Derecho: Privacy Pools with Proof-Carrying Disclosures

Josh Beal and Ben Fisch

Yale University
{josh.beal,ben.fisch}@yale.edu

Abstract. A *privacy pool* enables clients to deposit units of a cryptocurrency into a shared pool where ownership of deposited currency is tracked via a system of cryptographically hidden records. Clients may later withdraw from the pool without linkage to previous deposits. Some privacy pools also support hidden transfer of currency ownership within the pool. In August 2022, the U.S. Department of Treasury sanctioned Tornado Cash, the largest Ethereum privacy pool, on the premise that it enables illicit actors to hide the origin of funds, citing its usage by the DPRK-sponsored Lazarus Group to launder over \$455 million dollars worth of stolen cryptocurrency. This ruling effectively made it illegal for U.S. persons/institutions to use or accept funds that went through Tornado Cash, sparking a global debate among privacy rights activists and lawmakers. Against this backdrop, we present *Derecho*, a system that institutions could use to request cryptographic attestations of fund origins rather than naively rejecting all funds coming from privacy pools. Derecho is a novel application of *proof-carrying data*, which allows users to propagate allowlist membership proofs through a privacy pool's transaction graph. Derecho is backwards-compatible with existing Ethereum privacy pool designs, adds no significant overhead in gas costs, and costs users only a few seconds to produce attestations.

1 Introduction

Bitcoin, Ethereum, and other cryptocurrencies have achieved significant market capitalization and adoption over the past decade, yet the privacy guarantees of many popular blockchains remain lacking. The traceability of transactions in blockchains such as Bitcoin and Ethereum has been well-studied [AKR+13, MPJ+13, WMW+22], and even privacy-focused blockchains such as Monero are subject to deanonymization attacks [MSH+18]. Privacy solutions can be designed as add-on components to an existing blockchain or as independent blockchains.

In this work, we focus on privacy pools that use *zero-knowledge proofs* to enable anonymous transfers of assets on account-based smart contract platforms such as Ethereum. These pools are based on the design of *Zerocash* [BCG+14], which is also the basis for the cryptocurrency Zcash [HBHW22]. In a nutshell, these privacy pools enable users to deposit funds into a shared pool, anonymously transfer funds within the pool, and later withdraw funds without linkage to their previous transactions.

Tornado Cash (Nova) was the most widely used Ethereum privacy pool until U.S. regulators took action against the service in August 2022. The U.S. Department of the Treasury's Office of Foreign Assets Control (OFAC) added the Tornado Cash smart contract addresses to the Specially Designated Nationals (SDN) list, purportedly due to its usage for laundering more than \$9 billion worth of cryptocurrency since 2019, including by the DPRK state-sponsored Lazarus Group that was also sanctioned in 2019. This designation forbids U.S. users, including individuals and institutions, from interacting with the service [Uni22]. It has resulted in locked funds for U.S. users of the service and has limited the options for law-abiding users that seek to improve the privacy of their transactions on Ethereum. These sanctions have brought renewed attention to the clash between privacy and regulatory oversight on smart contract platforms. In October 2022, Coin Center filed a lawsuit against the Treasury Department arguing that OFAC exceeded its statutory authority in designating Tornado Cash [Coi22]. It also sparked discussion among researchers and privacy advocates [BKB22, Fis22, Sol22], questioning both the efficacy and necessity of privacy pool sanctions in addressing illicit finance, and seeking alternative technical solutions.

A simple solution would restrict deposits into and withdrawals from the privacy pool to accounts on a specific allowlist.[1] For example, the allowlist might be the set of all public Ethereum addresses

[1] Alternatively, the usage of the privacy pool could be limited to accounts that are not on a specific blocklist and not newly generated at the time of deposit. The second criterion is important to prevent the situation where an attacker move funds to a fresh address in order to evade the restrictions.

that are not on the U.S. Treasury's SDN list. However, allowlists are expected to vary by jurisdiction, and may be updated dynamically.

An alternative solution is for users to generate attestations when necessary, selectively disclosing information about the provenance of funds withdrawn from the pool. When cryptocurrency is deposited into a privacy pool like Tornado Cash, a digital receipt in the form of a cryptographic commitment is generated, and the depositor retains a secret key required to use this receipt later. A user withdraws x units of cryptocurrency from the pool by presenting a zero-knowledge proof that it knows the secret key of an unused receipt for this exact amount of cryptocurrency, and a keyed hash of the receipt called a *nullifier*. The nullifier still hides the receipt but prevents it from being used twice. While this zero-knowledge proof reveals little information by default (other than transaction validity), a user could choose to reveal more information about the origin of a withdrawal to an interested party (e.g., an exchange). In fact, zero-knowledge proofs can be used to selectively disclose information about the unique deposit receipt, including membership of the depositing account on an allowlist. Similar solutions were proposed more than a decade ago in the context of Tor and blocklisting of IP-addresses [BG13, TKCS09].

However, this system of user-generated disclosures becomes more challenging in pools that support in-pool transfers. The recipient of funds must retain the ability to prove facts about its provenance, in particular, that the funds originated via deposits from accounts on a given set of allowlists. We solve this problem using *proof-carrying data* [CT10], a generalization of incrementally verifiable computation [Val08] that offers a powerful approach to recursive proof composition. When a user makes their first transaction within the privacy pool, the user generates membership proofs for a set of allowlists. Subsequent transactions within the privacy pool generate new membership proofs that are derived from (i.e., prove knowledge of) the previous membership proofs of the transaction inputs and the details of the current transaction. These membership proofs, which we call *proof-carrying disclosures*, can be verified efficiently and may be communicated directly to the recipient of funds.

1.1 Our Contributions

To summarize, our main contributions are as follows:

- We formalize and present Derecho, a privacy pool with attested transactions based on *proof-carrying disclosures*. Our system addresses the key legal challenges of privacy pools through a novel application of proof-carrying data.
- We show that our system achieves practical proof generation and verification times for a range of system parameters. Since membership proofs are verified off-chain by the recipient, we find that our system is comparable in gas costs to existing Ethereum privacy pools.

1.2 Technical Overview

A key goal in the system design was to develop a solution that can be introduced as an add-on component to existing privacy pools on Ethereum and other smart contract platforms. To facilitate adoption of the system, the design should not require changes to the transaction functionality of the privacy pool contract or introduce any significant gas costs to the users of the contract. Furthermore, it should maintain the existing security properties of the privacy pool while optionally allowing for attestations of allowlist membership.

Derecho assumes the existence of a set of allowlists that are maintained external to the system, where each allowlist contains a list of Ethereum public-key addresses, along with a dynamic accumulator A (e.g., a Merkle tree) which aggregates the allowlists. That is, for each public key pk on allowlist with identifier al the element $H(\mathsf{al}||\mathsf{pk})$ is inserted into A, where H is a collision-resistant hash function. For simplicity we restrict to privacy pools that manage only one cryptocurrency asset at a time, but the system easily generalizes to pools that manage assets of multiple types. The pool contract maintains an accumulator R of records, where each record is a hash digest (i.e., cryptographic commitment). When a user first deposits x units of cryptocurrency into the privacy pool from a public Ethereum address pk_s, a record of the form $H(x||\mathsf{pk}_1||r_1)$ is added to the accumulator R, where pk_1 is a *shielded* public-key address and r is a nonce that will later be used to nullify the record upon a transfer or withdrawal. A transfer transaction may create a new record $H(x||\mathsf{pk}_2||r_2)$, a nullifier $\mathsf{n} = H(r_1)$, and would include a zero-knowledge proof that a record $c = H(x||\mathsf{pk}_1||r_1)$ exists in R

such that $\mathsf{n} = H(r_1)$. The transfer may also create multiple output records of the form $H(x_i||\mathsf{pk}_i||r_i)$ for $i \in [2, k]$, and the zero-knowledge proof would additionally attest that $\sum_i x_i = x$. A withdrawal contains a similar zero-knowledge proof, but publicly reveals the output amount y and a destination Ethereum address pk_d, at which point y units are withdrawn from the pool and delivered to pk_d.

We define membership of records on allowlists recursively as follows. The initial record created upon deposit is a member of allowlist al if and only if its source Ethereum address pk_s is a member of al. A record created as the output of a transfer transaction is a member of al if and only if all the inputs records to the transfer are members of al. Finally, we say that a withdrawal transaction is a member of al if and only if all the input records to this withdrawal are members of al. (Note that this final allowlist attestation refers to the withdrawal transaction itself rather than the Ethereum destination address pk_d, which may or may not be on the allowlist for other reasons.)

Since the initial record created upon deposit is publicly linked to the Ethereum source address pk_s via the on-chain deposit transaction, it is straightforward for a user to produce a membership proof of the deposit record on a list al by providing a membership proof for pk_s using the accumulator A, which could be verified given the Ethereum transaction log. Producing membership disclosure proofs for the output records of transactions is more subtle. If a user already has membership proofs for all the input records to a transfer transaction with respect to a list al, then it can create a membership proof for an output record of this transaction by proving its knowledge of valid al membership proofs for all the input records to the transaction. The same could be done for a withdrawal transaction. In more detail, since neither the output record nor the transaction log contains explicit references linking it to transaction inputs, but only nullifiers n_i for each input record, the zero-knowledge disclosure proof repeats the logic of the transfer proof: for each n_i, it proves knowledge of an input record $c_i = H(x_i||\mathsf{pk}_i||r_i)$ such that $n_i = H(r_i)$ and *additionally* knowledge of a valid membership proof π_i for c_i. This recursive proof of knowledge is possible via a proof-carrying data (PCD) scheme.

However, a problem immediately arises: the validity of π_i is not actually verifiable against the record commitment c_i alone. For example, verifying the initial membership proof of a deposit record c required checking against the blockchain transaction log to obtain the link between the record c and a source Ethereum address pk_s. Naively, if the public input required to verify a membership includes the entire blockchain transaction log then the recursive zero-knowledge proof statement would become impractically large. The standard trick around this problem is to replace the transaction log public input with an accumulator digest T: the membership disclosure proof for a deposit record c now includes both an accumulator membership proof for T of a transaction linking c to pk_s and an accumulator membership proof for A showing pk_s is on the list al.

However, yet another subtle complication arises when attempting to produce recursive membership proofs for the output records of transfers. Suppose the user has a membership proofs π for an input record c to a transfer creating an output record c'. Suppose further that the accumulator digest T commits to the transaction log state at the time t that π was created, and that the accumulator state T' commits to the transaction log state at the time t' that the new transfer is occurring. The value T is required as input to verify π, but is unknown to the recipient of the transfer at the time t'. Thus, T is not available as a public input to verify the recursive disclosure created for c', rather, the disclosure must prove knowledge of both π, c and T against which π is valid. Moreover, without additional restrictions, the prover would be free to invent a malicious proof π^* valid against a T^* unrelated to the true blockchain state at any point in history.

To resolve this problem we use history accumulators, which commit not only to the current state of a set but also all historical states. History accumulators provide an efficient mechanism to prove that a digest T represents a valid historical state σ, which can be verified against the current digest T' of the history accumulator. Altogether, these techniques result in a system that only requires a few seconds to produce attestations on a consumer-grade laptop. These attestations can be efficiently verified by the recipient of funds with respect to the current blockchain state. Due to the fact that these proofs do not need to be posted on-chain or verified by the smart contract, we were able to leverage recent developments in PCD that trade a larger proof size for very fast proving times [BCL+21].

