Haze: A Compliant Privacy Mixer

Maya Dotan[1,2][0000−0001−6576−9724] Ayelet Lotem[2][0000−0003−4037−1028] and Margarita Vald[2][0000−0003−1149−7182]

[1] The Hebrew University of Jerusalem
{mayadotan, ayelet.lotem}@mail.huji.ac.il
[2] Intuit Israel Inc. margarita.vald@cs.tau.ac.il

Abstract. Blockchains enable mutually distrustful parties to perform financial operations in a trustless, decentralized, publicly-verifiable environment. Blockchains typically offer little privacy, and thus motivated the construction of *privacy mixers*, a solution to make funds untraceable. Privacy mixers concern regulators due to their increasing use by bad actors to illegally conceal the origin of funds. Consequently, Tornado Cash, the largest privacy mixer to date is sanctioned by large portions of the Ethereum network.

In this work, we present Haze, a *compliant* privacy mixer. Haze guarantees users' privacy together with *compliance*, i.e., funds can be withdrawn as long as they were deposited from a non-banned address, without revealing any information on the matching deposit. We empirically evaluate our solution in a proof-of-concept system, demonstrating gas consumption for each deposit and withdrawal that is comparable to Tornado Cash for compliant users, and there is an optional feature for non-compliant funds to be released from the mixer to some predetermined entity. To the best of our knowledge, our solution is the first to guarantee compliance and privacy on the blockchain (on-chain) that is implemented via a smart contract. Finally, we introduce an alternative compliant privacy mixer protocol that supports de-anonymization of non-compliant users, at the cost of increased trust in the banned-addresses maintainer, which is realized in the two-server model.

1 Introduction

Blockchains and privacy. Blockchains are decentralized, publicly verifiable, and distributed append-only immutable ledgers that allow mutually distrustful parties to maintain a common state. Bitcoin [27] is the first blockchain system to go live, enabling parties to engage in money transfers using the native currency of the blockchain. Ethereum [40] is a blockchain platform that enables users, in addition to simple money transfers, to perform more complex operations in the form of a smart contract. A smart contract can be any program implemented on the blockchain. The state of the smart contract is maintained as part of the state of the blockchain. While Bitcoin and Ethereum offer users pseudonymity, in both blockchains funds are traceable. Over the years, there have been several attempts at adding various flavors of privacy to blockchains [5, 18, 31, 38]. One such flavor is untraceability of funds. A popular way to make funds untraceable in blockchains is through the use of privacy mixers [4, 17, 37]. A widely used privacy mixer in practice is Tornado Cash [28], which is decentralized and implemented via a smart contract on the blockchain. The untraceability property provided by privacy mixers aided a growing phenomenon of money being laundered via such systems. For instance, the Ethereum address $0x \ldots 383E2f96$ which belongs to the hacker group Lazarus of North Korea [34] used Tornado Cash to launder millions of dollars in stolen funds.

The U.S. Department of Treasury publishes the "Specially Designated Nationals And Blocked Persons List (SDN)" [35] that contains addresses of persons that the U.S. prohibits dealing with, as part as the OFAC list (Office of Foreign Assets Control). This list contains, amongst other things, blockchain addresses suspected to be involved in various types of illegal activity. In August 2022, following the Lazarus incident, the list was updated to include Tornado Cash [36]. This act changed the patterns of block-inclusion for Tornado Cash transactions by miners/validators. Today about a third of validators in the Ethereum network censor Tornado Cash transactions [23]. Currently, such a list is maintained on the Ethereum blockchain by Chainalysis [7].

The extensive usage of privacy mixers to move illicit funds and the addition of Tornado Cash to OFAC's list emphasizes the need for solutions that provide privacy only to "good" users, but do not allow access to the system to entities that do not comply with the policy. In this paper we refer to the problem of preventing addresses from OFAC's list from transferring funds through a privacy mixer as the "compliance" problem.

A *compliant privacy mixer* is therefore a mixer that preserves privacy in the sense of fund untraceability for honest users, ones that are not on the banned-addresses list, and does not enable the release of funds deposited from banned addresses on the list, even if the address only becomes banned after successfully depositing funds to the mixer. To construct such mixers one needs to take into account the dynamic nature of the banned-addresses list that is constantly updated to include new addresses. For this reason, a compliant privacy mixer must verify that, at

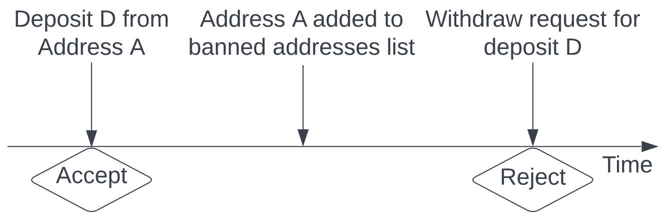

Deposit D from Address A
Address A added to banned addresses list
Withdraw request for deposit D
Accept
Reject
Time

Fig. 1: Deposit becoming non-compliant after entering the mixer.

the time of withdrawal, the funds being withdrawn did not originate from a banned address, see Fig. 1. However, this requirement that is essential for achieving compliance together with privacy induces a non-trivial combination, as at the time of withdrawal the mixer must be oblivious to the origin of the funds. Burleson et al. [6] were the first to introduce the question of compliant privacy mixers, and discuss at a high level the desired features of such a solution. However formal security definitions and implemented solutions are still missing. This raises the following question:

How can we construct a practical privacy mixer with compliance?

1.1 Our contribution

In this work we construct the first compliant privacy mixer, Haze, that guarantees the following security properties: (1) correctness - compliant users can always withdraw their funds, (2) soundness - funds cannot be double spent, (3) privacy in the form of deposit-withdrawal unlinkability, and (4) compliance - users on the banned-addresses list cannot withdraw funds from the mixer. Moreover, we formalize these properties and cast them into general security definitions. As mentioned before, we consider compliance with respect to the banned-addresses list, but nonetheless, our construction can be coupled with any general compliance policy that can be checked against a deposit transaction. We implement Haze and evaluate its performance together with a detailed comparison to the most prominent privacy mixer, Tornado Cash. We introduce an alternative compliant privacy mixer protocol with an additional feature of de-anonymization of non-compliant users. This feature comes at the cost of increased trust in the entity maintaining the banned-addresses list, which can be realized in the two-server model. We further show how both our protocols can be extended to allow funds deposited to the mixer from banned addresses (i.e., funds that cannot be withdrawn) to be released to a predetermined trusted entity. This enables, for example, stolen funds to be returned to their rightful owner instead of being locked forever inside the mixer.

Formalization of compliance for mixers. In order to formalize compliance, we consider an idealized compliant ledger. In this ledger, deposits that become non-compliant are “removed” from the ledger, alongside the funds that are associated with them. This implies that mixer protocols in the idealized compliant ledger are compliant by default, as non-compliant deposits are not inside the mixer and hence cannot be withdrawn. Informally, our compliance definition is the following: a mixer protocol is compliant if it behaves indistinguishably in the idealized compliant ledger and the standard (append-only) ledger. Concretely, any accepted withdrawal transaction by the mixer is also accepted if the ledger is replaced with the idealized compliant ledger and vice versa. This definition coincides with the intuitive notion of compliance - illicit funds can’t go through the mixer.

Overview of our construction. Haze is comprised of two entities, a user and a mixer. The mixer is a smart contract implemented on the blockchain, and the user is a client run locally by any user wishing to interact with the smart contract of the mixer. Users interact with the mixer by depositing and withdrawing funds, by means of transactions on the blockchain.

Similarly to Tornado Cash, our solution utilizes Merkle trees and zero-knowledge proofs. Deposits are made by submitting a leaf to the Merkle tree maintained by the mixer. Withdrawals are made by users by creating a zero-knowledge proof that asserts that they have an unspent deposit from a *compliant address* in the mixer. The proof is sent to the mixer alongside a *nullifier*, where both are based on some secret information known only to the depositor. The nullifier is then stored in the smart contract and is used to ensure funds cannot be double-spent. The proof is constructed and verified with respect to the *compliant* Merkle tree, a tree where leaves associated with deposits from non-compliant addresses are removed.

A non-compliant address is an address on the banned-addresses list. The banned-addresses list is implemented as a smart contract on the blockchain, and maintained by a trusted entity. At withdrawal, the mixer queries this list in order to keep the compliant Merkle tree up-to-date, and if an address of a deposit had become non-compliant (banned), it

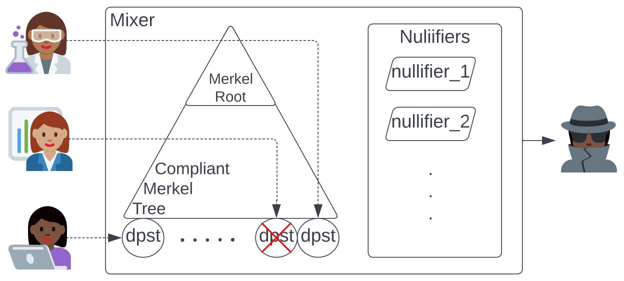


Fig. 2: Our privacy mixer with compliance, Haze. Deposits from non-compliant addresses are removed from the tree (the red X). The removal is triggered by a withdrawal transaction and done by zeroing their leaf value and updating the hashes at the nodes along the path from this leaf to the root. The privacy guarantee is that a withdrawal cannot be linked to its corresponding deposit.

removes the corresponding leaf from the Merkle tree and updates the relevant path in the tree accordingly. Funds belonging to removed leaves (equivalently, funds deposited from non-compliant addresses) are unrecoverable to the depositor, as it no longer can generate a successfully verifiable zero-knowledge proof to withdraw these funds, see Fig. 2. Therefore, Haze ensures that funds can never be withdrawn once the address they deposited from becomes non-compliant. Moreover, we emphasize that our technique for achieving compliance via removing leaves in the Merkle tree that are associated with non-compliant deposits can be applied with respect to any general compliance policy that can be checked against a deposit and not only policies defined by addresses. Thus, Haze captures a richer family of compliance policies.

Implementation. We implement both Haze's client (i.e., "user") in JavaScript and server (i.e., "mixer") in Solidity and evaluate the protocol's performance and gas consumption, demonstrating:

- No overhead at deposit. The gas consumption of a deposit transaction and the running time of the user are identical to Tornado Cash, i.e., $\sim$ 1M gas and 0.32 seconds, respectively.
- $\sim$ 1M gas consumption for the Merkle tree update per each newly non-compliant address that is associated with a leaf in the tree. The gas for the this update operation is paid by the withdrawal that triggers this

tree update (i.e., the first withdrawal transaction since this address entered the banned-addresses list).
- Negligible running time overhead at withdrawal. Concretely, 0.24 seconds user running time amortized per Merkle tree node, with a 0.0042 seconds difference from Tornado Cash. The time difference stems from fetching the banned-addresses list.
- The gas consumption per withdrawal is ~ 0.31M for the zero-knowledge proof and nullifier validation (as in Tornado Cash) plus ~ 1M gas per newly non-compliant address that requires an update of the Merkle tree.

