zero-knowledge transactions in smart contracts. *IEEE Access*, 11:56463–56480, 2023.
46. Hui Kang, Ting Dai, Nerla Jean-Louis, Shu Tao, and Xiaohui Gu. Fabzk: Supporting privacy-preserving, auditable smart contracts in hyperledger fabric. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 543–555. IEEE, 2019.
47. Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016.
48. Tianjian Liu, Dawei Zhang, Wei Wang, and Chang Chen. ASOZ: a decentralized payment system with privacy preserving and auditing on public blockchain. Cryptology ePrint Archive, Report 2023/1816, 2023.
49. James Lovejoy, Tarakaram Gollamudi, Jeremy Kassis, Narayanan Pillai, Jeremy Brotherton, and Eric Thompson. Heraclius: A byzantine fault tolerant database system with potential for modern payments systems, feb 2025.
50. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Berlin, Heidelberg, August 1988.
51. Ivan Miloševic and Katarina Savic. Blockchain and personal data protection, 2023.
52. Moneropedia. Moneropedia: View key.
53. Neha Narula, Willy Vasquez, and Madars Virza. zkLedger:Privacy-Preserving auditing for distributed ledgers. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 65–80, 2018.
54. Cameron Nili, Tom Patterson, and Carl Dukatz. Safeguarding central bank digital currency systems in the post-quantum computing age, may 2024.
55. NIST Selects HQC as Fifth Algorithm for Post-Quantum Encryption — nist.gov. `https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption`. [Accessed 21-04-2025].
56. Post-Quantum Cryptography | CSRC | CSRC — csrc.nist.gov. `https://csrc.nist.gov/projects/post-quantum-cryptography`. [Accessed 21-04-2025].
57. Bank of Canada, Swiss National Bank, European Central Bank, Bank of England, Bank of Japan, Board of Governors Federal Reserve System, Sveriges Riksbank, and Bank for International Settlements. Central bank digital currencies: System design, nov 2024.

58. Code of Federal Regulations. Title 31: Money and finance:treasury parts 500 to end, jul 2013.
59. Board of Governors of the Federal Reserve System. Central bank digital currency (cbdc): Frequently asked questions, apr 2023.
60. Rep. Michael G. Oxley. H.r.3763 - sarbanes-oxley act of 2002, 2002.
61. Parfin. Rayls selected to participate in kinexys by j.p. morgan's project epic. https://parfin.io/en/blog/rayls-selected-to-participate-in-kinexys-by-j-p-morgan-s-project-epic, jul 2024.
62. Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4, dec 2019.
63. Silvio Petriconi. Invited talk: What would it take to operationalize utxo-based settlement for central bank digital currency?, 2025.
64. Alexandra Prager, Keerthi Moudgal, and Nikhil Sharma. Project epic: Fueling tokenized finance with on-chain enterprise privacy, identity, and composability, 2024.
65. Rayls. Drex, the future of a tokenised economy in brazil, leveraging rayls privacy solution, feb 2025.
66. DTCC Press Releases. Dtcc announces new platform for tokenized real-time collateral management, apr 2025.
67. Antoine Rondelet and Michal Zajac. ZETH: on integrating zerocash on ethereum. *CoRR*, abs/1904.00905, 2019.
68. Aaron Stanley. Brazil abandons blockchain for its drex cbdc project. Forbes Digital Assets, aug 2025.
69. European Union. Proposal for a regulation of the european parliament and of the council on the establishment of the digital euro, jun 2023.
70. Chairman French Hill United States House Committee on Financial Services. House passes cbdc anti-surveillance state act, may 2024.
71. Nicolas van Saberhagen. Cryptonote v 2.0. Github, oct 2013.
72. Yi-Ran Wang, Chao-Qun Ma, and Yi-Shuai Ren. A model for cbdc audits based on blockchain technology: Learning from the dcep. *Research in International Business and Finance*, 63:101781, 2022.

A Related Work

Systems for privacy and confidentiality on blockchains have been built for more than 15 years at this point. Cryptocurrencies like Monero [71], Zerocash [9], and Dash [27] were some of the most notable first systems that used powerful cryptography for privacy-preserving payments. Early private smart contract functionality as developed in Hawk [47], and Tornadocash [62] caused private payment systems to gain attention even in the mainstream media. However, more recently there have been many tools built for privacy-preserving payments or tokens on blockchains, and we mention many of them here. We focus on systems that have either open source implementations with documentation or academic publications (or preferably, both); we note that there are many implementations that have gotten attention in the media that do not have either of these [61] that we therefore do not consider here.

Permissioned Blockchains Perhaps the easiest way to get extremely efficient privacy-preserving payment systems for blockchain is to use a permissioned blockchain, or some other customization that requires changes to a blockchain's core. While these systems certainly have merit, they are not generally extensible past the blockchain in which they are constructed, and thus are not useful for many real-world applications, including ours, as most digital assets currently reside on EVM-based chains [24].

ZkLedger [53] is a custom blockchain with good privacy and auditing guarantees, as well as excellent performance but is not post-quantum, does not have a formal model, and is not extensible to EVM-based blockchains. Solidus [17] cleverly utilizes ORAM to build efficient payments in a "bank-to-bank" model, but there is no explicit auditing mechanism, the system is not quantum-secure, and their model is not suitable for public chains as it is focused specifically on their "bank-to-bank" vision.

There are many systems that have been built on or for Hyperledger Fabric that are aimed at confidential tokens. In [2], the authors build a privacy token with an audit functionality that is explicitly for permissioned blockchains (including and mainly Hyperledger Fabric). Their definition of auditability is not too dissimilar from ours, but only works in a permissioned setting, and their modeling omits some necessary functionality in a more public setting (e.g. UTXO locking), although it could feasibly be added without too much difficulty. This protocol also requires consensus node participation, which may not be practical even in some permissioned settings, and is not PQ-secure. In FabZK [46], the authors use Pedersen commitments to build confidential transactions in a relatively standard way. However, their relatively simple construction is not PQ-secure, is specific to Hyperledger Fabric, and is light on formalization. Finally, the authors of [1] modified Hyperledger Fabric to build an extremely efficient construction of a platform for confidential digital assets. While the paper mentions that auditability can be done, it does not implement or address it. This construction is also not PQ-secure and requires use of a specialized (and not open source) version of Hyperledger Fabric.

Zerocash-based Works There have been many papers that have built upon the seminal Zerocash construction [9]. In [33], the authors build an anonymous cryptocurrency based on Zcash. Their construction allows the blockchain to enforce extra policies on transactions through more general ZKPs, but not full auditing in the sense that we require. It also lacks PQ-security. Zeth [67] essentially implements Zerocash directly as an Ethereum contract. While it has many nice features, it lacks auditability and PQ-security.

ASOZ [48] builds directly on zerocash to add auditability. They add a ciphertext to encrypt various aspects of a transfer under an audit public key. They choose an encryption scheme that allows the use of custom SNARKs (essentially sigma protocols) to prove that these were computed in a consistent manner. However, their scheme does not have quantum privacy (and their custom tools make this very hard to achieve) and lacks extra useful features like lockability that are essential in our use cases. Moreover, their scheme has a substantial drawback that unfortunately rules it out as a candidate for our use cases: the auditor must do a brute-force discrete log computation to find the encrypted value of a transaction; this means that auditing becomes extremely inefficient for all but the simplest use cases, and particularly if the auditing key is thresholdized. We note that Zether [16] requires such a brute-force computation of the *users*, which is a reasonable requirement, as the users will typically have some idea as to how much digital currency they should expect in their transactions, but this is much less reasonable when an auditor needs to examine many transactions.

Zether-based Works Zether [16] was a large breakthrough in confidential transactions on Ethereum and has inspired many follow-up works, some of which are relevant to us here. Unfortunately for us, Zether uses an account model. This enables many useful features for general-purpose computations on blockchain, but is significantly less efficient than a UTXO model construction for most payment, token, and asset systems, including ours. In [26], the author builds a more efficient version of anonymous Zether, although the construction is still in the account model, lacks explicit auditability, and is not PQ-secure.