1.3 Related Work

Sander and Ta-Shma [STS99] and Camenisch et al. [CHL06] established the foundations of accountable privacy for ecash systems. With the growing popularity of cryptocurrencies, several works have examined trade-offs between privacy and accountability/auditability in the design of decentralized payment systems. Garman et al. [GGM16] demonstrates how to add privacy-preserving policy enforcement mechanisms to the Zerocash design. UTT [TBA+22] designs a decentralized payment system that limits the the amount of currency sent per month using the notion of an anonymity budget. Platypus [WKDC22] and PEReDi [KKS22] explore the design of central bank digital currencies (CBDCs) with privacy-preserving regulatory functionality. Platypus [WKDC22] focuses on enforcement of anonymity budgets and total balance limits. PEReDi [KKS22] supports compliance with regulations such as Know Your Customer (KYC), Anti Money Laundering (AML), and Combating Financing of Terrorism (CFT). Their system aims to avoid a single point of failure by distributing the policy enforcement mechanism. CAP [Esp22] introduces Configurable Asset Privacy schemes, which support private transfers of heterogeneous assets with custom viewing and freezing policies. ZEBRA [RPX+22] develops anonymous credentials that support auditability and revocation while enabling efficient on-chain verification. We refer to [CBC21] for a more detailed study of these research challenges.

ZEXE [BCG+20] provides a general framework for privacy-preserving blockchain applications in which the application state is a system of records, transactions create and nullify records, and all records have birth and death predicates defining the conditions under which they can be created or nullified. Transactions contain zero-knowledge proofs that these predicates are satisfied. As the authors note, this captures membership proofs of records on allowlists/blocklists as a special case (described in detail through a "regulation-friendly private stablecoin" example). In terms of comparison to *Derecho*, the ZEXE regulation-friendly stablecoin example restricts users of the stablecoin to a single allowlist (or blocks users on a single blocklist), represented as a credential assigned to the public key address of a user, while Derecho does not alter the functionality of privacy-preserving cryptocurrencies, enabling users to separately disclose allowlist provenance off-chain. Unlike Derecho, ZEXE does not address how users can prove statements about the origin of records within a hidden transaction graph, nor the added challenge that the users themselves cannot see the full details of transaction history aside from allowlist membership proofs of their existing records.

Proof-carrying data [CT10] (PCD) generalizes the notion of incrementally verifiable computation [Val08] (IVC) from sequential computation to distributed computation over a directed acyclic graph. The initial paper proposing PCD proposed several applications to the integrity of distributed computations, including distributed program analysis, type safety, IT supply chains, and conjectured applications to financial systems. Naveh and Tromer [NT16] proposed an application of PCD to image authentication, i.e., proving the authenticity of photos even after they have been edited according to a permissible set of transformations (e.g., cropping, rotation, scaling), which would invalidate signatures on the original image data. PCD (and IVC as a special case) has been used to construct authenticated data structures with richer invariants, such as append-only dictionaries [TFZ+22] and incrementally verifiable ledger systems [CCDW20, BMRS20].

2 Building Blocks

2.1 Preliminaries

Notation We let $\lambda \in \mathbb{N}$ denote the security parameter with unary representation 1^λ. We let $\mathsf{negl}(\lambda)$ and $\mathsf{poly}(\lambda)$ denote the classes of negligible functions and polynomial functions, respectively. We let PPT denote probabilistic polynomial time. We let $\mathcal{A}$ denote a computationally-bounded adversary modeled as a PPT algorithm. We let $[l]$ denote the set of integers $\{0, \ldots, l-1\}$. We let $x \leftarrow\!\!\$\ S$ denote that x is sampled uniformly at random from a set S. We let $\mathbb{F}_q$ denote a finite field of order q.

Hash functions We use hash functions satisfying the collision resistance property defined in A.2 of [Esp22]. Our system samples hash functions of the form $H_q : \{0,1\}^* \to \mathbb{F}_q$. In this work, we use the arithmetic hash function Poseidon [GKR+21]. The design of arithmetic hash functions is an active area of research [AAB+20, GHR+22, GKL+22, BBC+22], and our system can be instantiated with any efficient construction of these hash functions.

Commitment schemes A commitment scheme $\mathcal{C} = (\mathsf{Com}, \mathsf{Vfy})$ is a pair of efficient algorithms defined over a message space $\mathcal{M}$ and a randomness space $\mathcal{R}$ where:

- $\mathsf{cm} \leftarrow \mathsf{Com}(\mathsf{m}; r)$ is a commit algorithm that produces a commitment cm given the message $\mathsf{m} \in \mathcal{M}$ to be committed and the randomness $r \leftarrow \mathcal{R}$.

– $b \leftarrow \mathsf{Vfy}(\mathsf{cm}, \mathsf{m}, r)$ is a verification algorithm that checks whether (m, r) is the correct opening of the commitment cm and outputs a bit $b \in \{0,1\}$ representing accept if $b = 1$ and reject otherwise.

Informally, a commitment scheme is called binding if it is infeasible to open a commitment to a different message. It is called hiding if the commitments of any two messages are indistinguishable. Formal definitions of the binding and hiding properties can be found in A.9 and A.10 of [Esp22]. Commitment schemes can be built from collision-resistant hash functions.

Public-key encryption schemes A public-key encryption scheme is of a triple of efficient algorithms $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where:

– $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ is a PPT key generation algorithm that outputs a key pair consisting of a public key pk and a private key sk. The public key defines a message space $\mathcal{M}_{\mathsf{pk}}$.
– $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{msg})$ is a PPT encryption algorithm that outputs a ciphertext ct when given a public key pk and a message $\mathsf{msg} \in \mathcal{M}_{\mathsf{pk}}$.
– $\mathsf{msg} \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$ is a polynomial-time decryption algorithm that given a ciphertext ct and the secret key sk whose corresponding public key pk was used to generate the ciphertext, outputs the encrypted message in plaintext. The output msg is a special reject value if decryption failed.

We require that $\Pr[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, \mathsf{msg})) = \mathsf{msg}] = 1$ for all key pairs and messages. We require that the scheme has the IND-CPA and IK-CPA properties, which are defined in A.7 and A.8 of [Esp22].

Accumulator schemes An accumulator scheme consists of a tuple of efficient algorithms $\mathsf{Acc} = (\mathsf{Init}, \mathsf{Update}, \mathsf{PrvMem}, \mathsf{VfyMem})$ where:

– $(\mathsf{rt}, \sigma) \leftarrow \mathsf{Init}(1^\lambda)$ sets up the initial state σ and digest rt of the accumulator.
– $(\mathsf{rt}', \sigma') \leftarrow \mathsf{Update}(\mathsf{rt}, \sigma, \mathsf{elem})$ inserts an element elem into the set and outputs an updated state σ' and digest rt'.
– $\pi \leftarrow \mathsf{PrvMem}(\sigma, \mathsf{elem})$ outputs a set membership proof π for the element elem in the set.
– $b \leftarrow \mathsf{VfyMem}(\mathsf{rt}, \pi, \mathsf{elem})$ outputs a bit $b \in \{0,1\}$ verifying whether π is a valid proof for the accumulation of elem in rt. The output is $b = 1$ if elem was accumulated and $b = 0$ otherwise.

History accumulator schemes A history accumulator is an authenticated data structure that commits to a current set state σ_n and also to all previous set states $\sigma_1, ..., \sigma_{n-1}$. When the accumulator digest rt_n for σ_n is incrementally updated for a new state σ_{n+1}, the new digest rt_{n+1} is a commitment to σ_{n+1} and all prior states accumulated by rt_n. Some history accumulators support history proofs of additional invariants, e.g., that the current state σ' of a history accumulator with digest rt' is a superset of all historical states. Specifically, a history accumulator scheme consists of a tuple of efficient algorithms $\mathsf{HA} = (\mathsf{Init}, \mathsf{Update}, \mathsf{PrvMem}, \mathsf{VfyMem}, \mathsf{PrvHist}, \mathsf{VfyHist})$ where the algorithms Init, Update, PrvMem and VfyMem work as above, and the algorithms $\mathsf{PrvHist}$ and $\mathsf{VfyHist}$ work as follows:

– $\pi \leftarrow \mathsf{PrvHist}(\mathsf{rt}, \sigma')$ given the current state σ' (which has a current digest rt') outputs a proof π that rt is a historical state of the history accumulator.
– $b \leftarrow \mathsf{VfyHist}(\mathsf{rt}, \mathsf{rt}', \pi)$ outputs a bit $b \in \{0,1\}$.

This scheme can be instantiated by Merkle history trees [CW09, MKL+20, BKLZ20, TFZ+22], which are known to support efficient membership proofs and history proofs [Cro10, LLK13, MKL+20].

zk-SNARKs A preprocessing zk-SNARK(zero-knowledge succinct non-interactive arguments of knowledge) with universal SRS (structured reference string) consists of a tuple of efficient algorithms $\mathsf{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ where:

– $\mathsf{srs} \leftarrow \mathcal{G}(1^\lambda, N)$ is a PPT generation algorithm that samples an SRS that supports indices of size up to N. This is the universal setup, which is carried out once and used across all future circuits.
– $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathcal{I}_{\mathsf{srs}}(i)$ is a polynomial-time indexing algorithm that outputs the proving key ek and verification key vk for a circuit with description i. This algorithm has oracle access to the SRS.
– $\pi \leftarrow \mathcal{P}(\mathsf{ek}, x, w)$ is a PPT proving algorithm that outputs the proof given the instance x and the witness w.

– $b \leftarrow \mathcal{V}(\mathsf{vk}, x, \pi)$ is a polynomial-time verification algorithm that outputs an accepting bit $b \in \{0,1\}$ given the verification key vk, the instance x, and a proof π. The bit $b = 1$ denotes acceptance of the proof for the instance, while $b = 0$ denotes rejection of the proof.

We require the standard security properties of completeness, knowledge soundness, zero knowledge, and succinctness. We additionally require (updatable) simulation extractability to ensure non-malleability of proofs. We refer to [GKK⁺22] for a formal definition of this property.

Proof-Carrying Data Proof-carrying data (PCD) [CT10] enables a set of parties to carry out an arbitrarily long distributed computation where every step is accompanied by a proof of correctness.

Let $V(\mathsf{G})$ and $E(\mathsf{G})$ denote the vertices and edges of a graph G. A transcript T is a directed acyclic graph where each vertex $u \in V(\mathsf{T})$ is labeled by local data $z_{\mathsf{loc}}^{(u)}$ and each edge $e \in E(\mathsf{T})$ is labeled by a message $z^{(e)} \neq \bot$. The output of a transcript T, denoted $\mathsf{o}(\mathsf{T})$, is $z^{(e')}$ where $e' = (u, v)$ is the first edge such that v is a sink in the lexicographic ordering of the edges.

A vertex $u \in V(\mathsf{T})$ is φ-compliant for a predicate $\varphi \in \mathsf{F}$ if for all outgoing edges $e = (u, v) \in E(\mathsf{T})$ either: (1) if u has no incoming edges, $\varphi(z^{(e)}, z_{\mathsf{loc}}^{(u)}, \bot, \ldots, \bot)$ evaluates to true or (2) if u has m incoming edges $e_1, ..., e_m$, $\varphi(z^{(e)}, z_{\mathsf{loc}}^{(u)}, z^{(e_1)}, \ldots, z^{(e_m)})$ evaluates to true. A transcript T is φ-compliant if all of its vertices are φ-compliant.

A proof-carrying data system PCD for a class of compliance predicates F consists of a tuple of efficient algorithms $(\mathbb{G}, \mathbb{I}, \mathbb{P}, \mathbb{V})$, known as the generator, indexer, prover, and verifier algorithms, for which the properties of completeness, knowledge soundness, and zero knowledge hold.