The gas consumption per Merkle tree update is identical to the gas consumption of a deposit transaction, as both only require updating the relevant path in the tree. Therefore, a reasonable solution to cover the additional gas consumption of withdrawal due to compliance is to charge an extra fee per deposit proportional to the gas consumption of a single deposit. The legitimacy for the fee is by having each user cover not only the cost of the deposit itself but also the cost of preserving compliance in case its address becomes non-compliant. This approach makes a deposit transaction in Haze cost at most twice compared to Tornado Cash, while maintaining the cost of a withdrawal identical to Tornado Cash. Concretely, the cost of withdrawal is comprised of the gas consumption of the zero-knowledge and nullifier verification and an additive factor proportional to the number of newly non-compliant addresses associated with leaves in the tree. The extra cost paid at withdrawal for the Merkle tree updates is refunded to the withdrawer by the mixer, and funded by the extra fee charged with each deposit transaction. The cost overhead in the deposit is refunded to compliant users, as described in Section 4.4, making this a type of limited-time escrow. Overall, together with this feature, our protocol does not incur a cost overhead over Tornado Cash.

De-anonymizing non-compliant users. Due to the strong privacy guarantees of Haze, users that manage to withdraw funds prior to becoming non-compliant succeed in concealing the trace of their illicit funds. However, it might be desirable to construct a solution that enables a trusted entity to publicly revoke the privacy of users that become non-compliant, even if they became non-compliant after withdrawing funds from the mixer. To achieve this property we suggest an alternative construction of a compliant privacy mixer as follows: when depositing funds to the mixer, a user additionally provides an encryption of its nullifier, encrypted with the trusted entity's public key. Later, if a user becomes non-compliant, the trusted entity updates the banned-addresses list with the user's address and the nullifier in plaintext. Upon a withdrawal request, the mixer checks the banned-addresses list. Instead of updating the tree, it simply adds the nullifier of newly non-compliant addresses to the nullifier set, making funds of deposits from non-compliant addresses non withdrawable. We note that this solution eliminates the cost overhead at withdrawal, as no tree updates are required for newly non-compliant addresses. In this solution, the cost overhead of a deposit transaction is fixed at ~ 300K gas, due to a validation of a zero-knowledge proof provided with a deposit transaction for ensuring consistency of the encrypted nullifier. We note that for the protocol to guarantee privacy for compliant users, we assume that the trusted entity is semi-honest in the sense that it does not decrypt nullifiers' of compliant users. We suggest instantiating it in the two server model using a CCA-2 secure threshold encryption scheme, for example of [3]. This type of solution resembles real world scenarios such as police requiring search warrant from court to execute a search.

Releasing non-compliant funds. Our construction ensures that funds deposited into Haze from addresses on the banned-addresses list cannot be withdrawn. However, this might mean that stolen funds deposited into Haze are locked forever in the smart contract and cannot be returned to their rightful owners. For this reason, Haze can be deployed with a predefined life-cycle, defined in the smart contract implementing the protocol (i.e. is publicly known), and users are to withdraw their funds within the life of the mixer. After this time, the mixer is closed and remaining funds are transferred to a predetermined trusted entity. Additionally, at the time of closing the mixer, all compliant depositors are refunded the fee overhead they were charged at the time of deposit to cover the cost if they were to become non-compliant. The entity can be a hard-coded address of a law enforcement agency that can then redistribute these funds. The entity can also have a dispute resolution mechanism for individuals that claim to be wrongly placed on the list.

Due to Haze's strong privacy guarantee, counting the amount of funds from banned addresses inside Haze is a non-trivial task, as the privacy feature means that it is indistinguishable whether a non-compliant user withdrew its funds or not. Nonetheless, in our alternative protocol with the de-anonymiztation feature, the counting problem becomes easier. The mixer can count and release non-compliant funds at any desired period during the life-time of the mixer. This is elaborated in Section 6.

1.2 Related Work

Flavors of privacy over the blockchain. Prior to this paper, extensive work has been done towards ensuring transaction privacy on the blockchain, e.g., Hopwood *et al.*, Sasson *et al.* etc. [2, 5, 18, 29, 31, 38] are blockchain solutions that utilize cryptography to anonymize transactions. Most of them utilize Merkle trees and nullifiers, as in our construction. However, these solutions tend to be slow and expensive deeming them less popular for use in practice.

Other works, such as [10, 25, 30] by Malatova *et al*, Roos *et al.*, etc. provide privacy to layer 2 systems implemented on top of the blockchain. They however do not address the privacy of on-chain transactions.

Another desired flavor of privacy is untraceability of funds over the blockchain, that is commonly achieved through the use of privacy mixers. These are sometimes referred to as "add-on" privacy solutions that derive privacy by mixing a user's funds with many other funds. Mixers can be either centralized, see Bonneau *et al.*, Heilamn *et al.*, etc. [4, 17, 37], and depend on a trusted central entity or decentralized, see Meiklejohn *et el.*, Pertsev *et al.*, Bunz *et al.* etc. [5,26,28] which means that the functionality is implemented via an on-chain smart contract. Several papers quantify the privacy achieved by existing systems. For instance, Wu *et al.* [41] and Wang *et al.* [39]. However, none of these systems provide any guarantees of compliance. Moreover, some of these systems have been prone to abuse by money launderers, as mentioned above.

Compliance with privacy over the blockchain. Additionally, several papers have studied the intersection of privacy and compliance in the blockchain setting. In particular, Goldwasser *et al.* [15] proposes a protocol that enables to prove that specific regulations are being adhered to while maintaining secrecy of recorded data. Burleson *et al.* [6] were the first to introduce the question of compliance for privacy mixer, and state it in the sense of a banned-addresses list. They however do not provide a concrete construction, or definitions of the desired properties.

Paper Organization

The rest of this paper is organized as follows. Preliminary terminology and definitions appear in Section 2. Our protocol for compliant privacy mixer in Section 3. Details on integrating our protocol with the blockchain in

Section 4. The implemented system and empirical evaluation in Section 5. An alternative compliant privacy mixer protocol with de-anonymization of non-compliant users in Section 6. Conclusions in Section 7.

2 Preliminaries

We use standard definitions for functions being *negligible* with respect to a system parameter λ called the *security parameter*, denoted $\mathrm{negl}(\lambda)$; similarly for *polynomial*, where ppt stands for *probabilistic polynomial time* in λ. See definitions in [22].

In the following we establish definitions and terminology required for the rest of the paper.

Hash functions. We call an efficiently computable family of keyed functions

$$\mathcal{H} = \{H_s : \{0,1\}^* \to \{0,1\}^t\}_{t=t(\lambda),\, s\in\{0,1\}^\lambda,\, \lambda\in\mathbb{N}}$$

collision resistant hash functions, if for every ppt adversary $\mathcal{A}$, and any λ, a uniformly random function $H_s \in \mathcal{H}$ satisfies that $\mathcal{A}$ cannot find $x \neq x'$ s.t $H(x) = H(x')$, except with negligible probability.

Merkle trees. A Merkle tree $\mathcal{T}$ is a complete binary tree equipped with a collision-resistant hash function H and computed on n leaves having values $[v_1], \ldots, [v_n]$, and the value of each internal node is $H = (a||b)$ where a and b are the values of its two children; (We assume n is a power of 2; if not, we fix a zero value for the missing leaves). We use a standard notion where each leaf value $[v]_i$ is a hash of some cleartext data d_i. An *authentication path* $O(\mathcal{T}, \ell)$ of a leaf with position ℓ in $\mathcal{T}$ is made up of the values of all "sibling" nodes on the path from leaf ℓ to the root denoted $R_\mathcal{T}$, as well as $[v]_\ell$ itself and d_ℓ. We use the Pedersen hash function [19] for the leaf values, and the MiMC hash function [16], to compute the internal nodes of $\mathcal{T}$.

Zero-knowledge succinct non-interactive Argument of Knowledge [16]. Let R be a polynomial time decidable binary relation over pairs (ϕ, w) where ϕ is the statement and w the witness.

An efficient-prover publicly verifiable non-interactive argument $\Phi = (\mathsf{ZK.Setup}, \mathsf{ZK.Prove}, \mathsf{ZK.Ver}, \mathsf{ZK.Sim})$ for R is a quadruple of ppt algorithms as follows:

– $(\sigma, \tau) \leftarrow \mathsf{ZK.Setup}(R)$: The setup produces a common reference string σ and a simulation trapdoor τ for R.
– $\pi \leftarrow \mathsf{ZK.Prove}(R, \sigma, \phi, w)$: The prover algorithm takes as input a common reference string σ and $(\phi, w) \in R$ and returns an argument π.
– $0/1 \leftarrow \mathsf{ZK.Ver}(R, \sigma, \phi, \pi)$: The verification algorithm takes as input a common reference string σ, a statement ϕ and an argument π and returns 0 (reject) or 1 (accept).
– $\pi \leftarrow \mathsf{ZK.Sim}(R, \tau, \phi)$: The simulator takes as input a simulation trapdoor and statement ϕ and returns a simulated argument π.

Definition 1 (Succinct non-interactive zero-knowledge argument of knowledge). *We say $\Phi = (\mathsf{ZK.Setup}, \mathsf{ZK.Prove}, \mathsf{ZK.Ver}, \mathsf{ZK.Sim})$ is a* perfect succinct non-interactive zero-knowledge argument of knowledge (ZK-SNARK) *for R if it has:*

– *perfect completeness: Given any true statement, an honest prover should be able to convince an honest verifier to accept it. Formally, for all $(\phi, w) \in R$*

$$\Pr[\mathsf{ZK.Ver}(R, \sigma, \phi, \pi) = 1 | (\sigma, \tau) \leftarrow \mathsf{ZK.Setup}(R); \\ \pi \leftarrow \mathsf{ZK.Prove}(R, \sigma, \phi, w)] = 1$$

– *perfect zero-knowledge: An argument that does not leak any information besides the truth of the statement. Formally, for all $(\phi, w) \in R$ and all adversaries $\mathcal{A}$*

$$\begin{aligned} &\Pr[\mathcal{A}(R, \sigma, \tau, \pi) = 1 | (\sigma, \tau) \leftarrow \mathsf{ZK.Setup}(R); \\ &\qquad \pi \leftarrow \mathsf{ZK.Prove}(R, \sigma, \phi, w)] \\ = &\Pr[\mathcal{A}(R, \sigma, \tau, \pi) = 1 | (\sigma, \tau) \leftarrow \mathsf{ZK.Setup}(R); \\ &\qquad \pi \leftarrow \mathsf{ZK.Sim}(R, \tau, \phi)] \end{aligned}$$

– *computational knowledge soundness: There exists an extractor that extracts a witness whenever the adversary produces a valid argument (given access to its internal state). Formally, for all non-uniform polynomial time adversaries $\mathcal{A}$ there exists a non-uniform polynomial time extractor $\chi_{\mathcal{A}}$, and a negligible function* $\mathrm{negl}(\cdot)$ *such that,*

$$\begin{aligned} &\Pr[(\phi, w) \notin R \text{ and } \mathsf{ZK.Ver}(R, \sigma, \phi, \pi) = 1 | \\ &(\sigma, \tau) \leftarrow \mathsf{ZK.Setup}(R); ((\phi, \pi); w) \leftarrow (\mathcal{A} || \chi_{\mathcal{A}})(R, \sigma)] \\ &< \mathrm{negl}(\lambda) \end{aligned}$$

– *The proof π is of polynomial size in λ and* $\mathsf{ZK.Ver}$ *is polynomial in $\lambda + |\phi|$.*

3 Compliant Privacy Mixer

In this section we present our protocol for privacy mixers with compliance (Section 3.1, and Fig. 5) and provide a security analysis. Formal definitions for the properties achieved by our protocols and listed in this section are available in Appendix A.