Azeroth[45] is a scheme based on Zether that adds auditability in a way similar to how we achieve it, by using a ciphertext coupled with a Zerocash-like scheme and a proof of equivalence between the appropriate values. Their solution is still in the account based model, although they use Merkle proofs for privacy in a manner similar to Zerocash. Everything is encrypted to an auditor public key, and the smart contract keeps all encrypted accounts. When a transfer is made, the party initiating the transfer asks the contract for its encrypted account information, updates it in the clear and then re-encrypts it to send back to the contract. It then proves using a SNARK that this was done correctly, and the value encrypted to the auditor is consistent. However, Azeroth does not have quantum privacy and lacks functionality necessary for real-world use like locking, making it unfortunately not suitable for our use cases. We note that Azeroth does achieve performance comparable to (and even slightly better than) Zeto.

Viewing Keys Viewing keys, as used in Monero [52] or ZCash [40], allow users to show others the private aspects of their transactions. While, at a first glance, it seems like viewing keys might be a good solution to our problem, this turns out to be a mirage: it is difficult to have all of efficiency, post-quantum security, and strong auditability in a direct viewing key-based system, and, to have all of these properties, it is necessary to have a "ciphertext equivalence to ZKP" system like we wonstruct, which is well out of the scope of these viewing key solutions.

Other Related Work In [72], the authors build a basic audit structure for CBDCs, but their system is not post-quantum, and there are no formal cryptographic guarantees shown. Quisquis [29] is a clever construction based on a new primitive that the authors call updatable public keys. Quisquis is UTXO but not post-quantum, and does not support auditability. It also is a blockchain system itself (rather than being something that can be run on an existing chain). Blockmaze [37] is a blockchain based on Go-Ethereum that is designed for private transactions. However, it uses an account model, is not PQ-secure, and also does not support auditability. PGC [19] is a cryptocurrency design focused on building a decentralized confidential payment system with auditability. PGC supports auditing and privacy, but the system is account-based (and thus, not as efficient as it could be) and is not PQ-secure. They use a specific form of encryption called twisted ElGamal encryption which means it would be very nontrivial to modify their system to be PQ-secure. In addition, while the authors of PGC do define a model for account-based auditable payments, their modeling is not as formal we desire for a UTXO-based system. ZEXE [12] is a very nice theoretical paper which has a feasible construction, but the system presented has extra generality that we do not need and is not nearly efficient enough for the digital asset use cases we aim to address here.

B ML-KEM (Kyber)

ML-KEM [43] (previously known as CRYSTALS-Kyber [11]) is *Module-Lattice-Based Key-Encapsulation Mechanism Standard* (FIPS 203), selected by NIST as a standard for quantum-secure key encapsulation mechanism. This selection is the result of an extensive, multi-year community effort toward standardization of post-quantum cryptography [56]. We will instantiate our Qurrency with ML-KEM. More precisely, we make use of the CPA-secure public key encryption scheme underlying ML-KEM (denoted K-PKE in FIPS 203), which is sufficient for our purposes and context (see Remark 5). Below, we provide a brief overview of the scheme's encryption algorithm. For completeness, the key generation and decryption algorithms are also briefly reviewed in Algorithms 17 and 18. For more details, refer to [43,11].

Notation For $x \in \mathbb{R}$, we denote by $\lceil x \rfloor$ the nearest integer to x. (If $x = y + 1/2$ for some integer y, we define $\lceil x \rfloor = y + 1$.) Let $\mathbb{Z}_q$ be the set $\{0, 1, \cdots, q-1\} \subseteq \mathbb{Z}$ equipped with addition and multiplication modulo q. The $\bmod q$ operation maps an integer to an element of the set $\mathbb{Z}_q$. We fix the base polynomial ring as $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. It consists of polynomials with degree less than n and coefficients in $\mathbb{Z}_q$. For a vector $\boldsymbol{a}$, we use $\boldsymbol{a}[i]$ to denote its i-th component. Similarly, for $a \in \mathcal{R}_q$, we use $a[i]$ to denote its i-th coefficient. Let B_η be the *centered* binomial distribution whose probability mass function is given as follows for $-\eta \leq k \leq \eta$.

$$\Pr[X = k] = \binom{2\eta}{\eta + k} \cdot 2^{-2\eta}$$

The distribution β_η is defined over $\mathcal{R}_q$, where each coefficient is independently sampled from B_η modulo q. We also define a helper function Compress and Decompress that maps $\mathbb{Z}_q$ element to $\mathbb{Z}_{2^d}$ element where $2^d < q$ and back. We also define helper functions, $\mathsf{Compress}_d$ and $\mathsf{Decompress}_d$, which map elements from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^d}$ for some $2^d < q$, and back, as follows. We extend these functions naturally to inputs in $\mathcal{R}_q$ by applying them coefficient-wise.

$$\begin{aligned} \mathsf{Compress}_d : \mathbb{Z}_q &\to \mathbb{Z}_{2^d} \qquad (1) \\ x &\mapsto \lceil (2^d/q) \cdot x \rfloor \mod 2^d \\ \mathsf{Decompress}_d : \mathbb{Z}_{2^d} &\to \mathbb{Z}_q \qquad (2) \\ x &\mapsto \lceil (q/2^d) \cdot x \rfloor \mod q \end{aligned}$$

Encryption Algorithm The encryption algorithm of ML-KEM is described in Algorithm 14, and the parameter set for ML-KEM-512 is provided in Table 3. This parameter set targets a security level comparable to, or exceeding, that required to resist key search attacks on a block cipher with a 128-bit key (e.g., AES128). The column *DFR* in Table 3 denotes the decryption failure rate.

Remark 3. The description of Algorithm 14 closely follows the standardized encryption algorithm, modulo some omitted implementation details (e.g. the use of the NTT or the byte-level encoding of the ciphertext). The only difference lies in the form of the public key: in the standard, $\boldsymbol{A}$ is not provided explicitly but rather compressed into a hash seed, which is then expanded within the encryption algorithm. In our presentation, however, we assume that the expansion has already been performed, and we take the public key as an explicit input in the form $(\boldsymbol{A}, \boldsymbol{t})$. This is to later simplify the proof of *well-formedness of ciphertexts* by allowing the deterministic expansion to be checked outside the proof system.

n	q	k	η_1	η_2	d_u	d_v	DFR
256	3329	2	3	2	10	4	$2^{-138.8}$

Table 3: Parameter Set for ML-KEM-512

B.1 Discussions

Remark 4 (On Choice of ML-KEM). While Qurrency can be instantiated with any post-quantum PKE scheme, we choose ML-KEM, as it is the only NIST-standardized post-quantum PKE scheme at the time of writing. Although HQC [41] was recently selected as a backup algorithm to ML-KEM, its standard is not expected to be finalized until 2027 [55]. Moreover, HQC has considerably larger ciphertext sizes than ML-KEM—a critical consideration in our setting, where ciphertexts are stored on-chain.

$\mathsf{Enc}(\mathsf{pk}, m; R)$

Input: Public key $\mathsf{pk} = (\boldsymbol{A}, \boldsymbol{t})$; $\boldsymbol{A} \in \mathcal{R}_q^{k \times k}$ and $\boldsymbol{t} \in \mathcal{R}_q^k$
Input: Message $m \in \mathcal{R}_q$; $m[i] \in \{0,1\}$ for all $0 \leq i < n$
Input: Randomness $R \in \{0,1\}^{256}$

$\boldsymbol{y} \leftarrow \beta_{\eta_1}^k$ ▷ via $\mathsf{Sample}(R)$ (Algorithm 15)
$\boldsymbol{e}_1 \leftarrow \beta_{\eta_2}^k$
$e_2 \leftarrow \beta_{\eta_2}$

$\boldsymbol{u} \leftarrow \boldsymbol{A}^\top \boldsymbol{y} + \boldsymbol{e}_1$
$v \leftarrow \boldsymbol{t}^\top \boldsymbol{y} + e_2 + \lceil \frac{q}{2} \rfloor \cdot m$

$c_1 \leftarrow \mathsf{Compress}_{d_u}(\boldsymbol{u})$ ▷ Eq. (1)
$c_2 \leftarrow \mathsf{Compress}_{d_v}(v)$

return $c \leftarrow (c_1, c_2)$

$\mathsf{Sample}(R)$

Input: Randomness $R \in \{0,1\}^{256}$
Output: Noises $\boldsymbol{y} \leftarrow \beta_{\eta_1}^k$, $\boldsymbol{e}_1 \leftarrow \beta_{\eta_2}^k$, and $e_2 \leftarrow \beta_{\eta_2}$