Completeness. PCD has perfect completeness if for every adversary $\mathcal{A}$ the following holds:

$$\Pr\left[\begin{array}{c} \varphi \in \mathsf{F} \\ \wedge\ \varphi(z, z_{\mathsf{loc}}, z_1, \ldots, z_m) = 1 \\ \wedge\ (\forall i, z_i = \bot \vee \forall i, \mathbb{V}(\mathsf{ivk}, z_i, \pi_i) = 1) \\ \Downarrow \\ \mathbb{V}(\mathsf{ivk}, z, \pi) = 1 \end{array} \middle| \begin{array}{c} \mathsf{pp}_{\mathsf{pcd}} \leftarrow \mathbb{G}(1^\lambda) \\ (\varphi, z, z_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{pcd}}) \\ (\mathsf{ipk}, \mathsf{ivk}) \leftarrow \mathbb{I}(\mathsf{pp}_{\mathsf{pcd}}, \varphi) \\ \pi \leftarrow \mathbb{P}(\mathsf{ipk}, z, z_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \end{array}\right] = 1.$$

Knowledge soundness. PCD has knowledge soundness with respect to an auxiliary input distribution $\mathcal{D}$ if for every expected polynomial-time adversary $\tilde{\mathbb{P}}$ there exists an expected polynomial-time extractor $\mathbb{E}_{\tilde{\mathbb{P}}}$ such that for every set Z:

$$\Pr\left[\begin{array}{c} \varphi \in \mathsf{F} \\ \wedge\ (\mathsf{pp}_{\mathsf{pcd}}, \mathsf{ai}, \varphi, \mathsf{o}(\mathsf{T}), \mathsf{ao}) \in Z \\ \wedge\ \mathsf{T} \text{ is } \varphi\text{-compliant} \end{array} \middle| \begin{array}{c} \mathsf{pp}_{\mathsf{pcd}} \leftarrow \mathbb{G}(1^\lambda) \\ \mathsf{ai} \leftarrow \mathcal{D}(\mathsf{pp}_{\mathsf{pcd}}) \\ (\varphi, \mathsf{T}, \mathsf{ao}) \leftarrow \mathbb{E}_{\tilde{\mathbb{P}}}(\mathsf{pp}_{\mathsf{pcd}}, \mathsf{ai}) \end{array}\right]$$

$$\geq \Pr\left[\begin{array}{c} \varphi \in \mathsf{F} \\ \wedge\ (\mathsf{pp}_{\mathsf{pcd}}, \mathsf{ai}, \varphi, \mathsf{o}, \mathsf{ao}) \in Z \\ \wedge\ \mathbb{V}(\mathsf{ivk}, \mathsf{o}, \pi) = 1 \end{array} \middle| \begin{array}{c} \mathsf{pp}_{\mathsf{pcd}} \leftarrow \mathbb{G}(1^\lambda) \\ \mathsf{ai} \leftarrow \mathcal{D}(\mathsf{pp}_{\mathsf{pcd}}) \\ (\varphi, \mathsf{o}, \pi, \mathsf{ao}) \leftarrow \tilde{\mathbb{P}}(\mathsf{pp}_{\mathsf{pcd}}, \mathsf{ai}) \\ (\mathsf{ipk}, \mathsf{ivk}) \leftarrow \mathbb{I}(\mathsf{pp}_{\mathsf{pcd}}, \varphi) \end{array}\right] - \mathsf{negl}(\lambda).$$

Zero knowledge. PCD has (statistical) zero knowledge if there exists a PPT simulator $\mathcal{S}$ such that for every honest adversary $\mathcal{A}$ the distributions below are statistically indistinguishable:

$$\left\{ (\mathsf{pp}_{\mathsf{pcd}}, \varphi, z, \pi) \middle| \begin{array}{c} \mathsf{pp}_{\mathsf{pcd}} \leftarrow \mathbb{G}(1^\lambda) \\ (\varphi, z, z_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{pcd}}) \\ (\mathsf{ipk}, \mathsf{ivk}) \leftarrow \mathbb{I}(\mathsf{pp}_{\mathsf{pcd}}, \varphi) \\ \pi \leftarrow \mathbb{P}(\mathsf{ipk}, z, z_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \end{array} \right\} \text{and} \left\{ (\mathsf{pp}_{\mathsf{pcd}}, \varphi, z, \pi) \middle| \begin{array}{c} (\mathsf{pp}_{\mathsf{pcd}}, \tau) \leftarrow \mathbb{S}(1^\lambda) \\ (\varphi, z, z_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{pcd}}) \\ \pi \leftarrow \mathbb{S}(\tau, \varphi, z) \end{array} \right\}$$

An adversary is honest if their output results in the implicant of the completeness condition being satisfied with probability 1, i.e., $\varphi \in \mathsf{F}$, $\varphi(z, z_{\mathsf{loc}}, z_1, \ldots, z_m) = 1$, and either $z_i = \bot$ or $\mathbb{V}(\mathsf{ivk}, z_i, \pi_i) = 1$ for each incoming edge z_i. A proof π has size $\mathsf{poly}(\lambda, |\varphi|)$; that is, the proof size is not allowed to grow with each application of the prover algorithm $\mathbb{P}$.

Our system uses the PCD construction of [BCL⁺21], which is based on split accumulation schemes. Other constructions are described in [BCCT13], [BCTV14], Halo [BGH19], [BCMS20], Fractal [COS20], and Halo Infinite [BDFG21]. Nova [KST22] is a recent IVC construction based on folding schemes, however we require the more general notion of PCD to support multiple transaction inputs.

3 Definitions

3.1 System Components

Our system consists of accounts, allowlists, clients, and the privacy pool contract.

- **Account**. An externally-owned account controls units of a cryptocurrency and has a corresponding public/private key pair. For simplicity, we refer to externally-owned accounts as accounts.
- **Allowlist**. A list of accounts that are not prohibited from financial interactions in certain settings, such as a geographic jurisdiction. This list is managed by a trusted party and updated regularly.
- **Client**. A client operates one or more accounts on the Ethereum blockchain and interacts with the privacy pool through cryptocurrency deposits, transfers, and withdrawals. The client generates membership proofs on a set of allowlists when transferring or withdrawing funds.
- **Privacy Pool Contract**. The privacy pool contract supports deposits and withdrawals of coins from the pool and anonymous transfers of coins within the pool. The contract stores allowlists to support membership proofs.

The following data structures are used in our system:

- **Public Parameters**. In the setup of the system, a trusted party generates public parameters pp that are available to all participants in the system.
- **User Key Pairs**. A user generates a key pair $(\mathsf{sk}, \mathsf{pk})$ when joining the privacy pool. The public key pk is used for receiving coins and the secret key sk is used for creating transactions. The public key is derived from the user's Ethereum address addr and the generated secret key. The user generates a key pair $(\mathsf{sk}', \mathsf{pk}')$ for encryption and decryption of owner memos.
- **Account List**. The user's public keys are stored in the account list $\mathsf{AccountList}$ of the privacy pool contract upon registration. This account list supports the anonymous transfer functionality.
- **Coin Commitments**. A coin commitment $\mathsf{cm} := \mathsf{Com}(\mathsf{amt}, \mathsf{pk}; r)$ is a commitment to an amount amt and a user's public key pk using randomness r. The opening of the coin commitment $\mathsf{open} := (\mathsf{amt}, \mathsf{pk}, r)$ is used in transaction creation.
- **Nullifier Sets**. A nullifier set $\mathsf{NullifierList}$ is used to prevent double-spending attacks. A nullifier null can be constructed from an opening of a coin commitment.
- **Owner Memos**. An owner memo memo is used by the coin owner to create the coin commitment from the encryption of the opening of the commitment. It can be shared with the recipient by posting the memo on the public ledger as part of the transaction or by sending the memo through a private communication channel.
- **Allowlists**. An allowlist consists of a unique identifier al and a set of authorized addresses $\mathsf{AuthAddressList}$.
- **Membership Proof Lists**. A membership proof list π is a set of membership proofs for a coin commitment. Each membership proof asserts membership on a specific allowlist in the system.
- **Membership Declarations**. A membership declaration $\mathsf{decl} := H_q(\mathsf{al}||\mathsf{pk})$ is a public reference to an allowlist identifier al and a user's public key pk.
- **Deposit Records**. A deposit record $\mathsf{rec}_{\mathsf{dep}} := H_q(\mathsf{amt}||\mathsf{pk}||\mathsf{cm}||\mathsf{uid})$ is a public record for a deposit into the privacy pool that is derived from the value amount amt, the user's public key pk, the coin commitment cm generated upon deposit, and the unique identifier uid of the deposit transaction.
- **Transfer Records**. A transfer record $\mathsf{rec}_{\mathsf{tfr}} := H_q(\mathsf{null}||\mathsf{cm})$ is a public record for a pool transfer that is derived from the nullifier null for a transaction input and the coin commitment cm for a transaction output. A transfer record is generated for each input-output pair.
- **Accumulators**. Our system uses sparse Merkle trees to efficiently prove set membership. An on-chain accumulator with digest rt_{c} maintains the set of published coin commitments.
- **History Accumulators**. Our system uses sparse Merkle history trees to efficiently prove set membership and ensure consistency of PCD messages. There are three history accumulators: the membership declaration history accumulator with digest $\mathsf{rt}_{\mathsf{id}}$, the deposit record history accumulator with digest $\mathsf{rt}_{\mathsf{dep}}$, and the transfer record history accumulator with digest $\mathsf{rt}_{\mathsf{tfr}}$. These history accumulators are maintained off-chain using public information derived from the blockchain state.

The client supports the following operations:

- $\mathsf{GenerateKeyPair}(\mathsf{pp}, \mathsf{addr}) \rightarrow (\mathsf{sk}, \mathsf{pk}, \mathsf{sk}', \mathsf{pk}')$. Given public parameters pp and an address addr, output a user key pair $(\mathsf{sk}, \mathsf{pk})$ and an encryption key pair $(\mathsf{sk}', \mathsf{pk}')$.