3.1 Our Protocol

In this section we formally describe our protocol. We enhance the Tornado Cash protocol [28] to obtain compliance in the sense of preventing withdrawals of funds that belong to non-compliant deposits, without exposing information on the deposit being withdrawn, thus maintaining privacy of the user. A deposit is non-compliant if it was deposited from an address that has become non-compliant in the duration leading to the withdrawal attempt. The main difference between our protocol and [28] is in the withdrawal phase, where we first manipulate the Merkle tree to remove leaves corresponding to non-compliant deposits. This treatment guarantees that even if deposits become non-compliant after entrance to the mixer, they cannot be withdrawn. More formally,

The protocol $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ consists of a pair of protocols, deposit and withdrawal where any user Usr can communicate with Srv to perform the following functionality:

- $\mathsf{deposit}$ enables users to deposit money to Srv. Depositing is done by user Usr generating a deposit transaction of a fixed amount and communicating it to Srv.
- $\mathsf{withdraw}$ enables users to withdraw deposited funds from Srv. Withdrawing is done by user Usr generating a withdraw transaction (of the same fixed amount) using undisclosed data generated by Usr during the deposit phase and communicating it to Srv.

Communication between users and Srv is done via the *Bulletin board* $\mathcal{F}_{bb}$, a functionality that models the blockchain. $\mathcal{F}_{bb}$ supports the following requests from any entity in the (blockchain) system: Write a message, and Read written messages. Written messages are stored in an append-only linked list, where each list node is a tuple containing: its index in $\mathcal{F}_{bb}$, sender's and recipient's address, and the message itself. Similarly to the blockchain, entities in the system can generate and possess multiple addresses, where each address is unique (w.h.p), and there is no linkability between the addresses and the identity of the user. Read requests return the content of the linked list at the request time (i.e., up to the node with the most recent index). When a user executes either $\mathsf{deposit}$ or $\mathsf{withdraw}$, the generated transaction is written to $\mathcal{F}_{bb}$ with the address of Srv as the recipient's address, which is hard-coded within the protocol Π. The sender's address is recorded as well. Users can access $\mathcal{F}_{bb}$ from any address they own, but cannot use other entities' addresses (as in the blockchain, sending a message from an address requires signing the message with the secret key associated with the address being used). See Fig. 3 for formal details.

Bulletin-board Functionality $\mathcal{F}_{bb}$

Functionality $\mathcal{F}_{bb}$ proceeds as follows: Set $index = 0$

Upload. Upon receiving $(\mathsf{Write}, \text{msg}, \mathsf{address}_B)$ from some address $\mathsf{address}_A$, store the tuple $(index, \text{msg}, (\mathsf{address}_A, \mathsf{address}_B))$, output $(index, \text{msg}, \mathsf{address}_A)$ to $\mathsf{address}_B$, and set $index{+}{+}$.

Read. Upon receiving Read request from a party return all stored records in $\mathcal{F}_{bb}$.

Fig. 3: The bulletin-board functionality

Compliance in the context of privacy mixers requires rejecting deposits and withdrawals of funds associated with banned addresses. Concretely, since the banned addresses are dynamic, in the sense that new banned addresses are added from time to time, we consider an interactive *banned-addresses functionality* $\mathcal{F}_{ban}^{Q}$, that stores the banned addresses and is updated only by a predefined entity (with a fixed address Q). The banned-addresses list can be read by any entity in the system, and in particular, by users and Srv in Π. The decision on which addresses are updated in $\mathcal{F}_{ban}^{Q}$ is left outside the model. See Fig. 4 for formal details.

Our protocol. We present our privacy mixer protocol with compliance $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ in Fig. 5. Our protocol modifies the Tornado Cash protocol to obtain compliance in the sense that deposits from banned addresses cannot be withdrawn. That is, Π operates in the presence of $\mathcal{F}_{ban}^{Q}$ and rejects withdrawal of funds that were deposited from addresses that are recorded in $\mathcal{F}_{ban}^{Q}$ at the moment of withdrawal. This guarantees blocking banned addresses, that were not necessarily in $\mathcal{F}_{ban}^{Q}$ when the deposit transaction communicated to Srv. The protocols $\mathsf{deposit}$ and $\mathsf{withdraw}$

Banned-addresses Functionality $\mathcal{F}_{ban}^{Q}$

Functionality $\mathcal{F}_{ban}^{Q}$ is parameterized on address Q and proceeds as follows:

Update. Upon receiving $(\mathsf{Ban}, \mathsf{address}_A, \mathsf{data})$ from address Q, record $(\mathsf{address}_A, \mathsf{data})$.

Read. Upon receiving Read request from any party, return all stored records in $\mathcal{F}_{ban}^{Q}$.

Fig. 4: The banned-addresses functionality. Records are pairs of banned address, together with an (optional) field containing data related to the address.

are non-interactive in the sense that users communicate with Srv, but not vice versa, and Srv only performs Read requests to the $\mathcal{F}_{bb}$ and $\mathcal{F}_{ban}^{Q}$ functionalities. The communication to Srv (i.e., deposit and withdrawal transactions) is done by users sending a Write request to $\mathcal{F}_{bb}$ with the transaction and $\mathsf{address}_{\mathsf{Srv}}$ being the recipient's address.

The difference between our protocol and Tornado cash resides in the $\mathsf{withdraw}$ protocol. In the original Tornado Cash withdrawal protocol [28], when a user Usr wants to withdraw funds that it deposited in a deposit transaction dtxn, it proceeds as follows: (1) computes the root $R_{\mathcal{T}}$ of a Merkle tree $\mathcal{T}$, where the leaves of $\mathcal{T}$ are all deposit transactions submitted to Srv so far, where ℓ is the leaf associated with the deposit dtxn. Then, (2) Usr computes $O(\mathcal{T}, \ell)$, the authentication path of ℓ in $\mathcal{T}$ as defined in Section 2. Next, (3) it computes a hash, called nullifier, over part of the randomness used to generate dtxn. Finally, (4) Usr produces a proof that it "knows" the authentication path for one of the leaves in $\mathcal{T}$ that has not been previously withdrawn.

The proof in (4) is generated using a ZK-SNARK scheme Φ for a polynomial time decidable binary relation R, where the statement is ($R_{\mathcal{T}}$, nullifier) and the witness is $(\mathsf{randomnees}_{\mathsf{dtxn}}, \ell, O(\mathcal{T}, \ell))$. The withdrawal transaction submitted by Usr consists of (nullifier, proof).

Upon receiving the withdrawal request, Srv fetches its locally stored $\mathcal{T}$ and verifies the proof wrt its root and the received nullifier (in addition to the nullifier uniqueness assertion). In our withdrawal protocol, we modify $\mathcal{T}$ and nullify the leaves that correspond to deposit transactions associated with an address that appears in $\mathcal{F}_{ban}^{Q}$. Consequently, if the ZK-SNARK proof verifies, it guarantees that the deposit transaction it withdraws is not from an address in $\mathcal{F}_{ban}^{Q}$, as those do not appear in $\mathcal{T}$ anymore.

The formal description of our protocol Π appears in Fig. 5.

Our protocols $\mathsf{deposit}$ and $\mathsf{withdraw}$ in Π provide the same time complexity as Tornado Cash except for an additive factor in $\mathsf{withdraw}$ for Srv, that is, for a security parameter λ:

– $\mathsf{deposit}$ has time complexity of Usr and Srv, which is $\text{poly}(\lambda)$ and $\text{poly}(\lambda) \cdot O(\log(n))$, respectively.
– $\mathsf{withdraw}$ has a Usr time complexity of $\text{poly}(\lambda) \cdot O(n \cdot \log(n))$, and an additive factor of $\Delta \cdot \text{poly}(\lambda) \cdot O(\log(n))$ on the Srv side, compared to Tornado Cash.

where n is the maximal number of leaves in $\mathcal{T}$, and Δ is the number of added addresses to $\mathcal{F}_{ban}^{Q}$, since the previous withdrawal transaction, that are associated with leaves in $\mathcal{T}$. See Section 5 for performance measurements of $\mathsf{withdraw}$ and $\mathsf{deposit}$.

Our protocol Π provides correctness, soundness, privacy, and compliance in the following sense. Detailed formalization these properties appears in Appendix A.

Correctness is in the sense that any deposited funds can be withdrawn (once) as long as the matching deposit transaction is compliant at the time of the withdrawal, i.e., the withdrawn funds were not deposited from an address in $\mathcal{F}_{ban}^{Q}$. Correctness stems from the collision resistance of $H(\cdot)$ together with the completeness property of Φ, and k being randomly sampled. Concretely, a valid deposit transaction $H(k||r)$ is a leaf in $\mathcal{T}$ as long as it is not from an address in $\mathcal{F}_{ban}^{Q}$. Therefore, on input (k, r) the $\mathsf{withdraw}$ protocol produces an accepting zero-knowledge proof π and a unique nullifier h.

Soundness is in the sense that a user cannot withdraw funds that it did not deposit. Soundness stems from the collision resistance of H, and the computational knowledge soundness of Φ. That is, a user that produces a valid proof for the instance $(R_{\mathcal{T}}, H(k))$, without possessing (k, r) for one of the leaves, can be used to break the collision resistance property of H. This is done by applying the knowledge extractor χ_{Usr}, guaranteed by the knowledge soundness of Π, to extract the witness $(k, r, \ell, O(\mathcal{T}, \ell))$ with non-negligible probability.

Privacy is in the sense that a withdrawal cannot be linked to any non withdrawn deposit. Privacy stems from the property of H being a random

Common parameters: A security parameter λ, a function H sampled uniformly at random from a collision-resistant hash function family $\mathcal{H}$, and a ZK-SNARK scheme $\Phi = (\mathsf{ZK.Setup}, \mathsf{ZK.Prove}, \mathsf{ZK.Ver}, \mathsf{ZK.Sim})$ for relation R as defined above.

Parties and addresses: A mixer Srv with public address $\mathsf{address}_{\mathsf{Srv}}$ and a user Usr with some addresses $\mathsf{address}_{\mathsf{Usr}_1}$ and $\mathsf{address}_{\mathsf{Usr}_2}$.

Storage: Srv locally stores a full binary Merkle tree $\mathcal{T}$ on n leaves, all initialized to zero, the location of the next available leaf $next = 0$, and, initially empty, nullifier set $\mathrm{S}_{\emptyset}$. Each leaf in $\mathcal{T}$ is associated with an address, initially set to $\perp$.