$N \leftarrow 0$
for $(i \leftarrow 0;\ i < k;\ i$++$)$ **do**
    $\boldsymbol{y}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(H(R||N))$
    ▷ H is a cryptographic hash function.
    $N \leftarrow N + 1$
end for
for $(i \leftarrow 0;\ i < k;\ i$++$)$ **do**
    $\boldsymbol{e}_1[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(H(R||N))$
    $N \leftarrow N + 1$
end for
$e_2 \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(H(R||N))$
return $\boldsymbol{y}, \boldsymbol{e}_1, e_2$

$\mathsf{SamplePolyCBD}_\eta(\boldsymbol{b})$

Input: Random Bitstring $\boldsymbol{b} \in \{0,1\}^\ell$, where $\ell = 2\eta \cdot n$
Output: Polynomial $a \in \mathcal{R}_q$ following the distribution β_η

for $(i \leftarrow 0;\ i < n;\ i$++$)$ **do**
    $x \leftarrow \sum_{j=0}^{\eta-1} \boldsymbol{b}[2\eta \cdot i + j]$
    $y \leftarrow \sum_{j=0}^{\eta-1} \boldsymbol{b}[2\eta \cdot i + \eta + j]$
    $a[i] \leftarrow x - y \mod q$
end for
return a

$\mathsf{KeyGen}(R)$

Input: Randomness $R \in \{0,1\}^{256}$

$\boldsymbol{A} \leftarrow \mathcal{R}_q^{k \times k}$ ▷ From randomness R
$\boldsymbol{s} \leftarrow \beta_{\eta_1}^k$
$\boldsymbol{e} \leftarrow \beta_{\eta_1}^k$

$\boldsymbol{t} \leftarrow \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e}$

return $\mathsf{pk} \leftarrow (\boldsymbol{A}, \boldsymbol{t})$ and $\mathsf{sk} \leftarrow \boldsymbol{s}$

$\mathsf{Dec}(\mathsf{sk}, c)$

Input: Secret key $\mathsf{sk} = \boldsymbol{s} \in \mathcal{R}_q^k$
Input: Ciphertext $c = (c_1, c_2)$

$\boldsymbol{u}' \leftarrow \mathsf{Decompress}_{d_u}(c_1)$ ▷ Eq. (2)
$v' \leftarrow \mathsf{Decompress}_{d_v}(c_2)$
$w \leftarrow v' - \boldsymbol{s}^\top \boldsymbol{u}'$
$m \leftarrow \mathsf{Compress}_1(w)$ ▷ Eq. (1)
return m

Remark 5 (Plaintext-awareness and CCA Security). The proof for *well-formedness of ciphertexts* enforces the so-called plaintext-awareness [6,39] property for the underlying PKE scheme. Note that plaintext-awareness directly implies CCA security: if an adversary already knows the response it would receive from a decryption oracle, then access to the oracle provides no additional advantage. Given this observation, it suffices for us to use a CPA-secure encryption scheme rather than a CCA-secure one. Accordingly, we instantiate our scheme using the CPA-secure PKE underlying ML-KEM.

Remark 6 (Correctness against Adversarial Encryptor of ML-KEM). In Qurrency, we need a stronger notion of correctness for underlying PKE, in which decryption succeeds not only on honestly generated ciphertexts, but even when the message and randomness are chosen by a computationally bounded adversary. This stronger guarantee is essential for auditability: specifically, to prevent malicious parties from evading audits by submitting ciphertexts that cannot be decrypted.

For perfectly correct schemes, this property can be ensured by requiring ciphertexts to include a proof that they were generated by honestly following the encryption algorithm with a valid message and randomness. However, in schemes like ML-KEM, which allow a decryption failure probability[16], special care is needed. An adversary might craft ciphertexts using specially chosen randomness to intentionally induce decryption failures. To address this and ensure correctness against adversarial encryptors[17], we pass the input randomness through a cryptographic hash function (SHAKE256), modeled as a random oracle, and use the resulting output as the effective randomness (Algorithm 15).

We emphasize that our approach does not alter the ML-KEM standard itself. In fact, FIPS 203 [43] already specifies the use of a cryptographic hash function H instantiated with SHAKE256, modeled as a *pseudorandom function (PRF)*, to securely derive encryption randomness from a short randomness seed, as shown in Algorithm 15. While the standard treats SHAKE256 as a PRF—sufficient for ensuring security in honest encryption scenarios—our auditability requirements necessitate modeling it as a random oracle to support security proofs in the presence of potentially malicious encryptors.

C Preliminaries

We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. The security parameter is denoted by $\lambda \in \mathbb{N}$. A function $f : \mathbb{N} \to \mathbb{N}$ is said to be polynomial if there exists a constant c such that $f(n) \leq n^c$ for all $n \in \mathbb{N}$, and we write $\mathsf{poly}(\cdot)$ to denote such a function. A function $f : \mathbb{N} \to [0, 1]$ is said to be negligible if for every $c \in \mathbb{N}$, there exists $N \in \mathbb{N}$ such that for all $n > N$, $f(n) < n^{-c}$, and we write $\mathsf{negl}(\cdot)$ to denote such a function. A probability is *noticeable* if it is not negligible, and *overwhelming* if it is equal to $1 - \mathsf{negl}(\lambda)$ for some negligible function $\mathsf{negl}(\lambda)$. For a set $\mathcal{S}$, we write $s \leftarrow \mathcal{S}$ to indicate that s is sampled uniformly at random from $\mathcal{S}$. For a random variable $\mathcal{D}$, we write $d \leftarrow \mathcal{D}$ to indicate that d is sampled according to $\mathcal{D}$. An algorithm $\mathcal{A}$ is PPT (probabilistic polynomial-time) if its running time is bounded by some polynomial in the size of its input. For two ensembles of random variables $\{\mathcal{D}_{0,\lambda}\}_{\lambda\in\mathbb{N}}$, $\{\mathcal{D}_{1,\lambda}\}_{\lambda\in\mathbb{N}}$, we write $\mathcal{D}_0 \approx_c \mathcal{D}_1$ to indicate that for all PPT $\mathcal{A}$, it holds that

$$\left| \Pr_{d\leftarrow\mathcal{D}_{0,\lambda}}[\mathcal{A}(d) = 1] - \Pr_{d\leftarrow\mathcal{D}_{1,\lambda}}[\mathcal{A}(d) = 1] \right| \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

[16] e.g. ML-KEM-512 has a decryption failure rate of $2^{-138.8}$.

[17] An alternative approach is to modify the parameters of ML-KEM to achieve perfect correctness. However, this would deviate from the standard and also significantly reduce efficiency.

C.1 Non-Interactive Zero-Knowledge Proofs

Let $\mathcal{L}$ be an NP-language and $\mathcal{R}$ the corresponding NP-relation. A Simulation Extractable-NIZK for $\mathcal{R}$ consists of the following algorithms:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{crs}, \mathsf{td})$: Takes as input a security parameter, and outputs a common reference string crs and trapdoor td.
- $\mathsf{Prove}(\mathsf{crs}, x, w) \to \pi$: Takes as input crs and any pair $(x, w) \in \mathcal{R}$ and outputs a proof π for the statement $x \in \mathcal{L}$.
- $\mathsf{Verify}(\mathsf{crs}, x, \pi) \to b$: Takes as input crs, statement x and proof π and outputs a bit b indicating whether verification has passed or failed.
- $\mathsf{SimProve}(\mathsf{crs}, \mathsf{td}, x) \to \pi$: Takes as input crs, trapdoor td and statement x and outputs a *simulated* proof π.