- $\mathsf{CreateDepositTx}(\mathsf{pp}, \mathsf{amt}, \mathsf{pk}) \to \mathsf{tx}_{\mathsf{dep}}$. Given public parameters pp, a value amount amt, and a user public key pk, output a deposit transaction $\mathsf{tx}_{\mathsf{dep}}$. The deposit transaction will result in the creation and accumulation of deposit records and will transfer amt units of value from the sender to the privacy pool contract.
- $\mathsf{CreateTransferTx}_{n,m}(\dots) \to \mathsf{tx}_{\mathsf{tfr}}$. Given public parameters pp, a list of input user secret keys $\mathbf{sk_{in}}$, a list of openings of input coin commitments $\mathbf{open_{in}}$, a list of input addresses $\mathbf{addr_{in}}$, a list of openings of output coin commitments $\mathbf{open_{out}}$, and a list of encryption public keys $\mathbf{pk'}$, output a transfer transaction $\mathsf{tx}_{\mathsf{tfr}}$. The transfer transaction will transfer value from the input coin owners to the output coin owners while ensuring that the input coins can no longer be spent after the transaction is executed. The transfer transaction will result in the creation and accumulation of transfer records for each of the transaction outputs. This algorithm is parametrized by the number of transaction inputs n and the number of transaction outputs m.
- $\mathsf{CreateWithdrawalTx}_{n}(\dots) \to \mathsf{tx}_{\mathsf{wdr}}$. Given public parameters pp, a list of sender secret keys $\mathbf{sk_{in}}$, a list of openings of input coin commitments $\mathbf{open_{in}}$, a list of input addresses $\mathbf{addr_{in}}$, an opening of a (placeholder) output coin commitment $\mathsf{open}_{\mathsf{out}}$, and an output address $\mathsf{addr}_{\mathsf{out}}$, output a withdrawal transaction $\mathsf{tx}_{\mathsf{wdr}}$. The withdrawal transaction will transfer amt units of value from the input coins to the output address. The withdrawal transaction will result in the creation and accumulation of transfer records for each of the transaction outputs. To maintain compatibility with the compliance predicate for membership proofs, a single output coin commitment is generated. However, this placeholder coin commitment is not accumulated, so it cannot be spent. This algorithm is parametrized by the number of transaction inputs n.
- $\mathsf{CreateMembershipProof}_{n,m}(\dots) \to (\boldsymbol{z}_{\mathbf{out}}, \boldsymbol{\pi}_{\mathbf{out}})$. Given the arguments to the transfer/withdrawal operation, a list of membership declarations $\mathbf{decl}$, a list of membership witnesses for the membership declarations $\mathbf{w_{id}}$, a list of history proofs for the membership declaration history accumulator $\mathbf{c_{id}}$, a list of deposit records $\mathbf{rec_{dep}}$, a list of unique deposit identifiers $\mathbf{uid}$, a list of membership witnesses for the deposit records $\mathbf{w_{dep}}$, a list of history proofs for the deposit record history accumulator $\mathbf{c_{dep}}$, a list of transfer records $\mathbf{rec_{tfr}}$, a list of membership witnesses for the transfer records $\mathbf{w_{tfr}}$, a list of history proofs for the transfer record history accumulator $\mathbf{c_{tfr}}$, a list of input PCD message lists $\boldsymbol{z}_{\mathbf{in}}$, and a list of input PCD proof lists $\boldsymbol{\pi}_{\mathbf{in}}$, output a list of PCD message lists $\boldsymbol{z}_{\mathbf{out}}$ and a list of PCD proof lists $\boldsymbol{\pi}_{\mathbf{out}}$. A membership proof list is generated for each of the output coins by the prover algorithm in the PCD scheme. This algorithm is parametrized by the number of transaction inputs n and the number of transaction outputs m.
- $\mathsf{CreateRegistrationTx}(\mathsf{pp}, \mathsf{pk}, \mathbf{al}) \to \mathsf{tx}_{\mathsf{reg}}$. Given public parameters pp, a user public key pk, and a set of allowlists $\mathbf{al}$, output a registration transaction $\mathsf{tx}_{\mathsf{reg}}$. This transaction will validate the user's membership for each of the allowlists and create membership declarations.

The privacy pool supports the following operations:

- $\mathsf{PrivacyPoolSetup}(1^{\lambda}) \to \mathsf{pp}$. This algorithm sets up the initial state of the system, including the accumulators and the configurable parameters. Returns the system's public parameters pp.
- $\mathsf{ProcessDepositTx}(\mathsf{pp}, \mathsf{tx}_{\mathsf{dep}}) \to (b, \mathsf{rec}_{\mathsf{dep}}, \mathsf{uid})$. This algorithm validates the deposit amount, verifies the deposit proof, creates and accumulates the deposit record, generates a unique identifier, and transfers funds from the sender address to the privacy pool contract address. Returns accept/reject bit b, deposit record $\mathsf{rec}_{\mathsf{dep}}$, and unique identifier uid for the deposit transaction. The deposit record $\mathsf{rec}_{\mathsf{dep}}$ and the unique identifier uid are used in the initial round of membership proof generation.
- $\mathsf{ProcessTransferTx}_{n,m}(\mathsf{pp}, \mathsf{tx}_{\mathsf{tfr}}) \to (b, \mathsf{rec}_{\mathsf{tfr}})$. This algorithm checks the value invariant and verifies the transfer proof. If the transaction is valid, input nullifiers are added to the nullifier set, output coin commitments are added to the coin commitment accumulator, and the transfer record is created and accumulated. Returns accept/reject bit b and the transfer record $\mathsf{rec}_{\mathsf{tfr}}$. The transfer record is used in a round of membership proof generation.
- $\mathsf{ProcessWithdrawalTx}_{n}(\mathsf{pp}, \mathsf{tx}_{\mathsf{wdr}}) \to (b, \mathsf{rec}_{\mathsf{tfr}})$. This algorithm validates the input coin commitments and verifies the withdrawal proof. If the transaction is valid, input nullifiers are added to the nullifier set, the transfer record is created and accumulated, and funds are sent from the privacy pool to the recipient. Returns accept/reject bit b and the transfer record $\mathsf{rec}_{\mathsf{tfr}}$.
- $\mathsf{ProcessRegistrationTx}(\mathsf{pp}, \mathsf{tx}_{\mathsf{reg}}) \to (b, \mathbf{decl})$. The contract will store the public key pk for the user in the account list. The contract verifies that the sender is authorized to declare membership on the allowlists in the set $\mathbf{al}$. Returns accept/reject bit b and a list of membership declarations $\mathbf{decl}$.

3.2 System Goals

The security goals of privacy pools consist of correctness, availability, confidentiality, and unlinkability.

Correctness ensures that a pool does not allow clients to spend coins that have already been spent or that they do not own. Availability ensures that clients cannot be prevented from using the privacy pool. Once coin commitments have been added to the contract state, clients cannot be prevented from spending coins that they own and have not previously spent. Confidentiality and unlinkability are the key privacy considerations. A pool ensures confidentiality of transactions if only the sender and recipient learn the value amount associated with each transaction. A pool ensures unlinkability of transfers and withdrawals if an adversary has a negligible advantage in guessing an input coin commitment associated with a given transfer or withdrawal transaction.

Derecho does not alter the functionality of the privacy pool and thus preserves these correctness and privacy properties. However, we also need to define additional correctness and privacy goals for *proof-carrying disclosures*. Correctness ensures that a client cannot attest membership of a transaction output on a given allowlist unless each of the transaction inputs has an attestation of membership on this allowlist or is a deposit from an address that is registered on this allowlist. Privacy ensures that the allowlist membership proof does not reveal anything besides the allowlist membership of the transaction output (e.g., it does not reveal transaction details).

In our construction, correctness will follow from the definition of the compliance predicate and privacy from the zero-knowledge property of the underlying PCD scheme.

4 Construction

4.1 Building Block Algorithms

- **Commitment Creation**. A commitment is computed using a hash function that is applied to the input elements and randomness. For a coin with value amt owned by public key pk, the coin commitment cm is computed by $\mathsf{Com}(\mathsf{amt}, \mathsf{pk}; r) := H_q(\mathsf{amt} \| \mathsf{pk} \| r)$. We may also write $\mathsf{cm} := \mathsf{Com}(\mathsf{open})$ for the opening $\mathsf{open} = (\mathsf{amt}, \mathsf{pk}, r)$.
- **Nullifier Creation**. A nullifier is computed using a hash function that is applied to the opening of the coin commitment and the user's secret key sk. The nullifier for a coin commitment with opening $\mathsf{open} = (\mathsf{amt}, \mathsf{pk}, r)$ is computed by $\mathsf{Nullify}(\mathsf{open}) := H_q(r)$.
- **Memo Encryption**. $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m; r)$ denotes an ElGamal encryption algorithm that computes ciphertext ct from public key pk, message m, and randomness r.
- **Membership Declaration Creation**. For an allowlist al and public key pk, the membership declaration is computed by $\mathsf{decl} := H_q(\mathsf{al} \| \mathsf{pk})$.
- **Deposit Record Creation**. A deposit record is computed using a hash function that is applied to the user's public key pk, the value amount amt, the coin commitment cm generated upon deposit, and the unique identifier uid of the deposit transaction. The deposit record is computed by $\mathsf{rec}_{\mathsf{dep}} := H_q(\mathsf{pk} \| \mathsf{amt} \| \mathsf{cm} \| \mathsf{uid})$ without reference to any private values.
- **Transfer Record Creation**. A transfer record is computed using a hash function that is applied to an input nullifier null and an output coin commitment cm. The transfer record is computed by $\mathsf{rec}_{\mathsf{tfr}} := H_q(\mathsf{null} \| \mathsf{cm})$ without reference to any private values.

4.2 Recursive Membership Proof

This section defines the PCD system for attestations of allowlist membership for a transaction output. Let n be the number of transaction inputs, m be the number of transaction outputs, and l be the number of allowlists. For simplicity, we fix the set of allowlists $(\mathsf{al}_j)_{j \in [l]}$ to yield a set of compliance predicates $(\varphi_j)_{j \in [l]}$. We refer to section 2.1 for a complete description of proof-carrying data.

The compliance predicate φ_j is a function of the message z, the local data z_{loc}, and the incoming messages $(z_i)_{i \in [n]}$. Each compliance predicate φ_j is defined with respect to a specific allowlist al_j. A message z consists of public data associated with the transaction output: the input nullifiers, the output coin commitment, and auxiliary data related to the deposit records, transfer records, and membership declarations. The local data z_{loc} for the message consists of private data associated with the transaction output: input and output coin commitment openings, secret keys for the transaction inputs, membership witnesses for the accumulated elements (i.e., deposit records, transfer records, and membership declarations), and proofs for the history accumulator digests.

Each transfer transaction corresponds to a vertex in the proof-carrying data graph G. This vertex typically has n incoming edges and m outgoing edges. However, this vertex has no incoming edges when all transaction inputs consist of fresh deposits to the privacy pool. For a node with incoming edges, each message z_i corresponds to a message that was generated as the output of a previous transaction. If the node has no incoming edges, $z_i = \perp$. A vertex u is φ_j-compliant if for all outgoing edges with message z either: (1) if u has no incoming edges, $\varphi_j(z, z_{\mathsf{loc}}, \perp, \ldots, \perp)$ evaluates to true or (2) if u has n incoming edges, $\varphi_j(z, z_{\mathsf{loc}}, z_1, \ldots, z_n)$ evaluates to true. Note that $z_i = \perp$ or $\mathbb{V}(\mathsf{ivk}, z_i, \pi_i) = 1$ for each incoming edge z_i. The prover generates an output proof $\pi = \mathbb{P}(\mathsf{ipk}, z, z_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m)$.

If a vertex has no incoming edges, this indicates that each transaction input is a coin that was generated upon deposit to the pool. This is the base case of the compliance predicate. In this case, the predicate performs a series of checks for each transaction input. The predicate checks that the deposit record is correctly computed from the public data (i.e, the value amount, the user's public key, the deposit coin commitment, and the unique identifier of the deposit transaction) and verifies that the deposit record is accumulated. The predicate checks that the membership declaration is correctly computed from the allowlist identifier and the user's public key and verifies that the membership declaration is accumulated. In this case, the prover is computing an attestation from public information in such a way that the initial attestation can be reused in subsequent attestations.

If a vertex has n incoming edges, this indicates that each transaction input is a coin that was the output of a previous transfer transaction. In this case, the membership proof for this transaction will attest to the validity of previous membership proofs with respect to previous messages. However, a problem arises where the history accumulator digests of previous messages may be stale with respect to the current state of the history accumulator for the current message. We thus additionally need to prove that the prior history accumulator digests represent correct historical states with respect to the current history accumulator digest. Otherwise, there is no guarantee that the prior history accumulator digests correspond to valid prior contract states. The predicate will ensure consistency by verifying that the prior history accumulator digest of message z_i is a valid historical digest according to the current history accumulator digest of message z. The predicate will verify history proofs for three history accumulators: the membership declaration history accumulator, the deposit record history accumulator, and the transfer record history accumulator.

In both cases, the predicate computes nullifiers for the transaction inputs based on the input coin commitment openings. The predicate computes the output coin commitment from its opening. The predicate ensures the consistency of the output coin commitments of the previous messages with the input coin commitment openings of the current local data. Finally, the predicate computes the transfer record for each pair of input nullifier and output coin commitment and verifies that the transfer record is accumulated.