Trusted setup: Bulletin-board functionality $\mathcal{F}_{bb}$, and Banned-addresses Functionality $\mathcal{F}_{ban}^{Q}$. A common reference string σ produced by running $\mathsf{ZK.Setup}$.

Deposit: $\mathsf{deposit}$ is executed by user Usr from some address $\mathsf{address}_{\mathsf{Usr}_1}$ and mixer Srv, as follows:

1. **User:** samples uniformly at random $k, r \leftarrow \{0,1\}^{t(\lambda)}$ for some polynomial $t(\cdot)$, computes $C = H(k||r)$, and sends $(\mathsf{Write}, C, \mathsf{address}_{\mathsf{Srv}})$ to $\mathcal{F}_{bb}$. We call C a *deposit transaction.*
2. **Mixer:** Upon receiving $(index, C, \mathsf{address}_{\mathsf{Usr}_1})$ from $\mathcal{F}_{bb}$, perform the following steps:
   (a) Check that C is in the range of H (else output 0).
   (b) Invoke the subroutine in Fig. 6 on $\mathsf{Update\_Tree}(\mathcal{T}, next, C, \mathsf{address}_{\mathsf{Usr}_1})$ to update $\mathcal{T}$, set $next = next+1$, and output 1 if successful. We call such a deposit transaction valid. [a]

Withdrawal: $\mathsf{withdraw}$ is executed by user Usr from some address $\mathsf{address}_{\mathsf{Usr}_2}$ and mixer Srv, as follows:

1. **User:** On input (k, r), to withdraw a deposit transaction $C = H(k||r)$ proceed as following:
   (a) Send Read request to $\mathcal{F}_{ban}^{Q}$ and denote by S_{ban} the received banned-addresses list.
   (b) Send Read request to $\mathcal{F}_{bb}$ and denote by $\mathrm{S}_{leaves} = \{(index_i, C_i, \mathsf{address}_i)\}_{i \in [k]}$ the subset of the returned records from $\mathcal{F}_{bb}$ where C_i is a valid deposit transaction, and $\mathsf{address}_i$ is the address associated with it. For each $i \in [k]$ such that $\mathsf{address}_i \in \mathrm{S}_{ban}$ set $C_i = 0$ and $\mathsf{address}_i = \perp$.
   (c) Construct a Merkle tree $\mathcal{T}$ with $(C_1, \ldots, C_k)$ being the leaves and let $R_{\mathcal{T}}$ be the root of $\mathcal{T}$.
   (d) Compute the authentication path $O(\mathcal{T}, \ell)$, where ℓ is the leaf index of C in the computed $\mathcal{T}$ (if no leaf C, abort).
   (e) Compute $h = H(k)$ and $\pi \leftarrow \mathsf{ZK.Prove}(\sigma, (R_{\mathcal{T}}, h), (k, r, \ell, O(\mathcal{T}, \ell)))$.
   (f) Send $(\mathsf{Write}, (h, \pi), \mathsf{address}_{\mathsf{Srv}})$ to $\mathcal{F}_{bb}$. We call (h, π) a *withdrawal transaction* and h its *nullifier.*
2. **Mixer:** Upon receiving $(index, (h, \pi), \mathsf{address}_{\mathsf{Usr}_2})$ from $\mathcal{F}_{bb}$, perform the following steps:
   (a) Send Read request to $\mathcal{F}_{ban}^{Q}$ and denote by S_{ban} the received banned-addresses list.
   (b) Zero leaves associated with deposits from banned addresses: For each leaf ℓ in $\mathcal{T}$ and an address $\mathsf{address}_{\ell}$ associated with it, check if $\mathsf{address}_{\ell} \in \mathrm{S}_{ban}$, and if so invoke the subroutine in Fig. 6 on $\mathsf{Update\_Tree}(\mathcal{T}, \ell, 0, \perp)$ to update $\mathcal{T}$.
   (c) Verify that h did not appear in any previous withdrawal transaction (output 0 otherwise).
   (d) Output $b \leftarrow \mathsf{ZK.Ver}(\sigma, (R_{\mathcal{T}}, h), \pi)$ and if $b = 1$ set $\mathrm{S}_{\emptyset} = \mathrm{S}_{\emptyset} \cup h$.

[a] in the blockchain implementation the mixer also verifies that the deposit transaction is funded.

Fig. 5: Compliant privacy mixer protocol Π

Subroutine Update_Tree executed by Srv on $(\mathcal{T}, \ell, C, \mathsf{address})$, and shared parameters as in Fig. 5, where $\mathcal{T}$ is a Merkle tree on n leaves (and height $\log(n)$).

For a node $v \in \mathcal{T}$ we denote by $[v]$ its value, and similarly by $[v]_{\mathrm{sb}}$ and $[v]_{\mathrm{pr}}$ its sibling and parent values, respectively.

The subroutine proceeds as follows:

```
1. Set the value of the $\ell$'th leaf to $C$ and denote this leaf by $v$.
2. while $v \neq R_{\mathcal{T}}$ compute:
   (a) $[v]_{\mathrm{pr}} = H([v]||[v]_{\mathrm{sb}})$ for left child $v$ and
       $[v]_{\mathrm{pr}} = H([v]_{\mathrm{sb}}||[v])$ for right child $v$.
   (b) $v = \mathrm{parent}(v)$
```

Fig. 6: The subroutine Update_Tree updates the hashes along the path from the ℓ'th leaf to the root in $\mathcal{T}$.

oracle (or alternatively H is hiding) and the zero-knowledge property of Φ. Concretely, the zero-knowledge proof guarantees to hide $(k, r, \ell, O(\mathcal{T}, \ell))$ and the random oracle H guarantees that $H(k)$ can be linked to $H(k||r)$ w.p at most negligibly larger than a random guess.

Compliance is in the sense that funds deposited from an addresses in $\mathcal{F}_{ban}^{Q}$ cannot be withdrawn. This follows immediately from the construction as leaves associated with deposit transactions from banned addresses are zeroed and do not appear in $\mathcal{T}$. Therefore, depositors from addresses in $\mathcal{F}_{ban}^{Q}$ cannot produce an accepting nullified and zero-knowledge proof to withdraw these funds.

See Appendix A for formal definition of these properties.

4 Integrating with the Blockchain

In this section we address the necessary adjustments needed to make our protocol in Fig. 5 securely deployed on a blockchain. In particular, it needs to remain correct, compliant, private, and sound on the blockchain. Our protocol uses primitives that are available on many contemporary platforms, and in particular all EVM blockchains, and therefore is broadly applicable. In Section 4.3 we introduce additions to our protocol to enable balanced distribution of the cost overhead induced by the compliance maintenance mechanism. Additionally, in Section 4.4, we propose a solution that enables releasing funds deposited from non-compliant addresses back to some predetermined entity.

4.1 Deployment on the Blockchain

In the blockchain deployment of our protocol, the *mixer* Srv is an on-chain smart contract that implements the logic of Fig. 5, and can be publicly audited. The Merkle tree $\mathcal{T}$ is stored in the smart contract's storage. Users interact with Srv by sending transactions to the blockchain. The address of a transaction sender is publicly visible and therefore a deposit to Srv can be linked to the address from which it originated. However, a user may send many transactions to Srv from multiple addresses.

The *Banned-addresses list functionality* $\mathcal{F}_{ban}^{Q}$ is an on-chain smart contract which receives queries from users and asserts whether an address has been included in the banned address list (sanctions list). Today, the company Chainalysis maintains such a contract [7] on the Ethereum blockchain and reflects the sanctions designations listed on economic/trade embargo lists from governments and organizations including the US, EU, and the UN.

A deposit transaction is an on-chain transaction transferring M coins to the smart contract Srv (we assume for ease of notations that $M = 1$, but any amount will work as long as it is the same amount across all users and all transactions). The transaction will also have as auxiliary data $C = H(k||r)$ as described in $\mathsf{deposit}$, Fig. 5, Step 1. The user must also include in the transaction additional funds to pay the gas fees.

A withdrawal transaction is an on-chain transaction from the user to Srv. In order to maintain our protocol's privacy guarantees, the transaction should originate from a previously unused address. The transaction does not transfer any funds into Srv, and includes in the auxiliary data the user inputs as described in $\mathsf{withdraw}$, Fig. 5, Step 1. When the withdrawal is processed by Srv, M coins will be released to the address designated in the transaction. The transaction should again include the gas fee needed to execute the withdrawal. Paying for gas of a withdrawal is preferably done via a relay in order to preserve privacy, see Section 4.2.

4.2 Security Concerns over the Blockchain

Care needs to be taken in order to deploy our protocol (we call the deployed protocol Haze) on top of the blockchain, due to the asynchronous nature of the blockchain and the fact that messages are not written to the blockchain directly by users. The blockchain is indeed an append-only linked list, as per our theoretical model. However, messages may arrive at the blockchain simultaneously and several messages may be included in a single block (a single state update). Messages are sent to the blockchain either via broadcast, through a trusted relay or directly to validators through private channels. Block builders\validators choose how to order transactions inside a block according to their own best interest (typically according to a tip-maximizing order) [8]. This means that in addition to the security properties mentioned above, over blockchains, several other security concerns arise. For example, the deployed protocol needs to guarantee resilience to hijacking and front-running.[3] We therefore have to take extra precautions when implementing our protocol as a smart contract to mitigate these concerns.

Hijacking resilience. In the idealized bulletin-board model, messages are written to $\mathcal{F}_{bb}$ directly without the possibility of interception. In the blockchain world messages are sent either via broadcast to the entire network, or through private channels to builders\validators. This introduces a previously undiscussed risk of messages being hijacked and modified in order to steal funds headed out of Srv. Haze prevents hijacking by including the recipient address in the zero-knowledge proof included in the withdrawal transaction.

Front-running resilience. The front-running problem arises when Alice issues a withdrawing transaction w.r.t some state, and before Alice's withdrawal is included in a block, a new deposit or withdrawal by Bob is made to the mixer and is included in a block, thus changing the state of the mixer and potentially deeming Alice's withdrawal invalid [24] (meaning Bob's transaction front-ran Alice's). Due to this concern, we extend Haze to be front-running resilient. Haze enables a withdrawal to reference any previous root, as long as there were no updates to the banned-addresses list since that root was valid. In this case, the previous root is simulated ("zeroing" the relevant deposits) which can be done in $O(\log(n))$ time, where n is the number of leaves in $\mathcal{T}$. This means the withdrawal transaction will be processed correctly.