We will drop the crs term wherever implicit. A SE-NIZK satisfies the following properties:

- *Perfect Completeness.* An SE-NIZK satisfies perfect completeness if for any $(x, w) \in \mathcal{R}$, we have

$$\Pr\left[\begin{matrix}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Setup}(1^\lambda) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, x, w)\end{matrix} : \mathsf{Verify}(x, \pi) = 1\right] = 1.$$

- *Proof of knowledge (and soundness).* For every PPT adversary $\mathcal{A}$, there is a PPT extractor Ext such that

$$\Pr\left[\begin{matrix}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (x, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) \\ w \leftarrow \mathsf{Ext}(\mathsf{crs})\end{matrix} : \begin{matrix}\mathsf{Verify}(x, \pi) = 1 \\ (x, w) \notin \mathcal{R}\end{matrix}\right] \leq \mathsf{negl}(\lambda).$$

- *Computational Zero-knowledge.* There exists a simulator Sim such that for all stateful distinguishers $\mathcal{A}$ the following probabilities are equal:

$$\left|\Pr\left[\begin{matrix}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (x, w) \leftarrow \mathcal{A}(\mathsf{crs}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, x, w)\end{matrix} : \begin{matrix}(x, w) \in \mathcal{R} \\ \mathcal{A}(\pi) = 1\end{matrix}\right] - \Pr\left[\begin{matrix}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (x, w) \leftarrow \mathcal{A}(\mathsf{crs}) \\ \pi \leftarrow \mathsf{SimProve}(\mathsf{crs}, \mathsf{td}, x)\end{matrix} : \begin{matrix}(x, w) \in \mathcal{R} \\ \mathcal{A}(\pi) = 1\end{matrix}\right]\right| \leq \mathsf{negl}(\lambda).$$

We will also demand that the proof is straight-line extractable. This means that the extractor can, on input a valid proof and (potentially) the list of random-oracle queries made by the adversary (prover), extract a witness with overwhelming probability without rewinding the prover.

C.2 The Random Oracle Model

In the random oracle model (ROM), parties are given oracle access to some function H that is sampled uniformly at random from the space of all functions $H : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are finite non-empty sets. That is, parties can query their oracle on an input $x \in X$ and receive in return $H(x) \in \mathcal{Y}$.

C.3 Sparse Merkle Trees

For this work, we will use Sparse Merkle Trees (SMT) [21]. These trees are a memory-efficient version of Merkle Trees [50]. Further, this work will only require addition and updation of leaf nodes in the SMT, and we thus limit our description to those algorithms for changes made to the tree.

$\mathsf{Setup}(1^\lambda) \to T$**:** Returns an empty tree T.
$\mathsf{addLeaf}(T, (i, v)) \to T$**:** Adds the node with index i and value v as a leaf to the tree T. Returns the updated tree.
$\mathsf{updateLeaf}(T, (i, v)) \to T$**:** Updates the value of the with index i to v in the tree T. Returns the updated tree.
$\mathsf{Prove}(T, \mathsf{rt}, (i, v)) \to \tau$**:** Produces a proof τ with respect to the root rt that the leaf (i, v) is in T.

$\mathsf{Verify}(\mathsf{rt}, (i, v)) \to b$**:** Checks if the proof τ with respect to the root rt for leaf (i, v) is valid.
$\mathsf{inTree}(T, (i, v)) \to b$**:** Checks if there is a leaf node with index i and value v in the tree T.
$\mathsf{indexInTree}(T, i) \to b$**:** Checks if there is a leaf node with index i (and *any* value) in the tree T.
$\mathsf{isInRoots}(T, r) \to b$**:** Checks is r was ever the root of the append only tree T.

We note that addLeaf, updateLeaf, Prove and Verify are implemented as standard Sparse Merkle Tree algorithms. To implement the other algorithms, one considers the implementation of a Sparse Merkle Tree that additionally maintains: (i) a list of all prior roots of the tree, adding the newest root to this list whenever there is an addLeaf or updateLeaf call; and (ii) a table of all nodes (internal and leaf) in the tree mapping the hash of the actual node. The list of all nodes in the tree allows to efficiently evaluate both inTree and indexInTree. For inTree one can directly check if the node exists in the table by hashing to check if the table contains the specified node at the hash, and for indexInTree one starts at the *current* root of the tree and follows the path to the leaf (using the index i to traverse down the left or right child of each node in the tree), retrieving the next node to traverse by looking up the hash of the node.

D NIZK Languages

We specify here the languages for the various transactions, cast as special instances of $\mathcal{L}$ (Fig. 1).

deposit:

$$\mathcal{L}_{\mathsf{deposit}} = \big\{(\mathsf{ct}, v, \{\mathsf{utxo}_i\}_{i\in[n]}) \mid (v, 0, \mathsf{ct}, \bot, \bot, \phi, \{\mathsf{utxo}_i\}_{i\in[n]}, \phi) \in \mathcal{L}\big\}$$

withdraw:

$$\mathcal{L}_{\mathsf{withdraw}} = \big\{(\mathsf{ct}, v, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}') \mid (0, v, \mathsf{ct}, \mathsf{rt}, \bot, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}', \phi) \in \mathcal{L}\big\}$$

transfer:

$$\mathcal{L}_{\mathsf{transfer}} = \big\{(\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}) \mid (0, 0, \mathsf{ct}, \mathsf{rt}, \bot, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n]}, \phi) \in \mathcal{L}\big\}$$

transferLocked:

$$\mathcal{L}_{\mathsf{transferLocked}} = \big\{(\mathsf{ct}, \mathsf{rt}, \mathsf{delegate}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}) \mid (0, 0, \mathsf{ct}, \mathsf{rt}, \mathsf{delegate}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n]}, \phi) \in \mathcal{L}\big\}$$

lock:

$$\mathcal{L}_{\mathsf{lock}} = \big\{(\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}) \mid (0, 0, \mathsf{ct}, \mathsf{rt}, \bot, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}) \in \mathcal{L}\big\}$$

E Smart Contract

We formally describe the smart contracts in **??**. When describing the smart contract, we use Solidity syntax in several places and provide explanations of the syntax below.

msg.sender sender of the message (current call).
msg.value amount of ETH (Wei) sent with the message.
address.transfer send given amount of ETH (Wei) to address.
require check conditions; current call reverts if condition fails.

Remark 7. We make the simplifying assumption in the description of these functions that the input nullifiers and UTXOs are all *distinct*. In practice, these are efficiently checked by sorting the inputs and checking consecutive elements in this sorted list.

Qurrency Smart Contract Helper: `Validate`

1: INPUTS: nullifiers $\{\mathsf{null}_i\}_{i\in[n]}$, output UTXOs $\{\mathsf{utxo}'_i\}_{i\in[n']}$, locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}$
2: Check (input ϕ immediately passes the corresponding check)
   1. $\forall i \in [n]$, $\mathsf{Nullifiers}[\mathsf{null}_i] = 0$.
   2. $\forall i \in [n']$, $\mathsf{inTree}(\mathsf{CommTree}, (\mathsf{utxo}'_i, \mathsf{utxo}'_i)) = 0$.
   3. $\forall i \in [\widetilde{n}]$, $\mathsf{inTree}(\mathsf{CommTree}, (\widetilde{\mathsf{utxo}}_i, \widetilde{\mathsf{utxo}}_i)) = 0$.
   4. $\forall i \in [n']$, $\mathsf{indexInTree}(\mathsf{LockedCommTree}, \mathsf{utxo}'_i,) = 0$.
   5. $\forall i \in [\widetilde{n}]$, $\mathsf{indexInTree}(\mathsf{LockedCommTree}, \widetilde{\mathsf{utxo}}_i,) = 0$.
3: Return 1 if all checks pass.

Qurrency Smart Contract Helper: `UpdateState`

1: INPUTS: nullifiers $\{\mathsf{null}_i\}_{i\in[n]}$ output UTXOs $\{\mathsf{utxo}'_i\}_{i\in[n']}$ locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}$ delegate address ethAddr
2: The update step is skipped for the input if it is ϕ.
   1. $\forall i \in [n]$, $\mathsf{Nullifiers}[\mathsf{null}_i] \coloneqq 1$.
   2. $\forall i \in [n']$, $\mathsf{addLeaf}(\mathsf{CommTree}, (\mathsf{utxo}'_i, \mathsf{utxo}'_i))$
   3. $\forall i \in [\widetilde{n}]$, $\mathsf{addLeaf}(\mathsf{LockedCommTree}, (\widetilde{\mathsf{utxo}}_i, \mathsf{ethAddr}))$

Qurrency Smart Contract: `Deposit`

1: INPUTS: amount v, output UTXOs $\{\mathsf{utxo}'_i\}_{i\in[n]}$, proofs π, ciphertext ct
2: require
   – $\texttt{Validate}(\phi, \{\mathsf{utxo}'_i\}_{i\in[n]}, \phi) = 1$
   – $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathsf{deposit}}, (\mathsf{ct}, v, \{\mathsf{utxo}'_i\}_{i\in[n]}), \pi) = 1$
3: $\texttt{UpdateState}(\phi, \{\mathsf{utxo}'_i\}_{i\in[n]}, \phi, \perp)$

Qurrency Smart Contract: `Withdraw`

1: INPUTS: amount v, nullifiers $\{\mathsf{null}_i\}_{i\in[n]}$, output UTXO utxo' (maybe $\perp$ if no output UTXO), proofs π, root rt, ciphertext ct
2: require
   - $\texttt{Validate}(\{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}', \phi) = 1$.
   - $\mathsf{isInRoots}(\mathsf{CommTree}, \mathsf{rt}) = 1$
   - $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathsf{withdraw}}, (\mathsf{ct}, v, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}'), \pi) = 1$.
3: $\texttt{UpdateState}(\{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}', \phi, \perp)$
4: Do $\mathsf{msg.sender.transfer}(v)$.