While these computations reference the (private) local data, the resulting proofs can be verified with respect to the corresponding (public) message. Each transaction output corresponds to an outgoing edge in the PCD graph, so each transaction output has a corresponding membership proof.

From the recipient's perspective, it is important to check the validity of the public information in the message z with respect to the privacy pool contract state, in addition to verifying the proof π with respect to the message z. Otherwise, it is not guaranteed that the membership proof is meaningful. The recipient should be able to perform this check at any time with access to the current state.

Our design offers flexibility in combining coins with membership proofs on distinct sets of allowlists that have a non-empty intersection. For instance, a coin with membership proofs on allowlists al_1 and al_2 may be combined with a coin with a membership proof on allowlist al_1 to produce transaction outputs with a membership proof on allowlist al_1 only.

- Message $z := (\mathbf{null}, \mathsf{cm}, \mathsf{rt}_{\mathsf{id}}, \mathbf{decl}, \mathsf{rt}_{\mathsf{dep}}, \mathbf{rec}_{\mathbf{dep}}, \mathbf{uid}, \mathsf{rt}_{\mathsf{tfr}}, \mathbf{rec}_{\mathbf{tfr}})$:
  - $\mathbf{null} := (\mathsf{null}_i)_{i \in [n]}$: List of nullifiers for the transaction inputs.
  - cm: Output coin commitment.
  - $\mathsf{rt}_{\mathsf{id}}$: Digest of membership declaration history accumulator.
  - $\mathbf{decl} := (\mathsf{decl}_i)_{i \in [n]}$: Membership declarations for the allowlist al_j.
  - $\mathsf{rt}_{\mathsf{dep}}$: Digest of deposit record history accumulator.
  - $\mathbf{rec}_{\mathbf{dep}} := (\mathsf{rec}_i^{\mathsf{dep}})_{i \in [n]}$: Deposit records derived from public information.
  - $\mathbf{uid} := (\mathsf{uid}_i)_{i \in [n]}$: Unique identifiers for the deposit transactions.
  - $\mathsf{rt}_{\mathsf{tfr}}$: Digest of transfer record history accumulator.
  - $\mathbf{rec}_{\mathbf{tfr}} := (\mathsf{rec}_i^{\mathsf{tfr}})_{i \in [n]}$: Transfer records derived from public information.

- Local data $z_{\mathsf{loc}} := (\mathbf{open_{in}}, \mathsf{open_{out}}, \mathbf{w_{id}}, \mathbf{c_{id}}, \mathbf{w_{dep}}, \mathbf{c_{dep}}, \mathbf{w_{tfr}}, \mathbf{c_{tfr}})$:
  - $\mathbf{open_{in}} := (\mathsf{amt}_i^{\mathsf{in}}, \mathsf{pk}_i^{\mathsf{in}}, r_i^{\mathsf{in}})_{i\in[n]}$: List of input coin commitment openings.
  - $\mathsf{open_{out}} := (\mathsf{amt_{out}}, \mathsf{pk_{out}}, r_{\mathsf{out}})$: Output coin commitment opening.
  - $\mathbf{w_{id}} := (\mathsf{w}_i^{\mathsf{id}})_{i\in[n]}$: Membership witnesses for membership declarations $\mathbf{decl}$ and digest $\mathsf{rt_{id}}$.
  - $\mathbf{c_{id}} := (\mathsf{c}_i^{\mathsf{id}})_{i\in[n]}$: History proofs for digest $\mathsf{rt_{id}}$ with respect to previous digests $\hat{\mathsf{rt}}_i^{\mathsf{id}}$.
  - $\mathbf{w_{dep}} := (\mathsf{w}_i^{\mathsf{dep}})_{i\in[n]}$: Membership witnesses for deposit records $\mathbf{rec_{dep}}$ and digest $\mathsf{rt_{dep}}$.
  - $\mathbf{c_{dep}} := (\mathsf{c}_i^{\mathsf{dep}})_{i\in[n]}$: History proofs for digest $\mathsf{rt_{dep}}$ with respect to previous digests $\hat{\mathsf{rt}}_i^{\mathsf{dep}}$.
  - $\mathbf{w_{tfr}} := (\mathsf{w}_i^{\mathsf{tfr}})_{i\in[n]}$: Membership witnesses for transfer records $\mathbf{rec_{tfr}}$ and digest $\mathsf{rt_{tfr}}$.
  - $\mathbf{c_{tfr}} := (\mathsf{c}_i^{\mathsf{tfr}})_{i\in[n]}$: History proofs for digest $\mathsf{rt_{tfr}}$ with respect to previous digests $\hat{\mathsf{rt}}_i^{\mathsf{tfr}}$.
- Previous messages $(z_i)_{i\in[n]}$:
  - $z_i := (\hat{\mathbf{null}}_i, \hat{\mathsf{cm}}_i, \hat{\mathsf{rt}}_i^{\mathsf{id}}, \hat{\mathbf{decl}}_i, \hat{\mathsf{rt}}_i^{\mathsf{dep}}, \hat{\mathbf{rec}}_i^{\mathbf{dep}}, \hat{\mathbf{uid}}_i, \hat{\mathsf{rt}}_i^{\mathsf{tfr}}, \hat{\mathbf{rec}}_{\mathbf{tfr}})$
  - z_i is a message for the i-th transaction input.
  - Each message z_i has the same format as z.
- Previous proofs $(\pi_i)_{i\in[n]}$:
  - π_i is a proof for the i-th transaction input.
  - Each proof π_i can be verified with respect to z_i.
- Compliance predicate $\varphi_j(z, z_{\mathsf{loc}}, z_1, \ldots, z_n)$ for allowlist al_j:
  - For $i \in [n]$:
    - ∗ For base case ($z_i = \bot$), check the following:
      - · $\mathsf{decl}_i = H_q(\mathsf{al}_j \| \mathsf{pk}_i^{\mathsf{in}})$
      - · $\mathsf{HA.VfyMem}(\mathsf{rt_{id}}, \mathsf{w}_i^{\mathsf{id}}, \mathsf{decl}_i)$
      - · $\mathsf{rec}_i^{\mathsf{dep}} = H_q(\mathsf{amt}_i^{\mathsf{in}} \| \mathsf{pk}_i^{\mathsf{in}} \| \mathsf{Com}(\mathsf{open}_i^{\mathsf{in}}) \| \mathsf{uid}_i)$
      - · $\mathsf{HA.VfyMem}(\mathsf{rt_{dep}}, \mathsf{w}_i^{\mathsf{dep}}, \mathsf{rec}_i^{\mathsf{dep}})$
    - ∗ Otherwise, check the consistency of messages:
      - · $\hat{\mathsf{cm}}_i^{\mathsf{out}} = \mathsf{Com}(\mathsf{amt}_i^{\mathsf{in}}, \mathsf{pk}_i^{\mathsf{in}}; r_i^{\mathsf{in}})$
      - · $\mathsf{HA.VfyHist}(\hat{\mathsf{rt}}_i^{\mathsf{id}}, \mathsf{rt_{id}}, \mathsf{c}_i^{\mathsf{id}})$
      - · $\mathsf{HA.VfyHist}(\hat{\mathsf{rt}}_i^{\mathsf{dep}}, \mathsf{rt_{dep}}, \mathsf{c}_i^{\mathsf{dep}})$
      - · $\mathsf{HA.VfyHist}(\hat{\mathsf{rt}}_i^{\mathsf{tfr}}, \mathsf{rt_{tfr}}, \mathsf{c}_i^{\mathsf{tfr}})$
  - For $i \in [n]$:
    - ∗ $\mathsf{null}_i = H_q(r_i^{\mathsf{in}})$
  - $\mathsf{cm} = \mathsf{Com}(\mathsf{amt_{out}}, \mathsf{pk_{out}}; r_{\mathsf{out}})$
  - For $i \in [n]$:
    - ∗ $\mathsf{rec}_i^{\mathsf{tfr}} = H_q(\mathsf{null}_i \| \mathsf{cm})$
    - ∗ $\mathsf{HA.VfyMem}(\mathsf{rt_{tfr}}, \mathsf{w}_i^{\mathsf{tfr}}, \mathsf{rec}_i^{\mathsf{tfr}})$

4.3 Transfer Proof

This is the zk-SNARK statement for the validity of an anonymous transfer in the privacy pool. The proof shows that the value amount is preserved in the transfer, the sender knows the secret keys for each of the input coin commitments, the input coin commitments are accumulated, the owner memo of each output is correctly encrypted, the nullifiers for the input coin commitments are correctly computed, and the output coin commitments are correctly computed.

Let n be the number of transaction inputs and m be the number of transaction outputs. We allow placeholder coins with a placeholder public key and a zero amount to support transaction padding.

- Statement:
  - For $i \in [n]$:
    - ∗ $\mathsf{pk}_i^{\mathsf{in}} = H_q(\mathsf{sk}_i^{\mathsf{in}} \| \mathsf{addr}_i^{\mathsf{in}})$
    - ∗ $\mathsf{null}_i = H_q(r_i^{\mathsf{in}})$
    - ∗ $\mathsf{Acc.VfyMem}(\mathsf{rt_c}, \mathsf{w}_i^{\mathsf{in}}, \mathsf{Com}(\mathsf{amt}_i^{\mathsf{in}}, \mathsf{pk}_i^{\mathsf{in}}; r_i^{\mathsf{in}}))$
  - $\sum_{i\in[n]} \mathsf{amt}_i^{\mathsf{in}} = \sum_{i\in[m]} \mathsf{amt}_i^{\mathsf{out}}$
  - For $i \in [m]$:
    - ∗ $\mathsf{cm}_i = \mathsf{Com}(\mathsf{amt}_i^{\mathsf{out}}, \mathsf{pk}_i^{\mathsf{out}}; r_i^{\mathsf{out}})$
    - ∗ $\mathsf{memo}_i = \mathsf{Enc}_{\mathsf{pk}_i'}((\mathsf{amt}_i^{\mathsf{out}}, \mathsf{pk}_i^{\mathsf{out}}, r_i^{\mathsf{out}}), \gamma_i)$
- Public inputs:

  - $(\mathsf{null}_i)_{i\in[n]}$: List of nullifiers for the transaction inputs.
  - $(\mathsf{cm}_i)_{i\in[m]}$: List of output coin commitments.
  - $(\mathsf{memo}_i)_{i\in[m]}$: List of output owner memos.
  - $\mathsf{rt_c}$: Digest of coin commitment accumulator.
- Private inputs:
  - $(\mathsf{amt}_i^{\mathsf{in}}, \mathsf{pk}_i^{\mathsf{in}}, r_i^{\mathsf{in}})_{i\in[n]}$: List of openings of input coin commitments.
  - $(\mathsf{addr}_i^{\mathsf{in}})_{i\in[n]}$: List of owner addresses for transaction inputs.
  - $(\mathsf{sk}_i^{\mathsf{in}})_{i\in[n]}$: List of user secret keys for transaction inputs.
  - $(\mathsf{w}_i^{\mathsf{in}})_{i\in[n]}$: List of membership witnesses for transaction inputs.
  - $(\mathsf{amt}_i^{\mathsf{out}}, \mathsf{pk}_i^{\mathsf{out}}, r_i^{\mathsf{out}})_{i\in[m]}$: List of openings of output coin commitments.
  - $(\mathsf{pk}'_i)_{i\in[m]}$: Public keys for encryption of transaction outputs.
  - $(\gamma_i)_{i\in[m]}$: Encryption randomness values for transaction outputs.