Withdrawal gas fees. In order to pay the gas fee of a withdrawal transaction, the initiator of the transaction needs to have sufficient funds. The recipient address of a withdrawal needs to be a fresh address to maintain

[3] We note that the protocol is replay resilience due to its soundness guarantee.

the privacy of our protocol. If the withdrawal were initiated by a fresh address, that address would need to somehow have these funds. However, the withdrawal address needs to be unlinkable to the address of the depositor. So the depositor can't simply fund this fresh address to pay the gas fees. This raises the question of how the gas fees can still be paid. For this reason, users should utilize relays to minimize the privacy loss. Relays exist in the original Tornado Cash implementation [28]. A relay receives a withdrawal transaction from the depositor via a private secure channel. In the recipient field of the withdrawal, the depositor will list a fresh address. The relay funds the gas costs for the withdrawal and forwards the withdrawal transaction to Srv. The relay cannot alter the recipient field since Haze is hijacking resilient. Srv processes the withdrawal and releases the funds to the fresh recipient address, minus a fee paid to the relay and the gas cost for the withdrawal transaction which are sent to the relay. This way, the withdrawal request cannot be linked to the depositor's address on-chain. However, this requires trust between the depositor and the relay, as the relay can link the depositor to the withdrawal request. See Fig. 7.

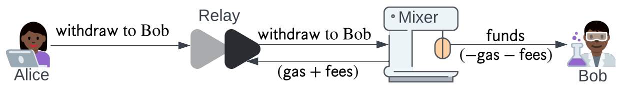


Fig. 7: In order to pay the gas costs of the withdrawal transaction withdraw addressed to Bob, the depositor Alice should utilize a relay. Alice and the relay communicate through a private channel. The relay forwards the withdrawal transaction to Srv. Srv then processes the withdrawal and sends the released funds to Bob, minus the gas cost and fee that is sent back to the relay. This way Alice remain unlinkable to Bob on the blockchain.

4.3 Economic Concerns over the Blockchain

In addition to the concerns addressed above, when implemented over the blockchain, Haze needs to not fail due to gas limitations. Concretely, the number of tree updates required per withdrawal depend on the newly banned addresses, thus making the cost of withdrawal non uniform per withdrawal and a priori unpredictable. In this section we provide a twofold treatment: once at the feasibility level, enforcing that such updates do not fail due to technical limitation of the host blockchain, and at the user level: a single user should not be made to cover the cost of a large update due to "bad timing". We also implement and evaluate the gas cost of a deposit and a withdraw operation in Section 5.

Balancing the gas costs associated with withdraw. As mentioned in Section 3, enforcing compliance introduces a cost proportional to the number of newly non-compliant deposits times the cost of a single path update in $\mathcal{T}$. This implies that the cost of withdrawal depends on the number of newly non-compliant deposits and hence is a priori unpredictable and also non uniform on all withdrawals. In order to resolve this issue and create uniformity in the cost of withdrawals, we suggest creating a fund in the smart contract of Srv which will be funded by a depositor fee for every deposit to refund these users that happen to bear the cost of updates induced by changes in the banned-addresses list. As an update costs the same amount of gas as a deposit, the maximal amount of fee Haze needs to charge in order to cover these costs (per deposit) is bounded by the gas fee per a deposit transaction. This makes the price of a deposit increase by at most a factor of 2. In Section 4.4 we show how this overhead can be refunded to compliant users, making the overall cost of using Haze comparable to Tornado Cash.

The cost of banned-addresses list spamming. In our protocol, whenever there is an update to the banned-addresses list, an update to the Merkle tree $\mathcal{T}$ is necessary. An attacker that wants to disrupt the operation of Haze might therefore be incentivized to spam the banned-addresses list in order to make its operation expensive. However, the cost of the attack grows linearly with the number of spammed addresses, which makes this type of attack less appealing. This is due to the fact that Haze only takes into account banned-addresses that have entered Srv. Since entering Srv requires depositing funds, spamming this intersection becomes expensive as these funds cannot be retrieved due to non-compliance of the addresses from which they originated.

Overcoming block gas limit for large simultaneous updates to $\mathcal{T}$. We first recall again that the only updates to the banned-addresses list of interest to Haze are ones of addresses that have made deposits into Haze. This limits the number of updates Haze needs to handle, as there potentially can be many updates to the banned-addresses list, and only a small fraction might affect Haze. Additionally, at every withdrawal request, Haze only needs to consider changes to the banned-addresses list that have occurred since the last withdrawal request was processed, which further limits the number of updates.

Even so, if a withdrawal transaction invokes a number of path updates in $\mathcal{T}$ that require gas that exceeds the gas limit for a single block, it could potentially fail. To mitigate this issue, in the blockchain implementation of Srv, the withdraw function is split into two functions - withdraw and update. withdraw handles the withdrawal logic, while update handles the logic for updating $\mathcal{T}$ according to the banned-addresses list. This way, if the backlog of updates is too big for a single withdrawal transaction to be processed, any user in the system can call the update function to relieve the backlog. The gas for this altruistic transaction can be funded by the fees collected upon deposit, and will be refunded to the caller of the update function. This solution is compatible with the incentives of users of Haze who want to be able to withdraw funds from Srv.

4.4 Releasing Non-compliant Funds

While our protocol in Fig. 5 is compliant in the sense that funds deposited from banned addresses cannot be withdrawn, it leads to the problem of these funds being permanently locked in the mixer. For this reason, we propose a mechanism that enables releasing these funds to a predetermined entity. We recall that due to the privacy property of Haze, it is indistinguishable whether a non-compliant user withdrew its funds or not, and thus counting the amount of non withdrawn funds from banned addresses inside Haze is difficult.

Our solution works as follows: the mixer will have a limited life-cycle of some predetermined amount of time[4]. At the end of its life-cycle, the mixer will no longer take new deposits and there will be a period in which all users are allowed to withdraw their remaining funds. At the end of this period, all funds that are not withdrawn are transferred to the predetermined entity. This entity can implement a dispute process in which users with compliant funds that for some reason did not withdraw their funds in time can request their funds by exposing the (k, r) that are associated with the disputed funds (this causes a privacy loss for the user). At the end of the mixer's life-cycle, Haze also enables refunding compliant users the fee overhead they paid at the time of deposit. We remind that the purpose of this overhead is to cover the cost of the tree update in the event a deposit would ever become non compliant. Since

[4] Time is measured in blocks.

at the end of the mixer's life-cycle the deposit is still compliant, this cost will never be realized. Therefore, Haze can safely refund the remaining fees to each compliant address that deposited funds to the mixer.

Recall that on the blockchain, a smart contract can only be triggered by a transaction signed by a user. We therefore design Haze in a way that enables anyone, including the trusted entity, to trigger the end-of-life of the mixer. There will be a hard-coded condition in the smart contract to verify that enough blocks have been added since the creation of the contract, and only upon meeting that requirement, can the end-of life be triggered. The refund to compliant users guarantees the incentive to trigger the end-of-life of the mixer. This construction maintains the properties of Haze: compliance, correctness, privacy, and soundness.

5 Empirical Evaluation

In this section we implement and empirically evaluate the performance of our protocol from Fig. 5 and compare it to Tornado Cash [28].

Setup In our experiments, we used an Intel(R) Core(TM) i9-12900H machine with 32G RAM to run client Usr. We deploy and manage the mixer Srv on an Ethereum local blockchain using Ganache [14].

5.1 Implementation

Our implementation comprises of two components:

– Server Srv: the mixer, implemented as a smart contract in Solidity [11].
– Client Usr: the user, implemented in JavaScript.

Both are implemented using a fork of the Tornado Cash mixer [32] and client [33] extended to our protocol. As in Tornado Cash, the hash functions used are the Pedersen hash function [19] and the MiMC hash function [1], which are implemented in the circomlib library [20]. The SNARK keypair and the Solidity verifier code are generated using SnarkJS [21], It uses the Groth16 [16] Protocol (3 point only and 3 pairings), PLONK [13] and FFLONK [12].

The changes we introduced in the code are as follows:

– The smart contract of the mixer now calls an external smart contract that manages the banned-addresses list.

– We store the Merkle tree $\mathcal{T}$ in the smart contract as a map with the node indices as keys. The map grows gradually as deposits enter our system. In addition we maintain a map of depositors' addresses to the indices of the tree leaves representing their deposits. This map is used to efficiently locate leaves in the tree associated with non-compliant deposits.
– We maintain a queue of the indices of the Merkle tree leaves associated with all non-compliant addresses that are currently known and not yet zeroed in the Merkle tree, based on the list of banned-addresses obtained from the external contract.
– Path updates for preserving the compliance of the Merkle tree are realized as follows: We implement an update function, that on input n zeroes the leaves associated with the first n indices in the queue and updates the values along their path to the root accordingly. We allow $n \leq 35$ as this is the maximum possible number of leaves that can be handled in a single transaction within the block gas limit of 30M, see Fig. 9a. We change the implementation of the withdraw function to call the update function as part of its internal logic.

5.2 Experiments and Results

We evaluate Haze's performance in terms of gas consumption and running time of the client, for deposits and withdrawals in different scenarios. We compare our measurements to Tornado Cash.

In all our experiments, we measure the actual gas consumption using the transaction receipt "gasUsed" field of corresponding requests. We set the block gas limit to $30 \cdot 10^6$ which is the gas limit used by the Ethereum mainnet today. We repeat each experiment 20 times and present the average result of all repetitions.

The purpose of our first experiment is to show that when there are no non-compliant addresses the gas consumption of the smart contract and running time of the client of our Haze is comparable to Tornado Cash, which is widely used.

Experiment A: comparison of our protocol to Tornado Cash. We measure the cost of deployment of Haze to the blockchain, as well as the cost of a single deposit and withdrawal when there are no non-compliant addresses (we denote it the baseline setting). We run the deposit and withdrawal measurements on the client side as well. We compare the baseline gas costs and running time of our protocol with the deposit and withdrawal of Tornado Cash.

The results appear in Table 1. We see that on average the gas cost has increased by 1.1% for our mixer compared to Tornado Cash for a withdrawal transaction, by 4.1% for a deposit transaction, and by 7.2% for the deployment. These increases are explained by the additional storage in Haze compared to Tornado Cash. The running time of the deposit on the client side of our protocol is identical to Tornado Cash and the withdrawal running time increases by 5.7% on average. This difference stems from the fact that in our implementation, in each withdrawal, the client checks for updates of the banned-addresses list.

	Server (gas)		Client (sec)	
Action	Tornado Cash	Ours (base line)	Tornado Cash	Ours (base line)
Deploy	1960209	2099721	—	—
Deposit	957037	996139	0.32	0.32
Withdraw	312254	315782	7.78	8.22

Table 1: Comparison of our protocol in the baseline setting vs. the Tornado Cash, for smart contract deployment (gas units), deposit (seconds), and withdrawal (gas units). Measurements taken when the mixer is populated with 1K deposits.

Experiment B: Gas cost vs. number of non-compliant addresses. After populating the mixer with 1K deposits we measure the gas consumption of a deposit transaction and a withdrawal transaction as the number of deposits associated with newly non-compliant addresses increases.

We find that the gas consumption of withdrawal increases linearly with the number of deposits associated with non-compliant addresses. For completeness, we also present the gas consumption of a deposit operation, which does not change with the number of non-compliant addresses. Moreover, we find that the maximal number of tree updates due to non-compliant address that can be supported in one transaction is at most 35, since after that the gas required surpasses the block gas limit of 30M gas. The results are summarized in Fig. 8. We note that the results of this experiment are independent of the number of deposits that were made to the mixer prior to the measurement. We verified this independence by repeating the experiment when populating the mixer with different numbers of deposits in $\{1, \ldots, 1000\}$, checking at increments of 100, and obtained the same results. The gas consumption of a withdrawal transaction consists of the zero-knowledge proof verification and tree updates, which result from each newly non-compliant address. The number of deposits in the mixer does not influence either of these components, as the updates depend only on tree height which is fixed throughout the lifetime of the mixer. Similarly, a deposit transaction depends on the tree height and is oblivious of the number of deposits inside the mixer, as well the number of non-compliant addresses.