Qurrency Smart Contract: `Transfer`

1: INPUTS: nullifiers $\{\mathsf{null}_i\}_{i\in[n]}$, output UTXOs $\{\mathsf{utxo}'_i\}_{i\in[n']}$, root rt, proofs π, ciphertext ct
2: require
   - $\texttt{Validate}(\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \phi) = 1$
   - $\mathsf{isInRoots}(\mathsf{CommTree}, \mathsf{rt}) = 1$
   - $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathsf{transfer}}, (\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}), \pi) = 1$.
3: $\texttt{UpdateState}(\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \phi, \perp)$.

Qurrency Smart Contract: `TransferLocked`

1: INPUTS: nullifiers $\{\mathsf{null}_i\}_{i\in[n]}$, output UTXOs $\{\mathsf{utxo}'_i\}_{i\in[n']}$, root rt, proofs π, ciphertext ct
2: require
   - $\texttt{Validate}(\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \phi) = 1$
   - $\mathsf{isInRoots}(\mathsf{LockedCommTree}, \mathsf{rt}) = 1$
   - $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathsf{transferLocked}}, (\mathsf{ct}, \mathsf{rt}, \mathsf{msg.sender}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}), \pi) = 1$.
3: $\texttt{UpdateState}(\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \phi, \perp)$.

Qurrency Smart Contract: `Lock`

1: INPUTS: nullifiers $\{\mathsf{null}_i\}_{i\in[n]}$, output UTXOs $\{\mathsf{utxo}'_i\}_{i\in[n']}$, locked output UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}$, proofs π, ciphertext ct, delegate address $\mathsf{ethAddr}$ root rt
2: require
   - $\texttt{Validate}(\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}) = 1$.
   - $\mathsf{isInRoots}(\mathsf{CommTree}, \mathsf{rt}) = 1$
   - $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathsf{lock}}, (\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}), \pi) = 1$.
3: $\texttt{UpdateState}(\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}, \mathsf{ethAddr})$.

Qurrency Smart Contract: `DelegateLock`

1: INPUTS: locked UTXOs $\{\widetilde{\mathsf{utxo}}_i\}_{i\in[\widetilde{n}]}$, delegate address $\mathsf{ethAddr}$
2: require
   - $\forall i \in [\widetilde{n}]$, $\mathsf{inTree}(\mathsf{LockedCommTree}, (\widetilde{\mathsf{utxo}}_i, \mathsf{msg.sender}))$.
3: $\forall i \in [\widetilde{n}]$, $\mathsf{updateLeaf}(\mathsf{LockedCommTree}, (\widetilde{\mathsf{utxo}}_i, \mathsf{ethAddr}))$.

F Algorithms

F.1 Setup Algorithm

Setup

Input: Security parameter λ (in unary) and auditor public key apk

1: $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^\lambda$.
2: $\mathsf{NIZK.pp} \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$.
3: Initialize $\mathsf{CommTree} \leftarrow \mathsf{SMT.Setup}(1^\lambda)$, $\mathsf{LockedCommTree} \leftarrow \mathsf{SMT.Setup}(1^\lambda)$ and table $\mathsf{Nullifiers} : \{0,1\}^\lambda \mapsto \{0,1\}$.
4: Deploy a smart contract with parameters H, $\mathsf{NIZK.pp}$, $\mathsf{CommTree}$, $\mathsf{LockedCommTree}$, $\mathsf{Nullifiers}$, apk.

F.2 User Algorithms

User Algorithm: `CreateKey`

1: INPUTS: 1^λ
2: OUTPUTS: $\mathsf{sk} \in \mathbb{Z}_p, \mathsf{pk} \in \mathbb{G}$
3: $\mathsf{sk} \leftarrow\!\$\ \mathbb{Z}_p$ and $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$

User Algorithm: `CreateDepositTx`

1: INPUTS: amount v in ETH, recipient amounts and owners $\{v'_i, \mathsf{pk}'_i\}_{i\in[n']}$
2: OUTPUTS: $\mathsf{tx}_{\mathsf{deposit}} = (v, \{\mathsf{utxo}'_i\}_{i\in[n]}, \mathsf{ct}, \pi_{\mathsf{deposit}})$, salts $\{\rho'_i\}_{i\in[n']}$
3: sample salts $\{\rho'_i \leftarrow\!\$\ \mathbb{Z}_q\}_{i\in[n']}$
4: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
5: set $m = (\phi, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \phi)$
6: sample $R \leftarrow\!\$\ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
7: prove $\pi_{\mathsf{deposit}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{deposit}})$:
   – $\mathsf{sta} : (\mathsf{ct}, v, \{\mathsf{utxo}'_i\}_{i\in[n']})$
   – $\mathsf{wit} : (R, m)$

User Algorithm: `CreateWithdrawTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, withdrawal amount v in ETH, owner of balance amount pk', root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{withdraw}} = (v, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}', \mathsf{rt}, \mathsf{ct}, \pi_{\mathsf{withdraw}})$, salt ρ'
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{CommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i)\}_{i\in[n]}$
4: if $\sum_{i\in[n]} v_i > v$:
   – compute $v' = \sum_{i\in[n]} v_i - v$
   – sample $\rho' \leftarrow\!\$\ \mathbb{Z}_q$
   – compute $\mathsf{utxo}' \leftarrow H(v', \rho', \mathsf{pk}')\}_{i\in[n]}$
5: else $\mathsf{utxo}' \leftarrow \perp$, $v' \leftarrow \perp$ and $\rho' \leftarrow \perp$.
6: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, (v', \rho', \mathsf{pk}'), \phi)$
7: sample $R \leftarrow\!\$\ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
8: prove $\pi_{\mathsf{withdraw}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{withdraw}})$:
   – $\mathsf{sta} : (\mathsf{ct}, v, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \mathsf{utxo}')$
   – $\mathsf{wit} : (R, m)$

User Algorithm: `CreateTransferTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, recipient amounts and owners $\{(v'_i, \mathsf{pk}'_i)\}_{i\in[n']}$, root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{transfer}} = (\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \mathsf{rt}, \mathsf{ct}, \pi_{\mathsf{transfer}})$, salts $\{\rho'_i\}_{i\in[n']}$
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{CommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i)\}_{i\in[n]}$
4: sample salts $\{\rho'_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[n']}$
5: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
6: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \phi)$
7: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
8: prove $\pi_{\mathsf{transfer}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{transfer}})$:
- sta : $(\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']})$
- wit : (R, m)

User Algorithm: `CreateLockTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, recipient amounts and owners $\{(v'_i, \mathsf{pk}'_i)\}_{i\in[n']}$, locked recipient amounts and owners $\{(\widetilde{v}_i, \widetilde{\mathsf{pk}_i})\}_{i\in[n']}$, delegate address ethAddr root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{lock}} = (\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}_i}\}_{i\in[\widetilde{n}]}, \mathsf{rt}, \mathsf{ct}, \mathsf{ethAddr}, \pi_{\mathsf{lock}})$, salts $\{\rho'_i\}_{i\in[n']}$ and $\{\widetilde{\rho}_i\}_{i\in[\widetilde{n}]}$
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{CommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{utxo}_i)\}_{i\in[n]}$
4: sample salts $\{\rho'_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[n']}$ and $\{\widetilde{\rho}_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[\widetilde{n}]}$
5: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
6: compute $\{\widetilde{\mathsf{utxo}_i} \leftarrow H(\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}_i})\}_{i\in[\widetilde{n}]}$
7: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \{\widetilde{v}_i, \widetilde{\rho}_i, \widetilde{\mathsf{pk}_i}\}_{i\in[\widetilde{n}]})$
8: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
9: prove $\pi_{\mathsf{lock}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{lock}})$:
- sta : $(\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \{\widetilde{\mathsf{utxo}_i}\}_{i\in[\widetilde{n}]})$
- wit : (R, m)