4.4 Withdrawal Proof

This is the zk-SNARK statement for the validity of a withdrawal from the privacy pool. The proof shows that the value amount is preserved in the withdrawal, the sender knows the secret keys for each of the input coin commitments, the input coin commitments are accumulated, and the nullifiers are correctly computed. Let n be the number of transaction inputs for the withdrawal. As in the case of the transfer statement, we allow placeholder coins to support transaction padding. We include the recipient address in the public inputs to ensure that a front-running adversary cannot change the address in the withdrawal transaction. This defense relies on the non-malleability of the proof, which is guaranteed by the simulation extractability property of the zk-SNARK scheme.

- Statement:
  - For $i \in [n]$:
    - $\mathsf{pk}_i^{\mathsf{in}} = H_q(\mathsf{sk}_i^{\mathsf{in}} \| \mathsf{addr}_i^{\mathsf{in}})$
    - $\mathsf{null}_i = H_q(r_i^{\mathsf{in}})$
    - $\mathsf{Acc.VfyMem}(\mathsf{rt_c}, \mathsf{w}_i^{\mathsf{in}}, \mathsf{Com}(\mathsf{amt}_i^{\mathsf{in}}, \mathsf{pk}_i^{\mathsf{in}}; r_i^{\mathsf{in}}))$
  - $\sum_{i\in[n]} \mathsf{amt}_i^{\mathsf{in}} = \mathsf{amt_{out}}$
- Public inputs:
  - $(\mathsf{null}_i)_{i\in[n]}$: List of nullifiers for the transaction inputs.
  - $\mathsf{addr_{out}}$: Output recipient address.
  - $\mathsf{amt_{out}}$: Output value amount.
  - $\mathsf{rt_c}$: Digest of coin commitment accumulator.
- Private inputs:
  - $(\mathsf{amt}_i^{\mathsf{in}}, \mathsf{pk}_i^{\mathsf{in}}, r_i^{\mathsf{in}})_{i\in[n]}$: List of openings of input coin commitments.
  - $(\mathsf{addr}_i^{\mathsf{in}})_{i\in[n]}$: List of owner addresses for transaction inputs.
  - $(\mathsf{sk}_i^{\mathsf{in}})_{i\in[n]}$: List of user secret keys for transaction inputs.
  - $(\mathsf{w}_i^{\mathsf{in}})_{i\in[n]}$: List of membership witnesses for transaction inputs.

4.5 System Algorithms

We present the client algorithms in Figure 1 and the privacy pool algorithms in Figure 2.

The client generates and registers a public-private key pair before transacting with the privacy pool. The client may create three types of financial transactions: deposit transactions, transfer transactions, and withdrawal transactions.

When a client transfers coins within the privacy pool, the client will separately create a membership proof list for each of the output coin commitments. As discussed above, we require special handling of the base case of the membership proof. When a client withdraws from the privacy pool, the client generates a final membership proof list. The privacy pool contract checks for transaction validity, but the contract does not have access to the proof-carrying disclosures. These proofs can be communicated through a direct channel to the recipient rather than being posted on a public bulletin board.

In an extension of the system, it would be possible to build upon the final membership proof lists for standard Ethereum transfers that occur after withdrawal from the privacy pool. Here we restrict our focus to transfers within the privacy pool.

```
$\textsc{CreateMembershipProof}_{n,m}(\ldots)$
Inputs: $\mathsf{pp}, \mathbf{sk_{in}}, \mathbf{open_{in}}, \mathbf{open_{out}}, \mathbf{decl}, \mathbf{w_{id}}, \mathbf{c_{id}}, \mathbf{rec_{dep}}, \mathbf{uid}, \mathbf{w_{dep}}, \mathbf{c_{dep}}, \mathbf{rec_{tfr}}, \mathbf{w_{tfr}}, \mathbf{c_{tfr}}, \boldsymbol{z}_{\mathbf{in}}, \boldsymbol{\pi}_{\mathbf{in}}$
  // Compute null and cm as specified in the deposit and withdrawal operations
Set $\boldsymbol{z}_{\mathbf{out}} = []$ and $\boldsymbol{\pi}_{\mathbf{out}} = []$
for $j \in [l]$ do
  for $i \in [m]$ do
    if $z^{\mathsf{in}}_{ij} = \perp$ (base case) then
      Set $z = (\mathbf{null}, \mathbf{cm}[i], \mathsf{rt_{id}}, \mathbf{decl}[\ldots][j], \mathsf{rt_{dep}}, \mathbf{rec_{dep}}, \mathbf{uid}, \mathsf{rt_{tfr}}, \mathbf{rec_{tfr}})$
      Set $z_{\mathsf{loc}} = (\mathbf{open_{in}}, \mathbf{open_{out}}[i], \mathbf{w_{id}}, \perp, \mathbf{w_{dep}}, \perp, \mathbf{w_{tfr}}, \perp)$
    else
      Set $z = (\mathbf{null}, \mathbf{cm}[i], \mathsf{rt_{id}}, \perp, \mathsf{rt_{dep}}, \perp, \perp, \mathsf{rt_{tfr}}, \mathbf{rec_{tfr}})$
      Set $z_{\mathsf{loc}} = (\mathbf{open_{in}}, \mathbf{open_{out}}[i], \perp, \mathbf{c_{id}}, \perp, \mathbf{c_{dep}}, \mathbf{w_{tfr}}, \mathbf{c_{tfr}})$
    endif
    Generate proof $\pi^{\mathsf{out}}_{ij} = \mathsf{PCD}.\mathbb{P}(\mathsf{ipk}_j, z, z_{\mathsf{loc}}, [z^{\mathsf{in}}_{ij}, \pi^{\mathsf{in}}_{ij}]^n_{i=1})$
    Set $\boldsymbol{z}_{\mathbf{out}}[i][j] = z$ and $\boldsymbol{\pi}_{\mathbf{out}}[i][j] = \pi^{\mathsf{out}}_{ij}$
  endfor
endfor
return $(\boldsymbol{z}_{\mathbf{out}}, \boldsymbol{\pi}_{\mathbf{out}})$
```

```
$\textsc{GenerateKeyPair}(\mathsf{pp}, \mathsf{addr})$
Generate encryption keys $(\mathsf{pk}', \mathsf{sk}') \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$
Generate secret key $\mathsf{sk} \leftarrow\!\$\ \{0,1\}^\lambda$
Generate public key $\mathsf{pk} = H_q(\mathsf{sk}\|\mathsf{addr})$
return $(\mathsf{sk}, \mathsf{pk}, \mathsf{sk}', \mathsf{pk}')$
```

```
$\textsc{CreateDepositTx}(\mathsf{pp}, \mathsf{amt}, \mathsf{pk})$
Sample $r \leftarrow\!\$\ \{0,1\}^\lambda$
Compute coin commitment $\mathsf{cm} = H_q(\mathsf{amt}\|\mathsf{pk}\|r)$
Set $\mathsf{open} = (\mathsf{amt}, \mathsf{pk}, r)$
Set $\mathsf{tx_{dep}} = (\mathsf{cm}, \mathsf{amt}, \mathsf{pk})$
return $\mathsf{tx_{dep}}$
```

```
$\textsc{CreateTransferTx}_{n,m}(\ldots)$
Inputs: $\mathsf{pp}, \mathbf{sk_{in}}, \mathbf{open_{in}}, \mathbf{addr_{in}}, \mathbf{open_{out}}, \mathbf{pk'}$
Set $\mathbf{null} = []$ and $\mathbf{w_{in}} = []$
for $i \in [n]$ do
  Set $\mathsf{sk} = \mathbf{sk_{in}}[i]$
  Parse $(\mathsf{amt}, \mathsf{pk}, r) = \mathbf{open_{in}}[i]$
  Compute coin commitment $\mathsf{cm} = H_q(\mathsf{amt}\|\mathsf{pk}\|r)$
  Compute nullifier $\mathsf{null} = H_q(r)$
  Compute $\mathsf{w} = \mathsf{Acc.PrvMem}(\sigma_{\mathsf{c}}, \mathsf{cm})$
  Add $\mathsf{null}$ to $\mathbf{null}$, $\mathsf{w}$ to $\mathbf{w_{in}}$
endfor
Set $\mathbf{cm} = [], \mathbf{memo} = []$, and $\boldsymbol{\gamma} = []$
for $i \in [m]$ do
  Parse $(\mathsf{amt}, \mathsf{pk}, r) = \mathbf{open_{out}}[i]$
  Compute coin commitment $\mathsf{cm} = H_q(\mathsf{amt}\|\mathsf{pk}\|r)$
  Sample $\gamma \leftarrow\!\$\ \{0,1\}^\lambda$
  Set $\mathsf{pk}' = \mathbf{pk'}[i]$
  Compute owner memo $\mathsf{memo} = \mathsf{Enc}_{\mathsf{pk}'}(\mathsf{amt}\|\mathsf{pk}\|r; \gamma)$
  Add $\mathsf{cm}$ to $\mathbf{cm}$, $\mathsf{memo}$ to $\mathbf{memo}$, $\gamma$ to $\boldsymbol{\gamma}$
endfor
Set instance $x = (\mathbf{null}, \mathbf{cm}, \mathbf{memo}, \mathsf{rt_c})$
Set witness $w = (\mathbf{open_{in}}, \mathbf{addr_{in}}, \mathbf{sk_{in}}, \mathbf{w_{in}}, \mathbf{open_{out}}, \mathbf{pk'}, \boldsymbol{\gamma})$
Generate transfer proof $\pi_t = \mathsf{ARG}.\mathcal{P}(\mathsf{ek}_t, x, w)$
Set $\mathsf{tx_{tfr}} = (\mathbf{null_{in}}, \mathbf{cm_{out}}, \mathbf{memo_{out}}, \mathsf{rt_c}, \pi_t)$
return $\mathsf{tx_{tfr}}$
```

```
$\textsc{CreateWithdrawalTx}_{n}(\ldots)$
Inputs: $\mathsf{pp}, \mathbf{sk_{in}}, \mathbf{open_{in}}, \mathbf{addr_{in}}, \mathsf{open_{out}}, \mathsf{addr_{out}}$
Set $\mathbf{null} = []$ and $\mathbf{w_{in}} = []$
for $i \in [n]$ do
  Set $\mathsf{sk} = \mathbf{sk_{in}}[i]$
  Parse $(\mathsf{amt}, \mathsf{pk}, r) = \mathbf{open_{in}}[i]$
  Compute coin commitment $\mathsf{cm} = H_q(\mathsf{amt}\|\mathsf{pk}\|r)$
  Compute nullifier $\mathsf{null} = H_q(r)$
  Compute $\mathsf{w} = \mathsf{Acc.PrvMem}(\sigma_{\mathsf{c}}, \mathsf{cm})$
  Add $\mathsf{null}$ to $\mathbf{null}$, $\mathsf{w}$ to $\mathbf{w_{in}}$
endfor
Set $\mathbf{cm} = []$
Parse $(\mathsf{amt}, \mathsf{pk}, r) = \mathsf{open_{out}}$
Compute coin commitment $\mathsf{cm} = H_q(\mathsf{amt}\|\mathsf{pk}\|r)$
Add $\mathsf{cm}$ to $\mathbf{cm}$
Set instance $x = (\mathbf{null}, \mathsf{amt}, \mathsf{addr_{out}}, \mathsf{rt_c})$
Set witness $w = (\mathbf{open_{in}}, \mathbf{addr_{in}}, \mathbf{sk_{in}}, \mathbf{w_{in}})$
Generate withdrawal proof $\pi_w = \mathsf{ARG}.\mathcal{P}(\mathsf{ek}_w, x, w)$
Set $\mathsf{tx_{wdr}} = (\mathsf{amt}, \mathsf{addr_{out}}, \mathbf{null}, \mathsf{cm}, \mathsf{rt_c}, \pi_w)$
return $\mathsf{tx_{wdr}}$
```