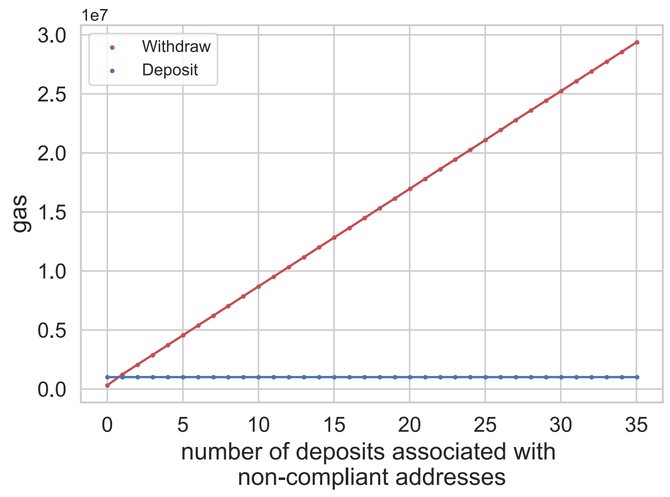


Fig. 8: The deposit cost (blue) and withdrawal cost (red) of our protocol in gas units vs. the number of non-compliant newly non-compliant addresses as part of the upcoming withdrawal transaction. Measurements taken when the mixer is populated with 1K deposits.

Experiment C: Running time vs. number of deposits in the mixer. We measure the client's withdrawal running time as the number of deposits in the mixer increases in the baseline setting (i.e., there are no non-compliant addresses).

Results are summarized in Fig. 9a. We see that the running time of the client for the withdrawal increases linearly with the number of deposits in the mixer. The increase in running time stems from increasing number of nodes in the constructed tree by the client.

Next, we assert that the number of newly non-compliant addresses does not influence the running of the client both for deposit and withdrawal transactions. More formally,

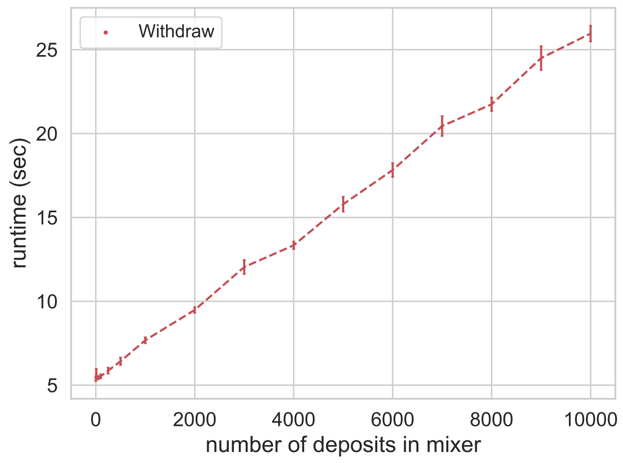
Experiment D: Running time vs. number of non-compliant addresses After populating the mixer with 1K deposits we measure the running time of the client for a deposit transaction and a withdrawal transaction as the number of deposits associated with newly non-compliant addresses increases.

The results are summarized in Fig. 9b. We find that the running time of withdrawals is unaffected by the number of non-compliant addresses, per fixed number of deposit populating the mixer. For different numbers of mixer deposit populations, we get similar constant lines, in accordance to the results in Fig. 9a. We verified this by repeating the measurements for different deposit populations in the mixer, for the same values presented in Fig. 9a. This is due to the fact that the client, similar to Tornado Cash, rebuilds the entire tree in each withdrawal request. Moreover, the deposit running time does not change with the amount of non-compliant addresses and the number of deposits in the mixer, similarly as for gas cost of deposits.

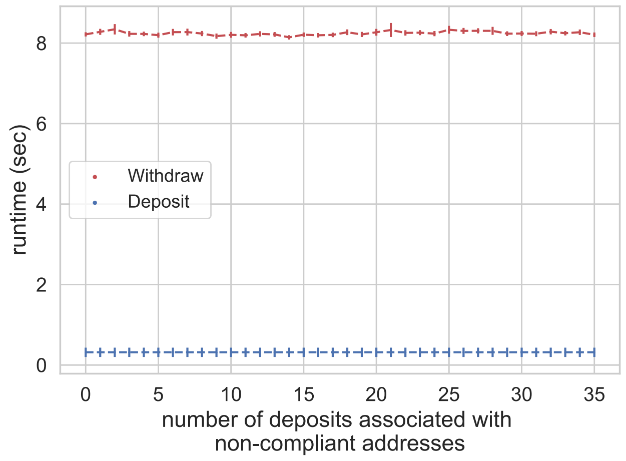
6 De-anonymizing Non-compliant Users

In this section we propose a protocol for compliant privacy mixer that provides de-anonymization of non-compliant users. Our protocol in Section 3 maintains the privacy of users, even in the event they become non-compliant after withdrawing their funds. In such cases, a user successfully launders money and cannot be traced. This strong post-withdrawal privacy guarantee for deposits that became non-compliant might not be acceptable in practice. For instance, funds of non-compliant users are not traceable even in the event of a court order.

Due to this concern, we suggest an alternative protocol to de-anonymize withdrawals of funds that originated from non-compliant addresses, even if these funds were successfully withdrawn from the mixer. This requires enhanced trust in the predetermined trusted entity that manages the banned-addresses list. We emphasize that in the case the trusted entity is corrupted, it can potentially violate the privacy of all users, compliant



(a) The time in seconds it takes the client to prepare a `withdraw` transaction. This measurement is taken in the baseline setting when there are no non-compliant addresses.



(b) The running time of the client in deposit and withdrawal in seconds as the numbers of newly non-compliant addresses increases. Measurements taken when the mixer is populated with 1K deposits.

Fig. 9: Results of Section 5.2 and Section 5.2

and non-compliant alike. The privacy revocation, however does not give the trusted entity any advantage in stealing funds from the mixer.

The protocol presented in this section has the following advantages: (1) Revoking privacy of non-compliant users and (2) less funds are needed upfront for deposit, and no cost overhead at withdrawal (3) releasing non-compliant funds does not require active participation of the users, and does not terminate the operation of mixer.

More formally, the protocol presented in this section, denoted by Ψ, relies on a trusted entity Q that in addition to uploading the non-compliant addresses to $\mathcal{F}_{ban}^{Q}$, also provides for each such address a data field that enables to publicly disclose the trace of funds deposited from this address to the mixer. Concretely, it enables linking the withdrawal to the address of the deposit, and hence revoking privacy. The privacy and compliance of the protocol rely on Q to perform the data extraction (1) correctly and (2) only on deposits to the mixer that are from non-compliant addresses. We discuss an optional realization of Q later in this section. The protocol Ψ instructs the user to encrypt its nullifier as part of the deposit, and in case its address becomes non-compliant the trusted entity decrypts and publishes the nullifier. The idea in Ψ is to guarantee compliance by blocking withdrawals that present a nullifier that is associated with a non-compliant address. Concretely:

– In $\mathsf{deposit}$, in addition to sampling k, r and computing $H(k||r)$, the Usr computes an encryption of the nullifier under the public-key pk_Q of the trusted entity Q, and a "consistency" proof. The proof is generated using a ZK-SNARK scheme Φ for a polynomial time decidable binary relation $R_{consist}$, where the statement is $(H(k||r), pk_Q, \text{ciphertext})$ and the witness is $(k, r, \text{ciphertext randomness})$. The deposit transaction submitted by Usr consists of $((H(k||r), \text{ciphertext}), \text{proof})$.

– The $\mathsf{withdraw}$ protocol of Ψ differs from Tornado Cash only in its nullifier treatment. That is, the Srv compares the nullifier in the withdraw transaction not only to nullifiers of prior withdrawals but also to the nullifiers in the data field in $\mathcal{F}_{ban}^{Q}$, and rejects withdrawal if appears in either.

We emphasize that in Ψ the mixer is not required to maintain a compliant Merkle tree, and in particular does not perform any tree updates for non-compliant leaves. This significantly reduces the gas consumption of Ψ compared to the protocol in Fig. 5. The formal description of our protocol Ψ appears in Fig. 10.

The protocol Ψ provides correctness, privacy, soundness and compliance in the same sense as in Section 3, where privacy is guaranteed only for compliant users.

Instantiating the trusted entity Q. Motivated by real world settings, such as search warrants, where authorities that have achieved some threshold of permissions can "de-anonymize" a user's call log, bank transactions, etc. We follow a similar approach and reduce trust in Q by adding another (non-colluding) server B and replacing the CPA-secure scheme in Fig. 10 with a CCA-2 secure threshold encryption as defined in Devevey *et al.*, section 2.4 in [9]. For example, Ψ can be instantiated with the scheme of Boneh *et al.* [3]. The trusted entity Q and B hold key shares and together decrypt nullifiers associated with deposits from non-compliant addresses by running threshold decryption. To ensure only privacy of non-compliant users is revoked, Q is required to submit to B a non-compliant address and a proof of it being non-compliant in order for B to engage in the decryption process. We note as long as Q and B are not colluding, the privacy of compliant users is maintained.

Releasing non-compliant funds. In the protocol presented in this section the trusted entity publicly exposed the nullifier of non-compliant users, which makes the counting of non withdrawn banned funds easy. This is since in Ψ we can distinguish if a non-compliant deposit has already been withdrawn or not. This makes it so that the mixer can count banned funds at any desired period during the life-time of the mixer. This amount can be released to some predetermined trusted entity.

7 Conclusions

In this work we presented a compliant privacy mixer that is the first to attain all the following desired properties: correctness, soundness, privacy, and compliance. We ran extensive experiments using Solidity for the mixer Srv and JavaScript for the client Usr, demonstrating efficient user running time (0.32s per deposit and 8.22s per withdrawal for 1K deposits to the mixer in the client) and realistic gas requirements comparable to the standard protocol of Tornado Cash (1M gas per deposit and $\sim$ 315K gas per basic withdrawal plus 1M gas per update that is eventually paid only by non-compliant users). Our protocol can be deployed and used over the blockchain guaranteeing resiliency against: transaction hijacking, front-running, and banned-addresses list spamming. Moreover, our solution can be instantiated for any banning policy that can be applied on deposit transactions. We introduce an alternative protocol for a compliant privacy mixer, that supports de-anonymization of non-compliant users. In this protocol, the privacy of compliant users is guaranteed in the two server model. In addition, we propose a solutions for responsible release of banned funds due to non-compliance in both protocols.

An interesting question left for future work is to obtain compliant privacy mixers with de-anonymization requiring weaker trust assumptions.