User Algorithm: `CreateTransferLockedTx`

1: INPUTS: input UTXO owner information $\{(\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$, recipient amounts and owners $\{(v'_i, \mathsf{pk}'_i)\}_{i\in[n']}$, root rt, state st of SC
2: OUTPUTS: $\mathsf{tx}_{\mathsf{transferLocked}} = (\{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']}, \mathsf{rt}, \mathsf{ct}, \pi_{\mathsf{lock}})$, salts $\{\rho'_i\}_{i\in[n']}$
3: compute $\{\mathsf{null}_i \leftarrow H(v_i, \rho_i, \mathsf{sk}_i)\}_{i\in[n]}$ and $\{\tau_i \leftarrow \mathsf{LockedCommTree.Prove}(\mathsf{st}, \mathsf{rt}, (\mathsf{utxo}_i, \mathsf{msg.sender})\}_{i\in[n]}$
4: sample salts $\{\rho'_i \leftarrow\$ \mathbb{Z}_q\}_{i\in[n']}$
5: compute $\{\mathsf{utxo}'_i \leftarrow H(v'_i, \rho'_i, \mathsf{pk}'_i)\}_{i\in[n']}$
6: set $m = (\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{sk}_i, \mathsf{pk}_i = g^{\mathsf{sk}_i}, \tau_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']}, \phi)$
7: sample $R \leftarrow\$ \mathsf{D}_R$ and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{apk}, m; R)$
8: prove $\pi_{\mathsf{transfer}} \leftarrow \mathsf{Prove}(\mathcal{L}_{\mathsf{transferLocked}})$:
- sta : $(\mathsf{ct}, \mathsf{rt}, \{\mathsf{null}_i\}_{i\in[n]}, \{\mathsf{utxo}'_i\}_{i\in[n']})$
- wit : (R, m)

User Algorithm: `CreateDelegateLockTx`

1: INPUTS: input UTXO $\{(\mathsf{utxo}_i)\}_{i\in[n]}$, delegate ethAddr, state st of SC
2: OUTPUT: $\mathsf{tx}_{\mathsf{delegateLock}} = (v, \{\mathsf{utxo}\}_{i\in[n]}, \mathsf{delegate})$

User Algorithm: `ReadValue`

1: INPUTS: UTXO utxo, salt ρ, amount v in ETH, public key pk
2: OUTPUT: $b = 1$ if $\mathsf{utxo} = H(v, \rho, \mathsf{pk})$, else output $b = 0$.

User Algorithm: `Audit`

1: INPUTS: auditor secret key ask, transaction tx
2: OUTPUT: $\{\mathsf{utxo}_i, v_i, \rho_i, \mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, \{v'_i, \rho'_i, \mathsf{pk}'_i\}_{i\in[n']} = \mathsf{Dec}(\mathsf{ask}, \mathsf{ct})$ where ct is contained in tx.

User Algorithm: `NullifierCheck`

1: INPUTS: UTXO utxo, salt ρ, amount v in ETH, public key pk, secret key sk, nullifier null
2: OUTPUT: $b = 1$ if $\mathsf{utxo} = H(v, \rho, \mathsf{pk})$ AND $\mathsf{null} = H(v, \rho, \mathsf{sk})$, else output $b = 0$.

User Algorithm: `checkSpend`

1: INPUTS: UTXO utxo, salt ρ, amount v in ETH, public key pk, secret key sk, nullifier null, state st of SC
2: OUTPUT: $b = 1$ if utxo $= H(v, \rho, \mathsf{pk})$ AND null $= H(v, \rho, \mathsf{sk})$, else output $b = 0$.

User Algorithm: `checkUnlock`

1: INPUTS: UTXO utxo, state st of SC
2: OUTPUT: $b = 1$ if inTree(CommTree, (utxo, utxo)) $= 1$, else output $b = 0$.

G Ideal Functionality $\mathcal{F}_{\text{pq-aud-priv-cash}}$

We describe below the ideal functionality $\mathcal{F}_{\text{pq-aud-priv-cash}}$, that interacts with the parties and a special *auditor* party $\mathcal{P}_{\text{Auditor}}$. The adversary $\mathcal{A}$ corrupts a set of parties $\mathcal{C}$. We will interchangeably use "party" to also refer to corresponding Ethereum addresses (though participants can create multiple addresses). When a coin is created with value v, its existence would typically be checked against a ledger (like the Ethereum ledger). For simplicity, we do not formally model the ledger functionality or its interaction with $\mathcal{F}_{\text{pq-aud-priv-cash}}$, although we note that it would be straightforward to do so with existing ledger models (e.g. [34]).

Data Structures. The ideal functionality maintains the following lists

keysList: This list contains tuples of the form $(\mathsf{pk}, \mathcal{S})$ which maps the public key pk to the set of parties that have access to transactions addressed to pk. For honest parties, it is simply the party that registers pk, whereas for corrupted parties this is the set of *all* corrupted parties.

Coins: This is a list of the form $(c, \mathsf{pk}, v, \mathcal{P}$ or $\perp)$ which has a handle c, the public key pk of the owner, the value v associate with a coin, and the party $\mathcal{P}$ that the coin is locked to. If it is unlocked, this is set to $\perp$.

We make a few observations regarding the ideal functionality definition. Specifically, as long as a protocol realizes the ideal functionality, the following properties hold.

Correctness The correctness follows directly from the ideal functionality description.

Balance For every transaction that is submitted, the functionality ensures that: (i) the coins to be spent exist; and (ii) the sum of values in the spent coins equals to the sum of values in the output coins and the optional additional public value. Since the coins are deleted immediately after being spent, it removes the possibility of the same coin being spent twice.

Locked Soundness For a coin that is locked to a specific party, only that party is allowed to spend the coin.

Auditability The auditor party is informed of the details of every transaction that is sent to the ideal functionality, i.e. the coins involved in the transaction along with the corresponding values and public keys for each coin is sent to $\mathcal{P}_{\text{Auditor}}$. It is important to note that while the auditor learns the contents of every transaction, even a corrupted $\mathcal{P}_{\text{Auditor}}$ is not able to affect the balance properties of the scheme, i.e. "soundness" of the scheme is preserved in the presence of a corrupted auditor.

Privacy Except in the case that $\mathcal{P}_{\text{Auditor}}$ is corrupted, the adversarial parties do not learn any information about the coins that it was involved in. In particular it only learns information about the coins that are sent to it, or the coins that are created in a transaction from the adversary.

An additional point of note is that when coins are created, even if the recipient is the honest party they are not immediately notified. This is done to model the fact that when a coin is created for a party, there is an additional out-of-band communication required to communicate the information necessary to spend the coin. While an honest party will always do this, an adversarial party may create a coin intended for an honest party and never send the additional spending information. To capture this we allow the adversary to decide when it wants to inform the honest party of the coin since in the ideal world, an honest party informed of the coin would be able to immediately spend it.

$\mathcal{F}_{\text{pq-aud-priv-cash}}$

Special auditor party $\mathcal{P}_{\text{Auditor}}$, and corrupted set $\mathcal{C}$ that may contain $\mathcal{P}_{\text{Auditor}}$.