Fig. 1: Client Algorithms.

```
$\textsc{PrivacyPoolSetup}(1^\lambda)$
* Sample hash function $H_q : \{0,1\}^* \to \mathbb{F}_q$
* Specify the maximum deposit amount $\mathsf{amt}_{\mathsf{max}} \in \mathbb{N}$ and the Merkle Tree depth $d \in \mathbb{N}$
* Create $(\mathsf{rt}_{\mathsf{dep}}, \sigma_{\mathsf{dep}}) = \mathsf{HA.Init}(1^\lambda)$, $(\mathsf{rt}_{\mathsf{tfr}}, \sigma_{\mathsf{tfr}}) = \mathsf{HA.Init}(1^\lambda)$, and $(\mathsf{rt}_{\mathsf{id}}, \sigma_{\mathsf{id}}) = \mathsf{HA.Init}(1^\lambda)$
* Run universal setup for zk-SNARK: $\mathsf{srs} = \mathsf{ARG}.\mathcal{G}(1^\lambda)$
* Construct circuit descriptions $t$ for transfer proof and $w$ for withdrawal proof
* Generate keys for circuit: $\mathsf{ek_t}, \mathsf{vk_t} \leftarrow \mathsf{ARG}.\mathcal{I}_{\mathsf{srs}}(t)$ and $\mathsf{ek_w}, \mathsf{vk_w} \leftarrow \mathsf{ARG}.\mathcal{I}_{\mathsf{srs}}(w)$
* Construct compliance predicates $(\varphi_j)_{j\in[l]}$ for membership proofs
* Run setup for PCD: $\mathsf{pp_{pcd}} = \mathsf{PCD}.\mathbb{G}(1^\lambda)$
for $j \in [l]$ do
  * Generate keys for PCD: $\mathsf{ipk}_j, \mathsf{vpk}_j \leftarrow \mathsf{PCD}.\mathbb{I}(\mathsf{pp_{pcd}}, \varphi_j)$
endfor
Create $(\mathsf{rt_c}, \sigma_{\mathsf{c}}) = \mathsf{Acc.Init}(1^\lambda)$
Set $\mathsf{NullifierList} = \emptyset$ and $\mathsf{DigestList} = \emptyset$
Set $\mathsf{AccountList} = \{\}$ and $\mathsf{AuthAccountList} = \{\}$
Populate $\mathsf{AuthAccountList}$ for each allowlist $\mathsf{al}_j$
Set $\mathsf{pp} = \{H_q, \mathsf{amt_{max}}, d, \mathsf{rt_c}, \sigma_{\mathsf{c}}, \mathsf{rt_{id}}, \sigma_{\mathsf{id}}, \mathsf{rt_{dep}}, \sigma_{\mathsf{dep}}, \mathsf{rt_{tfr}}, \sigma_{\mathsf{tfr}}, \mathsf{ek_t}, \mathsf{vk_t}, \mathsf{ek_w}, \mathsf{vk_w}, (\mathsf{ipk}_j, \mathsf{vpk}_j)_{j\in[l]}$
      $\mathsf{NullifierList}, \mathsf{DigestList}, \mathsf{AccountList}, \mathsf{AuthAccountList}\}$
return $\mathsf{pp}$
```

```
$\textsc{ProcessRegistrationTx}(\mathsf{pp}, \mathsf{tx_{reg}})$
Parse $(\mathsf{pk}, \mathbf{al}) = \mathsf{tx_{reg}}$
Set $\mathsf{AccountList}[\mathsf{tx_{reg}.sender}] = \mathsf{pk}$
Set $\mathbf{decl} = []$
for $\mathsf{al} \in \mathbf{al}$ do
  Require $\mathsf{tx_{reg}.sender}$ is in $\mathsf{AuthAddressList}[\mathsf{al}]$
  Compute $\mathsf{decl} = H_q(\mathsf{pk}\|\mathsf{al})$
  Add $\mathsf{decl}$ to $\mathbf{decl}$
  * Update $\mathsf{rt_{id}} = \mathsf{HA.Update}(\mathsf{rt_{id}}, \sigma_{\mathsf{id}}, \mathsf{decl})$
endfor
return $(1, \mathbf{decl})$
```

```
$\textsc{ProcessDepositTx}(\mathsf{pp}, \mathsf{tx_{dep}})$
Parse $(\mathsf{cm}, \mathsf{amt}, \mathsf{pk}) = \mathsf{tx_{dep}}$
Check $\mathsf{amt}$ is less than or equal to $\mathsf{amt_{max}}$
Update $\mathsf{rt_c} = \mathsf{Acc.Update}(\mathsf{rt_c}, \sigma_{\mathsf{c}}, \mathsf{cm})$
Add $\mathsf{rt_c}$ to $\mathsf{DigestList}$
Transfer $\mathsf{amt}$ units from $\mathsf{tx_{dep}.sender}$ to contract address
* Compute unique identifier $\mathsf{uid}$ for deposit
* Create deposit record $\mathsf{rec_{dep}} = H_q(\mathsf{amt}\|\mathsf{pk}\|\mathsf{cm}\|\mathsf{uid})$
* Update $\mathsf{rt_{dep}} = \mathsf{HA.Update}(\mathsf{rt_{dep}}, \sigma_{\mathsf{dep}}, \mathsf{rec_{dep}})$
return $(1, \mathsf{rec_{dep}}, \mathsf{uid})$
```

```
$\textsc{ProcessTransferTx}_{n,m}(\mathsf{pp}, \mathsf{tx_{tfr}})$
Parse $(\mathbf{null}, \mathbf{cm}, \mathbf{memo}, \mathsf{rt}'_{\mathsf{c}}, \pi_t) = \mathsf{tx_{tfr}}$
Require $\mathsf{rt}'_{\mathsf{c}} \in \mathsf{DigestList}$
for $i \in [n]$ do
  Set $\mathsf{null} = \mathbf{null}[i]$
  Require $\mathsf{null} \notin \mathsf{NullifierList}$
  Add $\mathsf{null}$ to $\mathsf{NullifierList}$
endfor
for $i \in [m]$ do
  Set $\mathsf{cm} = \mathbf{cm}[i]$
  Update $\mathsf{rt_c} = \mathsf{Acc.Update}(\mathsf{rt_c}, \sigma_{\mathsf{c}}, \mathsf{cm})$
  for $j \in [n]$ do
    Set $\mathsf{null} = \mathbf{null}[j]$
    * Create transfer record $\mathsf{rec_{tfr}} = H_q(\mathsf{null}\|\mathsf{cm})$
    * Update $\mathsf{rt_{tfr}} = \mathsf{HA.Update}(\mathsf{rt_{tfr}}, \sigma_{\mathsf{tfr}}, \mathsf{rec_{tfr}})$
  endfor
endfor
Add $\mathsf{rt_c}$ to $\mathsf{DigestList}$
Require $\mathsf{ARG}.\mathcal{V}(\mathsf{vk}_t, [\mathbf{null}, \mathbf{cm}, \mathbf{memo}, \mathsf{rt}'_{\mathsf{c}}], \pi_t)$
return $(1, \mathsf{rec_{tfr}})$
```

```
$\textsc{ProcessWithdrawalTx}_n(\mathsf{pp}, \mathsf{tx_{wdr}})$
Parse $(\mathsf{amt}, \mathsf{addr}, \mathbf{null}, \mathsf{cm}, \mathsf{rt}'_{\mathsf{c}}, \pi_w) = \mathsf{tx_{wdr}}$
Require $\mathsf{rt}'_{\mathsf{c}} \in \mathsf{DigestList}$
for $i \in [n]$ do
  Set $\mathsf{null} = \mathbf{null}[i]$
  Require $\mathsf{null} \notin \mathsf{NullifierList}$
  Add $\mathsf{null}$ to $\mathsf{NullifierList}$
  * Create transfer record $\mathsf{rec_{tfr}} = H_q(\mathsf{null}\|\mathsf{cm})$
  * Update $\mathsf{rt_{tfr}} = \mathsf{HA.Update}(\mathsf{rt_{tfr}}, \sigma_{\mathsf{tfr}}, \mathsf{rec_{tfr}})$
endfor
Require $\mathsf{ARG}.\mathcal{V}(\mathsf{vk}_w, [\mathbf{null}, \mathsf{amt}, \mathsf{addr}, \mathsf{rt}'_{\mathsf{c}}], \pi_w)$
Transfer $\mathsf{amt}$ units from contract address to recipient $\mathsf{addr}$
return $(1, \mathsf{rec_{tfr}})$
```

Fig. 2: Privacy Pool Algorithms. Operations prefixed by * occur off-chain.

Component	Sub-component	Constraints
Membership	Value Computation	1,258
	Membership Proof	6,161
	History Proof	6,161
Deposit Record	Value Computation	6,420
	Membership Proof	6,161
	History Proof	6,161
Transfer Record	Value Computation	1,990
	Membership Proof	6,161
	History Proof	6,161
Transaction	Value Computation	3,705
	Value Consistency	1
Total	–	**50,340**

Fig. 3: Component-wise breakdown of R1CS constraints for the compliance predicate in the recursive membership proof.

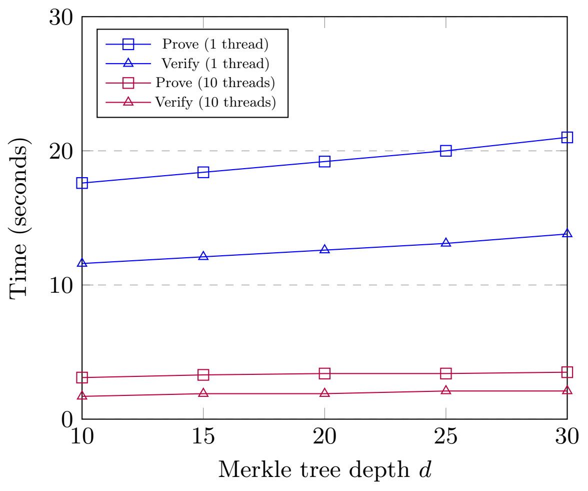

Fig. 4: Recursive membership proof generation time and verification time for a range of system parameters and parallel processing configurations.

5 Evaluation

We implement our construction of proof-carrying disclosures using Rust and the Arkworks ecosystem [ac22]. Our implementation consists of ≈ 5500 lines of code and is available open source.[2]

We additionally evaluated a prototype of privacy pool based on Ethereum to measure the costs of registration relative to the standard operations of the privacy pool. This prototype is derived from an open-source implementation that replicates the functionality of Tornado Cash and includes optimizations suggested in the Tornado Cash audit report, such as replacing the MiMC hash function with the Poseidon hash function for efficiency.

The constraints and proof-carrying data primitives are implemented within the Arkworks ecosystem. We instantiate the PCD scheme of [BCL+21] with the Pasta cycle of elliptic curves [Hop20]. This PCD scheme uses a transparent setup.

The compliance predicate circuit consists of 50,340 constraints. A component-wise breakdown of the constraints is provided in Figure 3. The verification of the membership proofs and history proofs accounts for the majority of the constraints in the compliance predicate. The constraint count has been optimized through usage of the Poseidon [GKR+21] hash function.