References

1. M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer, 2016.
2. K. M. Alonso and J. Herrera-Joancomartí. Monero - privacy in the blockchain. *IACR Cryptol. ePrint Arch.*, 2018:535, 2017.
3. D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. M. R. Rasmussen, and A. Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 565–596, Cham, 2018. Springer International Publishing.
4. J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*, pages 486–504. Springer, 2014.
5. B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards privacy in a smart contract world. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*, pages 423–443. Springer, 2020.
6. J. Burleson, M. Korver, and D. Boneh. Privacy-protecting regulatory solutions using zero-knowledge proofs. `https://api.a16zcrypto.com/wp-content/uploads/2022/11/ZKPs-and-Regulatory-Compliant-Privacy.pdf`, 2022.
7. Chainalysis. Chainalysis oracle for sanctions screening. `https://go.chainalysis.com/chainalysis-oracle-docs.html`, 2023.
8. P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (S&P)*, pages 910–927. IEEE, 2020.
9. J. Devevey, B. Libert, K. Nguyen, T. Peters, and M. Yung. Non-interactive cca2-secure threshold cryptosystems: achieving adaptive security in the standard model without pairings. In *IACR International Conference on Public-Key Cryptography*, pages 659–690. Springer, 2021.
10. M. Dotan, S. Tochner, A. Zohar, and Y. Gilad. Twilight: A differentially private payment channel network. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 555–570, 2022.

11. Ethereum Foundation. Solidity programming language. `https://docs.soliditylang.org/en/latest/`.
12. A. Gabizon and Z. J. Williamson. fflonk: a fast-fourier inspired verifier efficient version of plonk. *Cryptology ePrint Archive*, 2021.
13. A. Gabizon, Z. J. Williamson, and O. Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.
14. Ganache. `https://www.trufflesuite.com/docs/ganache/overview`.
15. S. Goldwasser and S. Park. Public accountability vs. secret laws: Can they coexist? *Cryptology ePrint Archive*, 2018.
16. J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.
17. E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and distributed system security symposium*, 2017.
18. D. Hopwood, S. Bowe, T. Hornby, N. Wilcox, et al. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 4(220):32, 2016.
19. Iden3. edersen hash. `https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/pedersen-hash/pedersen.html`, 2019.
20. iden3. Circomlib/circuits. `https://github.com/iden3/circomlib/tree/master/circuits`, 2022.
21. iden3. Javascript and pure web assembly implementation of zksnark and plonk schemes. `https://github.com/iden3/snarkjs`, 2022.
22. J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC press, 2020.
23. Labrys. Mev watch. `https://web.archive.org/web/20230428094150/https://www.mevwatch.info/`, 2023.
24. D. V. Le and A. Gervais. Amr: Autonomous coin mixer with privacy preserving reward distribution. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 142–155, 2021.
25. G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei. Silentwhispers: Enforcing security and privacy in decentralized credit networks. In *NDSS*, 2017.
26. S. Meiklejohn and R. Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(2):105–121, 2018.
27. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.
28. A. Pertsev, R. Semenov, and R. Storm. Tornado cash privacy solution version 1.4. `https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf`, 2019.
29. A. Rondelet and M. Zajac. Zeth: On integrating zerocash on ethereum. *arXiv preprint arXiv:1904.00905*, 2019.
30. S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *arXiv preprint arXiv:1709.05748*, 2017.
31. E. B. 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, 2014.

32. Tornado Cash. Tornado cash privacy solution. `https://github.com/tornadocash/tornado-core`, 2019.
33. Tornado Cash. Tornado-cli. `https://github.com/tornadocash/tornado-cli`, 2019.
34. TRM Labs. North korea's lazarus group moves funds through tornado cash. `https://www.trmlabs.com/post/north-koreas-lazarus-group-moves-funds-through-tornado-cash`, 2022.
35. U.S. Department Of Treasury. Specially designated nationals and blocked persons list (sdn) human readable lists. `https://home.treasury.gov/policy-issues/financial-sanctions/specially-designated-nationals-and-blocked-persons-list-sdn-human-readable-lists`, 2022.
36. U.S. Department Of Treasury. U.s. treasury sanctions notorious virtual currency mixer tornado cash. `https://home.treasury.gov/news/press-releases/jy0916`, 2022.
37. L. Valenta and B. Rowan. Blindcoin: Blinded, accountable mixes for bitcoin. In *Financial Cryptography and Data Security: FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, pages 112–126. Springer, 2015.
38. N. Van Saberhagen. Cryptonote v 2.0. 2013.
39. Z. Wang, S. Chaliasos, K. Qin, L. Zhou, L. Gao, P. Berrang, B. Livshits, and A. Gervais. On how zero-knowledge proof blockchain mixers improve, and worsen user privacy. Cryptology ePrint Archive, Paper 2023/341, 2023. `https://eprint.iacr.org/2023/341`.
40. G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
41. 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.

A Formal Definitions

Our solution is compliant, private, sound, and correct in the following sense. *Correctness* is in the sense that any deposited funds can be withdrawn (once) as long as the matching deposit transaction is compliant at the time of the withdrawal, i.e., the withdrawn funds were not deposited from an address that is banned in $\mathcal{F}_{ban}^{Q}$. *Soundness* is in the sense that no user can withdraw more than it deposited. *Privacy* is in the sense that a withdrawal cannot be linked to any non withdrawn deposit. *Compliance* is in the sense that funds belonging to deposit transactions associated with an address in $\mathcal{F}_{ban}^{Q}$ cannot be withdrawn.

To formally state these properties we first set some notations. We denote the view of user Usr in an execution of the $\mathsf{deposit}$ and $\mathsf{withdraw}$ protocols in Figure 5, by

$$(r, \mathsf{dtxn}, \mathsf{address}_A) \leftarrow \mathsf{view}^{\mathsf{deposit}}_{\mathsf{Usr}}(\lambda) \text{ and}$$
$$(\mathsf{wtxn}, \mathsf{address}_B) \leftarrow \mathsf{view}^{\mathsf{withdraw}}_{\mathsf{Usr}}(r, \lambda)$$

respectively, where the view consists of the party's randomness, the generated (deposit/withdrawal) transaction, and the address associated with the transaction. We note that $\mathsf{view}^{\mathsf{withdraw}}_{\mathsf{Usr}}(r, \lambda)$ is defined wrt the first execution of $\mathsf{withdraw}$ on input r. We denote the output of Srv by

$$\mathsf{out}^{\mathsf{deposit}}_{\mathsf{Srv}}(\mathsf{dtxn}, \lambda) = b \text{ and}$$
$$\mathsf{out}^{\mathsf{withdraw}}_{\mathsf{Srv}}(\mathsf{wtxn}, \lambda) = b$$

where the bit b is the output of Srv. We call a transaction valid if $b = 1$. We remark that $\mathcal{F}^{Q}_{ban}$ and $\mathcal{F}_{bb}$ are accessible to any party in the system, and the transactions as well as the output of Srv may depend on their content.

Definition 2 (Correctness). *A mixer protocol $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ is* correct *if for every $\lambda \in \mathbb{N}$ and*

$$(r, \mathsf{dtxn}, \mathit{address}_A) \leftarrow \mathsf{view}^{\mathsf{deposit}}_{\mathit{Usr}}(\lambda) \text{ and}$$
$$(\mathsf{wtxn}, \mathit{address}_B) \leftarrow \mathsf{view}^{\mathsf{withdraw}}_{\mathit{Usr}}(r, \lambda)$$

the following holds with probability $\geq 1 - \mathrm{negl}(\lambda)$:

- $\mathsf{out}^{\mathsf{deposit}}_{\mathit{Srv}}(\mathsf{dtxn}, \lambda) = 1$
- *if $\mathit{address}_A$ is not recorded in $\mathcal{F}^{Q}_{ban}$ prior to* wtxn *generation then* $\mathsf{out}^{\mathsf{withdraw}}_{\mathit{Srv}}(\mathsf{wtxn}, \lambda) = 1$

where the probability is over the randomness of Usr and Srv.

Intuitively, the soundness property needs to capture that any user cannot withdraw more funds than it deposited. We formalize this by requiring that for any user in any point of time, represented by *index* in $\mathcal{F}_{bb}$, the number of valid withdrawals made from addresses belonging to a user must not exceed the number of successful deposits made by the user.

Definition 3 (Soundness). *A mixer protocol $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ is* sound *if for every index $\in \mathbb{N}$, and any ppt user Usr, associated with* *address set* $\mathrm{S}_{\mathsf{Usr}}$*, the following holds with probability* $\geq 1 - \mathrm{negl}(\lambda)$ *over the randomness of* Usr *and* Srv*:*

$$\left|\{(i, \mathsf{dtxn}_i, (\mathsf{address}_i, \mathsf{address}_{\mathsf{Srv}}))\}_{i<index \text{ and } \mathsf{address}_i \in \mathrm{S}_{\mathsf{Usr}}}\right| \\ \geq \left|\{(i, \mathsf{wtxn}_i, (\mathsf{address}_i, \mathsf{address}_{\mathsf{Srv}}))\}_{i\leq index \text{ and } \mathsf{address}_i \in \mathrm{S}_{\mathsf{Usr}}}\right|$$

where the tuples are recorded in $\mathcal{F}_{bb}$*, and for every* i *it holds that:* dtxn_i *and* wtxn_i *are valid deposit and withdrawal transactions, respectively.*

Intuitively, the definition of privacy captures the idea that an adversary should not be able to, given two deposit transactions and a withdrawal transaction belonging to one of the deposits, distinguish which of the deposits the withdrawal belongs to. This should hold true even if the adversary gets to freely interact with system.

Privacy. We define privacy for a mixer protocol $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ using the following experiment between a challenger Chal and an adversary $\mathcal{A}$ with access to $\mathcal{F}^{Q}_{ban}$:

The privacy experiment $\mathsf{EXP}_{\mathcal{A},\Pi,\mathcal{F}^{Q}_{ban}}(\lambda)$**:**

1. The adversary $\mathcal{A}$ can send to Chal a deposit and withdrawal requests that are processed by Chal as follows:
   - *Honest deposit generation:* Upon receiving a (deposit) request from $\mathcal{A}$ it executes deposit and simulates $\mathcal{F}_{bb}$ using the stored values. Then Chal stores $(r, \mathsf{dtxn}, \mathsf{address}_A) \leftarrow \mathsf{view}^{\mathsf{deposit}}_{\mathsf{Usr}}(\lambda)$ and sends $(\mathsf{dtxn}, \mathsf{address}_A)$ to $\mathcal{A}$.
   - *Honest withdrawal generation:* Upon receiving a (withdrawal, dtxn) request from $\mathcal{A}$ it fetches r that is associated with dtxn (if no such exists return $\perp$ to $\mathcal{A}$), it executes withdraw on input r and simulates $\mathcal{F}_{bb}$ using the stored values. Then Chal stores $(\mathsf{wtxn}, \mathsf{address}_B) \leftarrow \mathsf{view}^{\mathsf{withdraw}}_{\mathsf{Usr}}(r, \lambda)$ and sends wtxn to $\mathcal{A}$.
   - *Adversarial deposit/withdrawal submission:* In addition, $\mathcal{A}$ can submit a deposit or withdrawal transaction of its choice to Chal that records it in the simulated $\mathcal{F}_{bb}$ with appropriate *index*.
2. $\mathcal{A}$ outputs a pair of deposit transactions $\mathsf{dtxn}_0, \mathsf{dtxn}_1$ that correspond to two deposit transactions previously generated by Chal and were not requested to be withdrawn in Item 1.
3. Chal chooses a random bit $b \in \{0, 1\}$ and fetches r_b that is associated with dtxn_b. Then it executes withdraw on input r_b and send

to $\mathcal{A}$ the generated withdrawal transaction i.e., $(\mathsf{wtxn}, \mathsf{address}_B) \leftarrow \mathsf{view}^{\mathsf{withdraw}}_{\text{Chal}}(\mathsf{dtxn}_b)$. We call wtxn the challenge transaction. $\mathcal{A}$ continues to have access to Chal, interacting as in Item 1.
4. The adversary $\mathcal{A}$ outputs a bit b'. The experiment's output is defined to be 1 if $b' = b$, and 0 otherwise.