`Register` : On receiving pk from party $\mathcal{P}$,

1. If $\mathcal{P}$ is corrupted, store $(\mathsf{pk}, \mathcal{C})$ in keysList
2. Else store $(\mathsf{pk}, \{\mathcal{P}\})$ in keysList

`Deposit` : On receiving v, $\{v_i, \mathsf{pk}_i\}_{i\in[n]}$ from $\mathcal{P}$

1. Assert that $v = \sum_{i=1}^{n} = v_i$.

2. Assert that for every $i \in [n]$, there exists some $\mathcal{S}_i$ such that $(\mathsf{pk}_i, \mathcal{S}_i)$ in keysList.
3. For every $i \in [n]$,
   (a) Sample c_i at random.
   (b) Send (c, v_i, pk_i) to $\mathcal{S}_i$ if $\mathcal{S}_i = \mathcal{C}$;
   (c) If $\mathcal{P}$ not corrupted send (c, v_i, pk_i) to $\mathcal{P}_i$ where $\mathcal{S}_i = \{\mathcal{P}_i\}$; //coins to honest party not sent if $\mathcal{P}$ is corrupted
   (d) Add $(c_i, \mathsf{pk}_i, v_i, \bot)$ to the list Coins.
4. Return $(c_1, \ldots, c_n)$ to $\mathcal{P}$.
5. Send $\{(c_i, \mathsf{pk}_i, v_i, \bot)\}_{i\in[n]}$ to the $\mathcal{P}_{\mathsf{Auditor}}$.
6. Broadcast $\mathtt{Deposit}(v)$

DelegateLock: On receiving c, $\mathcal{P}'$ from $\mathcal{P}$

1. Assert that there exists pk, v such that $(c, \mathsf{pk}, v, \mathcal{P})$ in Coins.
2. Modify $(c, \mathsf{pk}, v, \mathcal{P})$ to be $(c, \mathsf{pk}, v, \mathcal{P}')$.
3. Send $(c, v, \mathsf{pk}, \mathcal{P}')$ to $\mathcal{P}'$ where v is the coin value.
4. Broadcast $\mathtt{Lock}(\mathcal{P}')$.

InformCoin : On receiving c from the adversary $\mathcal{A}$

1. Assert there exists v, pk, $\mathcal{P}$ such that $(c, v, \mathsf{pk}, \mathcal{P})$ or $(c, v, \mathsf{pk}, \bot)$ in Coins.
2. If $(\mathsf{pk}, \{\mathcal{P}_i\}) \in$ keysList, send $(c, v, \mathsf{pk}, \star)$ to $\mathcal{P}_i$ where $\star$ is $\mathcal{P}$ or $\bot$.

//Only need to inform the party that owns the coin c since the locking will already broadcasts to the party that a locked coin was created.

Transact: On receiving Type, v, $\{c_i\}_{i\in[n]}$, $\{v'_i, \mathsf{pk}'_i\}_{i\in[n']}$, $\mathcal{P}'$ from $\mathcal{P}$

1. If $\mathtt{Type} = \mathtt{LockedTransfer}$ assert that for every $i \in [n]$
   (a) there exists v_i such that $(c_i, \star, v_i, \mathcal{P})$ in Coins
2. Else if $\mathtt{Type} = \mathtt{UnlockedTransfer}$ or $\mathtt{CreateLockedCoins}$, assert that for every $i \in [n]$:
   (a) there exists pk_i, v_i such that $(c_i, \mathsf{pk}_i, v_i, \bot)$ in Coins; and
   (b) $\mathcal{P} \in \mathcal{S}_i$ where $(\mathsf{pk}_i, \mathcal{S}_i)$ in keysList.
3. Assert that $\sum_{i=1}^{n'} v'_i + v = \sum_{i=1}^{n} v_i$.
4. For every $i \in [n']$,
   (a) Sample c'_i at random.
   (b) If $\mathtt{Type} = \mathtt{UnlockedTransfer}$
      i. Add $(c'_i, \mathsf{pk}'_i, v'_i, \bot)$ to the list Coins.
      ii. Send $(c'_i, \mathsf{pk}'_i, v'_i, \bot)$ to $\mathcal{S}_i$ if either $\mathcal{S}_i = \mathcal{C}$, or $\mathcal{P}$ is honest.
   (c) Else if $\mathtt{Type} = \mathtt{CreateLockedCoins}$
      i. Add $(c'_i, \mathsf{pk}'_i, v'_i, \mathcal{P}')$ to the list Coins.
      ii. Send $(c'_i, \mathsf{pk}'_i, v'_i, \mathcal{P}')$ to $\mathcal{S}_i$ if either $\mathcal{S}_i = \mathcal{C}$, or $\mathcal{P}$ is honest.
5. Remove for every $i \in [n]$, $(c_i, \mathsf{pk}_i, v_i, \star)$ from Coins.
6. Return $(c'_1, \ldots, c'_{n'})$ to $\mathcal{P}$.
7. Send $\{(c_i, \mathsf{pk}_i, v_i, \bot)\}_{i\in[n]}$ $\{(c'_i, \mathsf{pk}'_i, v'_i, \bot)\}_{i\in[n']}$ to $\mathcal{P}_{\mathsf{Auditor}}$.
8. If $\mathtt{Type} = \mathtt{CreateLockedCoins}$ broadcast $\mathtt{Transact}(\mathtt{CreateLockedCoins}, \mathcal{P}', v)$, else broadcast $\mathtt{Transact}(\mathtt{Type}, v)$

Fig. 2: Ideal Functionality.

H Security Proof for Qurrency

In this section, we prove that the Qurrency protocol securely realizes $\mathcal{F}_{\mathsf{pq\text{-}aud\text{-}priv\text{-}cash}}$.

H.1 Simulator Sim

We now describe the simulator strategy for our Qurrency protocol. For every adversary $\mathcal{A}$, we construct a Sim that interacts with the adversary $\mathcal{A}$ and the ideal functionality $\mathcal{F}_{\mathsf{pq\text{-}aud\text{-}priv\text{-}cash}}$ such that no polynomial

time environment $\mathcal{E}$ can distinguish between the real and ideal world. $\mathcal{E}$ provides inputs to the honest parties, and the messages to $\mathcal{A}$. For our work it will suffice to think of the adversary $\mathcal{A}$ as sending the messages provided by $\mathcal{E}$.

For simplicity we assume that each transaction always has exactly n input coins, and n' output coins. If not, the ideal functionality can be modified to broadcast the number of input and output coins involved in each transaction, which can be construed as public leakage from a transaction. We note that in the Qurrency protocol, each transaction includes, in the clear, both the nullifiers corresponding to spent UTXO and the output UTXOs. Thus the number of input and output UTXOs is known to all participants in the system, and can be explicitly captured by the ideal functionality as well. If this is not considered acceptable leakage in certain applications, one can reduce this leakage by padding with dummy transactions, although care has to be taken to build a system that allows for padding with "dummy" nullifiers.

Setup The simulator generates the NIZK public parameters along with the trapdoor, i.e. $(\mathsf{NIZK.pp}, \mathsf{td}) \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$. The data structures are also initialized as in the setup. If $\mathcal{P}_{\mathsf{Auditor}} \in \mathcal{C}$, then obtain apk from $\mathcal{A}$, else the honest party broadcasts apk. Sim also maintains a list CoinUtxo that will contain the mapping between ideal world coins c and real world UTXOs UTXOs.