We evaluated the performance on a laptop with an Apple M1 Max processor. For a tree depth of 20 and single-threaded execution, the setup time was 5.4 seconds, the proving time was 19.2 seconds, and the verification time was 12.6 seconds. With multi-threaded execution, the setup time was 2.0 seconds, the proving time was 3.4 seconds, and the verification time was 1.9 seconds. Figure 4 contains proving/verification times for a range of Merkle tree depth values. The proof size was 6.3MB for a tree depth of 20. Other PCD schemes such as [BCTV14] could be used, resulting in different tradeoffs in the setup type, proving/verification time, and proof size.

Our design does not introduce additional gas costs in the core functionality of the privacy pool contract. The history accumulators can be maintained offline based on the public blockchain state to support generation of membership proofs. The allowlist functionality incurs minimal costs to the pool operator for contract deployment (559k gas) and minimal costs to the user for the one-time registration operation (49k gas for an allowlist of size 2^{20}). Furthermore, the cost of the registration operation increases slowly as it is based on Merkle path verification (4.7k gas for a 2x increase in allowlist size). For comparison, the deployment of the Tornado Cash contract incurs costs of 5.38m gas, whereas deposit transactions and withdrawal transactions incur costs of 857k gas and 331k gas, respectively. As a result, users do not incur significant additional gas costs in this design. These results should extend to any smart contract platform with cost-efficient hash function operations.

[2] `https://github.com/joshbeal/derecho`

6 Future Work

Derecho supports attestations of membership on allowlists. Allowlists can also be used to implement blocklists by simply including every *unblocked* address on the allowlist. The challenge with supporting blocklists more directly is that a user can easily transfer funds from a blocklisted address pk_s to a fresh Ethereum address pk'_s before depositing into the privacy pool. The funds might then be transfered several hops within the pool before the blocklist could be updated to include pk'_s. At this stage, the output records of these hops would include valid proofs of non-membership. It would be infeasible to require these proofs to be updated relative to the newer states of the blocklists because the holders of those records do not have knowledge of the records' origin, only the non-membership attestations that were valid against the previous blocklist states. In the case of allowlists, users can be required to register new addresses on the allowlists so that freshly created addresses are not automatically included, preventing users from undermining the disclosure proof system by creating a fresh Ethereum address before depositing into the pool. On the other hand, a similar challenge would arise if addresses are removed from allowlists, but we do not anticipate this would happen as frequently. Given that using allowlists to implement blocklists results in an enormously large list, it would be interesting to develop alternative constructions that more directly support blocklist non-membership proofs, while preventing attacks of the nature described above. One solution is to blocklist all fresh addresses, although this may result in a similar issue of blocklist bloat. We leave this question for future work.

While it may be desirable to extend Derecho to support *revocation* of allowlist membership, this functionality could result in a direct conflict with the privacy goals of the system. Suppose there exists a construction where a single address addr may be removed from an allowlist al to create a new allowlist al' such that old membership proofs for al can be updated to support verification against al'. Then a membership proof that was previously valid against the old allowlist al but not the new allowlist al' can be used to break the privacy guarantee of unlinkability. More precisely, this implies that the party who is able to compute al' and the party who is able to update a membership proof π for funds stored at a given record within the privacy pool would be able to collude at any time to discover all addresses on al from which the funds originated. We leave exploration of relaxed privacy models that might be compatible with revocation for future work.

7 Acknowledgments

This work was supported by the Algorand Centres of Excellence programme managed by Algorand Foundation. 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 Algorand Foundation.

References

AAB$^+$20. A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Trans. Symm. Cryptol.*, 2020(3):1–45, 2020.

ac22. arkworks contributors. `arkworks` zksnark ecosystem, 2022. `https://github.com/arkworks-rs/`.

AKR$^+$13. E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in bitcoin. In *International conference on financial cryptography and data security*, pages 34–51. Springer, 2013.

BBC$^+$22. C. Bouvier, P. Briaud, P. Chaidos, L. Perrin, and V. Velichkov. Anemoi: Exploiting the link between arithmetization-orientation and CCZ-equivalence. Cryptology ePrint Archive, Report 2022/840, 2022. `https://eprint.iacr.org/2022/840`.

BCCT13. N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *45th ACM STOC*, pages 111–120. ACM Press, June 2013.

BCG$^+$14. E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

BCG$^+$20. S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.

BCL$^+$21. B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. Proof-carrying data without succinct arguments. LNCS, pages 681–710. Springer, Heidelberg, 2021.

BCMS20. B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. Proof-carrying data from accumulation schemes. In *Theory of Cryptography*. Springer, 2020.

BCTV14. E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO 2014, Part II*, *LNCS* 8617, pages 276–294. Springer, Heidelberg, August 2014.

BDFG21. D. Boneh, J. Drake, B. Fisch, and A. Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In *Annual International Cryptology Conference*, pages 649–680. Springer, 2021.

BG13. S. Bayer and J. Groth. Zero-knowledge argument for polynomial evaluation with application to blacklists. In *EUROCRYPT 2013*, *LNCS* 7881, pages 646–663. Springer, Heidelberg, May 2013.

BGH19. S. Bowe, J. Grigg, and D. Hopwood. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive*, 2019. `https://eprint.iacr.org/2019/1021`.

BKB22. J. Burleson, M. Korver, and D. Boneh. Privacy-protecting regulatory solutions using zero-knowledge proofs. 2022. `https://a16zcrypto.com/wp-content/uploads/2022/11/ZKPs-and-Regulatory-Compliant-Privacy.pdf`.

BKLZ20. B. Bünz, L. Kiffer, L. Luu, and M. Zamani. FlyClient: Super-light clients for cryptocurrencies. In *2020 IEEE Symposium on Security and Privacy*, pages 928–946. IEEE Computer Society Press, May 2020.

BMRS20. J. Bonneau, I. Meckler, V. Rao, and E. Shapiro. Coda: Decentralized cryptocurrency at scale. 2020. `https://eprint.iacr.org/2020/352`.

CBC21. P. Chatzigiannis, F. Baldimtsi, and K. Chalkias. Sok: auditability and accountability in distributed payment systems. In *International Conference on Applied Cryptography and Network Security*, pages 311–337. Springer, 2021.

CCDW20. W. Chen, A. Chiesa, E. Dauterman, and N. P. Ward. Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Report 2020/1522, 2020. `https://eprint.iacr.org/2020/1522`.

CHL06. J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Balancing accountability and privacy using e-cash. In *International conference on security and cryptography for networks*, pages 141–155. Springer, 2006.

Coi22. Coin Center. Tornado cash complaint, 2022. `https://www.coincenter.org/app/uploads/2022/10/1-Complaint-Coin-Center-10-12-22.pdf`.

COS20. A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT 2020, Part I*, *LNCS* 12105, pages 769–793. Springer, Heidelberg, May 2020.

Cro10. S. A. Crosby. *Efficient tamper-evident data structures for untrusted servers.* Rice University, 2010.

CT10. A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS 2010*, pages 310–331. Tsinghua University Press, January 2010.

CW09. S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security 2009*, pages 317–334. USENIX Association, August 2009.

Esp22. Espresso Systems. Configurable asset privacy. 2022. `https://github.com/EspressoSystems/cap`.

Fis22. B. Fisch. Privacy-protecting regulatory solutions using zero-knowledge proofs. 2022. `https://www.espressosys.com/blog/configurable-privacy-case-study-partitioned-privacy-pools`.

GGM16. C. Garman, M. Green, and I. Miers. Accountable privacy for decentralized anonymous payments. In *FC 2016*, *LNCS* 9603, pages 81–98. Springer, Heidelberg, February 2016.

GHR$^+$22. L. Grassi, Y. Hao, C. Rechberger, M. Schofnegger, R. Walch, and Q. Wang. Horst meets fluid-spn: Griffin for zero-knowledge applications. 2022. `https://eprint.iacr.org/2022/403`.

GKK$^+$22. C. Ganesh, H. Khoshakhlagh, M. Kohlweiss, A. Nitulescu, and M. Zając. What makes fiat–shamir zksnarks (updatable srs) simulation extractable? In *International Conference on Security and Cryptography for Networks*, pages 735–760. Springer, 2022.

GKL$^+$22. L. Grassi, D. Khovratovich, R. Lüftenegger, C. Rechberger, M. Schofnegger, and R. Walch. Reinforced concrete: A fast hash function for verifiable computation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1323–1335, 2022.

GKR$^+$21. L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. pages 519–535. USENIX Association, 2021.

HBHW22. D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. 2022. `https://zips.z.cash/protocol/protocol.pdf`.

Hop20. D. Hopwood. The pasta curves for halo 2 and beyond, 2020. `https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/`.

KKS22. A. Kiayias, M. Kohlweiss, and A. Sarencheh. Peredi: Privacy-enhanced, regulated and distributed central bank digital currencies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1739–1752, 2022.

KST22. A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.

LLK13. B. Laurie, A. Langley, and E. Kasper. Rfc 6962: Certificate transparency, 2013. `https://www.rfc-editor.org/rfc/rfc6962`.

MKL$^+$20. S. Meiklejohn, P. Kalinnikov, C. S. Lin, M. Hutchinson, G. Belvin, M. Raykova, and A. Cutter. Think global, act local: Gossip and client audits in verifiable data structures. *arXiv preprint arXiv:2011.04551*, 2020.

MPJ$^+$13. S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140, 2013.

MSH$^+$18. M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan, et al. An empirical analysis of traceability in the monero blockchain. *Proceedings on Privacy Enhancing Technologies*, 3:143–163, 2018.

NT16. A. Naveh and E. Tromer. PhotoProof: Cryptographic image authentication for any set of permissible transformations. In *2016 IEEE Symposium on Security and Privacy*, pages 255–271. IEEE Computer Society Press, May 2016.

RPX$^+$22. D. Rathee, G. V. Policharla, T. Xie, R. Cottone, and D. Song. Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi. 2022. `https://eprint.iacr.org/2022/1286`.

Sol22. A. Soleimani. Permissioned privacy pools. 2022. `https://ethresear.ch/t/permissioned-privacy-pools/13572`.

STS99. T. Sander and A. Ta-Shma. Flow control: a new approach for anonymity control in electronic cash systems. In *International conference on financial cryptography*, pages 46–61. Springer, 1999.

TBA$^+$22. A. Tomescu, A. Bhat, B. Applebaum, I. Abraham, G. Gueta, B. Pinkas, and A. Yanai. UTT: Decentralized ecash with accountable privacy. Cryptology ePrint Archive, Report 2022/452, 2022. `https://eprint.iacr.org/2022/452`.

TFZ$^+$22. N. Tyagi, B. Fisch, A. Zitek, J. Bonneau, and S. Tessaro. Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2793–2807, 2022.

TKCS09. P. P. Tsang, A. Kapadia, C. Cornelius, and S. W. Smith. Nymble: Blocking misbehaving users in anonymizing networks. *IEEE Transactions on Dependable and Secure Computing*, 8(2):256–269, 2009.

Uni22. United States Department of the Treasury. U.s. treasury sanctions notorious virtual currency mixer tornado cash. 2022. `https://home.treasury.gov/news/press-releases/jy0916`.

Val08. P. Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC 2008*, *LNCS* 4948, pages 1–18. Springer, Heidelberg, March 2008.

WKDC22. K. Wüst, K. Kostiainen, N. Delius, and S. Capkun. Platypus: A central bank digital currency with unlinkable transactions and privacy-preserving regulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2947–2960, 2022.

WMW$^+$22. M. Wu, W. McTighe, K. Wang, I. A. Seres, N. Bax, M. Puebla, M. Mendez, F. Carrone, T. De Mattey, H. O. Demaestri, et al. Tutela: An open-source tool for assessing user-privacy on ethereum and tornado cash. *arXiv preprint arXiv:2201.06811*, 2022. `https://arxiv.org/abs/2201.06811`.