Definition 4 (Privacy). *A protocol $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ is* private *if for all* ppt *adversaries $\mathcal{A}$, there exists a negligible function* $\mathrm{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$,*

$$\Pr[\mathsf{EXP}_{\mathcal{A},\Pi,\mathcal{F}^{Q}_{ban}}(\lambda) = 1] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

where the probability is taken over the random coins used by $\mathcal{A}$ and Chal.

Intuitively, a compliant protocol should not allow the flow of illicit funds through the mixer. Additionally, compliance is somewhat meaningless for non sound protocols, i.e., ones that release funds without an appropriate assurance of their deposit by the withdrawing entity. Therefore, we focus our attention on compliance for sound protocols, according to Definition 3, and define compliance as follows: Our definition considers an idealized world where compliance is enforced by an ideal compliant ledger that "magically" deletes deposits from non compliant addresses, as if they never happened. Our compliance definition requires the protocol to behave indistinguishably when executed in our standard (append-only) ledger and the idealized world. In particular, we require that any valid withdrawal transaction is also valid in the idealized world, where no deposits from non-compliant addresses reside in the mixer. Combined with soundness this guarantees that the protocol enables withdraw funds only for compliant deposits. Formally,

Definition 5 (Compliance). *Let $\mathcal{F}_{bb}$ and $\mathcal{F}^{Q}_{ban}$ be the functionalities from Fig. 3 and Fig. 4, respectively, and let $\Pi = (\mathsf{deposit}, \mathsf{withdraw})$ be a* sound *mixer protocol as defined in Definition 3, with all entities having access to $\mathcal{F}_{bb}$ and $\mathcal{F}^{Q}_{ban}$. We say that Π is* compliant *if the following holds:*

- *The mixer* Srv *is stateless (i.e., it does not maintain state between executions of* deposit *or* withdraw *and it only performs* Read *requests to $\mathcal{F}_{bb}$).*
- *The user* Usr *only performs* Upload *requests to $\mathcal{F}_{bb}$ in* deposit.
- *for every tuple $(index, \mathsf{wtxn}, (\mathsf{address}_B, \mathsf{address}_{\mathsf{Srv}}))$ in $\mathcal{F}_{bb}$, it holds that:*

wtxn *is a valid withdrawal transaction if and only if*

$$\Pr[\mathsf{out}_{\mathit{Srv}}^{\mathsf{withdraw}(\mathcal{F}_{bb}^{*})}(\mathsf{wtxn}, \lambda) = 1] = 1$$

where $\mathsf{withdraw}(\mathcal{F}_{bb}^{*})$ *is the* $\mathsf{withdraw}$ *protocol of Π, where calls to $\mathcal{F}_{ban}^{Q}$ are ignored and $\mathcal{F}_{bb}$ is replaced with the following functionality $\mathcal{F}_{bb}^{*}$:*

- *Write and Read requests are treated as in $\mathcal{F}_{bb}$.*
- *every request $(\mathit{Ban}, \mathit{address}_A, \mathit{data})$ from address Q to $\mathcal{F}_{ban}^{Q}$ is forwarded to $\mathcal{F}_{bb}$ and treated by overwriting every tuple*

$$(index, \text{msg}, (\mathit{address}_A, \mathit{address}_{\mathit{Srv}}))$$

in $\mathcal{F}_{bb}$ to $(index, 0, (\perp, \mathit{address}_{\mathit{Srv}}))$.[5]

We note that though Srv in our protocol in Fig. 5 is not stateless, it can be equivalently defined as stateless that does not store the entire tree $\mathcal{T}$, but rather recreates it with each withdrawal call by reading the bulletin-board $\mathcal{F}_{bb}$. We avoid this in order to increase efficiency. Therefore compliance of our non-stateless protocol follows.

[5] A similar treatment could suggest removing the records from $\mathcal{F}_{bb}$ instead of overwriting, however for the ease of presentation we define it as above.

Common parameters: A security parameter λ, a function H sampled uniformly at random from a collision-resistant hash function family $\mathcal{H}$, and a ZK-SNARK scheme $\Phi = (\mathsf{ZK.Setup}, \mathsf{ZK.Prove}, \mathsf{ZK.Ver}, \mathsf{ZK.Sim})$ for relation R as defined in Section 3.1 and $R_{consist}$ as defined above. A CPA-secure PKE scheme $\mathcal{E} = (\mathsf{E.Gen}, \mathsf{E.Enc}, \mathsf{E.Dec})$.

Parties and addresses: A mixer Srv with public address $\mathsf{address}_{\mathsf{Srv}}$ and a user Usr with some addresses $\mathsf{address}_{\mathsf{Usr}_1}$ and $\mathsf{address}_{\mathsf{Usr}_2}$.

Storage: Srv locally stores a full binary Merkle tree $\mathcal{T}$ on n leaves, all initialized to zero, the location of the next available leaf $next = 0$, and, initially empty, nullifier set $\mathrm{S}_\emptyset$. Each leaf in $\mathcal{T}$ is associated with an address, initially set to $\perp$.

Trusted setup: Bulletin-board functionality $\mathcal{F}_{bb}$ and a Banned-addresses Functionality $\mathcal{F}_{ban}^{Q}$, containing records $(\mathsf{address}_\ell, \mathsf{data}_\ell)$, where $\mathsf{data}_\ell = \mathsf{E.Dec}_{sk_Q}(e)$ for e appearing in a deposit from $\mathsf{address}_\ell$. A common reference string σ produced by running $\mathsf{ZK.Setup}$. A key pair (sk_Q, pk_Q) produced by running $\mathsf{E.Gen}$, where $\mathcal{F}_{ban}^{Q}$ is parameterized on pk_Q, and sk_Q is securely stored by the predefined entity with address Q.

Deposit: $\mathsf{deposit}$ is executed by user Usr from some address $\mathsf{address}_{\mathsf{Usr}_1}$ and mixer Srv, as follows:

1. **User:** samples uniformly at random $k, r, r_e \leftarrow \{0,1\}^{t(\lambda)}$ for some polynomial $t(\cdot)$ and performs the following:
   (a) computes $C = H(k||r)$ and $e = \mathsf{E.Enc}_{pk_Q}(H(k), r_e)$.
   (b) $\pi_{\text{in}} \leftarrow \mathsf{ZK.Prove}(\sigma, (C, pk_Q, e), (k, r, r_e))$.
   (c) Sends $(\mathsf{Write}, ((C, e, \pi_{\text{in}}), \mathsf{address}_{\mathsf{Srv}})$ to $\mathcal{F}_{bb}$.
2. **Mixer:** Upon receiving $(index, (C, e, \pi_{\text{in}}), \mathsf{address}_{\mathsf{Usr}_1})$ from $\mathcal{F}_{bb}$, perform the following steps:
   (a) Compute $b \leftarrow \mathsf{ZK.Ver}(\sigma, (C, pk_Q, e), \pi_{\text{in}})$ and if $b = 0$ output 0.
   (b) Invoke the subroutine in Fig. 6 on $\mathsf{Update\_Tree}(\mathcal{T}, next, C, \mathsf{address}_{\mathsf{Usr}_1})$ to update $\mathcal{T}$, set $next = next + 1$, and output 1 if successful. We call such a deposit transaction valid.

Withdrawal: $\mathsf{withdraw}$ is executed by user Usr from some address $\mathsf{address}_{\mathsf{Usr}_2}$ and mixer Srv, as follows:

1. **User:** On input (k, r), to withdraw a deposit transaction $C = H(k||r)$ proceed as following:
   (a) Send Read request to $\mathcal{F}_{bb}$ and denote by $\mathrm{S}_{leaves} = \{(index_i, C_i, \mathsf{address}_i)\}_{i\in[k]}$ the subset of the returned records from $\mathcal{F}_{bb}$ where C_i is a valid deposit transaction, and $\mathsf{address}_i$ is the address associated with it.
   (b) Construct a Merkle tree $\mathcal{T}$ with $(C_1, \ldots, C_k)$ being the leaves and let $R_{\mathcal{T}}$ be the root of $\mathcal{T}$.
   (c) Compute the authentication path $O(\mathcal{T}, \ell)$, where ℓ is the leaf index of C in the computed $\mathcal{T}$ (if no leaf C, abort).
   (d) Compute $h = H(k)$ and $\pi_{\text{out}} \leftarrow \mathsf{ZK.Prove}(\sigma, (R_{\mathcal{T}}, h), (k, r, \ell, O(\mathcal{T}, \ell)))$.
   (e) Send $(\mathsf{Write}, (h, \pi_{\text{out}}), \mathsf{address}_{\mathsf{Srv}})$ to $\mathcal{F}_{bb}$. We call (h, π) a *withdrawal transaction* and h its *nullifier*.
2. **Mixer:** Upon receiving $(index, (h, \pi_{\text{out}}), \mathsf{address}_{\mathsf{Usr}_2})$ from $\mathcal{F}_{bb}$, perform the following steps:
   (a) Send Read request to $\mathcal{F}_{ban}^{Q}$ and denote by $\mathrm{S}_{ban} = \{(\mathsf{address}_\ell, \mathsf{data}_\ell)\}_{\ell\in[m]}$ the received banned-addresses list.
   (b) Add nullifiers associated with deposits from banned addresses: For each leaf ℓ in $\mathcal{T}$ and an address $\mathsf{address}_\ell$ associated with it, check if $\mathsf{address}_\ell \in \mathrm{S}_{ban}$, and if so $\mathrm{S}_\emptyset = \mathrm{S}_\emptyset \cup \mathsf{data}_\ell$.
   (c) Verify that $h \notin \mathrm{S}_\emptyset$ (output 0 otherwise).
   (d) Output $b \leftarrow \mathsf{ZK.Ver}(\sigma, (R_{\mathcal{T}}, h), \pi_{\text{out}})$ and if $b = 1$ set $\mathrm{S}_\emptyset = \mathrm{S}_\emptyset \cup h$.

Fig. 10: Compliant privacy mixer protocol Ψ, with de-anonymization of non-compliant users.