Adversary messages When Sim receives transactions tx from $\mathcal{A}$,

1. If tx was *not* from `CreateDelegateLockTx`, use the NIZK trapdoor td to extract from the proof π in the transaction. Use the extracted values to check if the contents in the transaction are correctly computed using the witness, and if any of them do not match output $\perp$. If during extaction, Sim extracts a pk that is not registered, Sim sends a $(\texttt{Register}, \mathsf{pk})$ message to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ on behalf of a corrupted party. Next, for each different type of tx, Sim does the following, where the information used to make the query to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ is obtained from tx and the extracted witness from the proof.
   Deposit tx**:** Send $(\texttt{Deposit}, (v, \{v_i, \mathsf{pk}_i\}_{i\in[n]}))$ to $\mathcal{F}_{\text{pq-aud-priv-cash}}$. On receiving $\{c_1, \ldots, c_n\}$ from $\mathcal{F}_{\text{pq-aud-priv-cash}}$ store (utxo'_i, c_i) in CoinUtxo.
   Withdraw tx**:** Send $(\texttt{Transact}, (\texttt{UnlockedTransfer}, v, \{c_i\}_{i\in[n]}, v', \mathsf{pk}))$ to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ where c_i is such that (utxo_i, c_i) in CoinUtxo. On receiving c' from $\mathcal{F}_{\text{pq-aud-priv-cash}}$ store (utxo', c) in CoinUtxo.
   Transfer tx**:** Send $(\texttt{Transact}, (\texttt{UnlockedTransfer}, 0, \{c_i\}_{i\in[n]}, \{v'_i, \mathsf{pk}'_i\}_{i\in[n']}, \perp))$ to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ where c_i is such that (utxo_i, c_i) in CoinUtxo. $\{c_1, \ldots, c_n$ from $\mathcal{F}_{\text{pq-aud-priv-cash}}$ store (utxo'_i, c_i) in CoinUtxo.
   Create Lock tx**:** Send $(\texttt{Transact}, (\texttt{CreateLockedCoins}, 0, \{c_i\}_{i\in[n]}, \{v'_i, \mathsf{pk}'_i\}_{i\in[n']}, \mathcal{P}'))$ to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ where c_i is such that (utxo_i, c_i) in CoinUtxo. $\{c_1, \ldots, c_n$ from $\mathcal{F}_{\text{pq-aud-priv-cash}}$ store (utxo'_i, c_i) in CoinUtxo.
   Transfer Locked tx**:** Send $(\texttt{Transact}, (\texttt{LockedTransfer}, 0, \{c_i\}_{i\in[n]}, \{v'_i, \mathsf{pk}'_i\}_{i\in[n']}, \perp))$ to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ where c_i is such that (utxo_i, c_i) in CoinUtxo. $\{c_1, \ldots, c_n$ from $\mathcal{F}_{\text{pq-aud-priv-cash}}$ store (utxo'_i, c_i) in CoinUtxo.
2. Else, for each utxo in the transaction find the corresponding coin c and send $(\texttt{DelegateLock}, (c, \mathcal{P}))$ where $\mathcal{P}$ corresponds to the Ethereum address in tx.

When Sim receives an "out-of-band" message $(\mathsf{utxo}, v, \rho, \mathsf{pk})$ intended for an honest party, Sim checks that $\mathsf{utxo} = \mathsf{H}(v, \rho, \mathsf{pk})$. If the check passes, find coin c in CoinUtxo that corresponds to utxo and send $(\texttt{InformCoin}, c)$ to $\mathcal{F}_{\text{pq-aud-priv-cash}}$.

Honest messages The environment $\mathcal{E}$ provides inputs to the honest parties. When Sim receives a message from the ideal functionality, it needs to simulate the messages for $\mathcal{A}$. If $\mathcal{P}_{\mathsf{Auditor}}$ is also corrupted, Sim also gets all the details of the transaction.

$\mathcal{P}_{\mathsf{Auditor}}$ **is corrupted:** In this case, for every transaction Sim uses the correct values, the auditor public key apk, and samples fresh randomness to compute the transaction honestly. Note that this can be done since Sim has all the necessary information.

$\mathcal{P}_{\mathsf{Auditor}}$ **is honest:** In this case, whenever $\mathcal{F}_{\text{pq-aud-priv-cash}}$ broadcasts a `Deposit` or `Transact` message Sim prepeares the corresponding tx to send to $\mathcal{A}$. The root is updated as in the honest protocol, and can be done without requiring any private honest input. Next, every honest null and utxo are computed as the hash of a random input to H. Every ciphertext ct is computed as an encryption of 0. Lastly, every NIZK proof is simulated using td.

H.2 Hybrids

We describe below the hybrids to show that for every adversary $\mathcal{A}$, the simulator Sim described above produces indistinguishable view to any environment (or distinguisher) $\mathcal{E}$.

Hybrid 0 - Real World This is the real world execution of the protocol. Here Sim has access to the honest party inputs in the clear, and uses them to emulate the honest behavior of the honest parties.

Hybrid 1 In this hybrid, the only change is that Sim uses trapdoor td and uses td to extract the witnesses from the NIZK proofs sent by $\mathcal{A}$, and check consistency with the tx sent by $\mathcal{A}$. It also checks honest behavior to see if the adversary is able to spend a utxo to which they do not have the corresponding secret key. If any of the consistency checks with extracted value fail, output $\perp$.

Claim. Assuming the proof of knowledge property of NIZK and the collision resistance of H, the probability that $\mathcal{A}$ produces accepting NIZK proof but Sim outputs $\perp$ is negligible.

This follows in a straightforward manner from the proof of knowledge property of the NIZK, as the only way that Sim outputs $\perp$ is if $\mathcal{A}$ used a "different" witness which would only be possible if $\mathcal{A}$ was able to break the collision resistanece of H. Thus the "true" witness, and the "different" witness are the corresponding collision. This claim also implicitly assumes that the secret key is hidden given the public key, a requirement for any secure encyrption, and thus do not state is separately.

Note that conditioned on Sim output not being $\perp$, the Hybrid 1 is identical to Hybrid 0. Since the probability of $\perp$ is only negligible, the hybrids are computationally indistinguishable.

Hybrid 2 In this hybrid, Sim begins to emulate the ideal functionality. In particular, in addition to maintaining the data structures needed to emulate $\mathcal{F}_{\text{pq-aud-priv-cash}}$, it also maintains the list CoinUtxo. On extracting the witnesses for tx from $\mathcal{A}$, it uses the inputs to create an appropriate message to $\mathcal{F}_{\text{pq-aud-priv-cash}}$ as in the description of Sim in Appendix H.1. It then stores (utxo, c) in CoinUtxo where c is the randomly sampled coin corresponding to utxo. Since it still has access to the honest party inputs, in addition to sending the corresponding message to the ideal functionality, Sim also uses the ideal functionality to honestly compute tx.

This hybrid is identical to the previous hybrid since the changes are syntactical since Sim does not output $\perp$.

Hybrid 3 In this hybrid, Sim now uses td to simulate the NIZK proofs in the tx generated on behalf of the honest parties if $\mathcal{P}_{\text{Auditor}}$ is honest. If $\mathcal{P}_{\text{Auditor}}$ is corrupted, continue to use the honest inputs to compute tx.

Claim. Assuming the simulation extractable property of NIZK the change to the hybrid is indistinguishable.

Hybrid 4 In this hybrid, if $\mathcal{P}_{\text{Auditor}}$ is corrupted, there is no change. Otherwise, Sim changes the ciphertext in each tx generated on behalf of the honest party to be an encryption of 0. It is able to do so because it has replaced the NIZK proofs with a simulated proof.

Claim. Assuming the IND-CPA security of the public-key encryption scheme, the change to the hybrid is indistinguishable.

Hybrid 5 In this hybrid, if $\mathcal{P}_{\text{Auditor}}$ is corrupted, there is no change. Otherwise, Sim changes the nullifier in each transfer tx generated on behalf of the honest party to be H applied to a random value. Again, it is able to do so because it has replaced the NIZK proofs with a simulated proof.

Claim. Assuming the hiding property of H, the change to the hybrid is indistinguishable.

Hybrid 6 In this hybrid, if $\mathcal{P}_{\text{Auditor}}$ is corrupted, there is no change. Otherwise, Sim changes the utxo in each transaction tx generated on behalf of the honest party to be H applied to a random value except for utxo that is destined for a corrupted party, which is computed honestly. Note that for utxos that are computed honestly, Sim only needs information from the (emulated) ideal functionality to compute utxo and not the honest inputs. Again, Sim is able to make these changes because it has replaced the NIZK proofs with a simulated proof.

Claim. Assuming the hiding property of H, the change to the hybrid is indistinguishable.

Note that at this point, Sim no longer uses the honest party inputs and is able to simulate the interaction with $\mathcal{A}$ using only the ideal functionality output.

Hybrid 6 - Ideal World Instead of emulating the ideal functionality using the honest party inputs, Sim now directly interacts with the ideal functionality $\mathcal{F}_{\text{pq-aud-priv-cash}}$.

This change is only syntactic, and thus the ideal world is indentical to the previous hybrid.

Quantum Privacy. We note that by instanting the NIZK using a perfect zero-knowledge NIZK such as Groth16 [36], by the post-quantum IND-CPA security of the underlying PKE (ML-KEM [43]) and the post-quantum preimage-resistance of the underlying hash function used to implement UTXOs (Poseidon [35]) we note that the changes made in the hybrid to hide the honest party inputs (from hybrid 2 onwards) are indistinguishable to a quantum adversary, providing post-quantum privacy for Qurrency.