Pay Less for Your Privacy: Towards Cost-Effective On-Chain Mixers

Zhipeng Wang ✉
Imperial College London, UK

Marko Cirkovic ✉
University of Bern, Switzerland

Duc V. Le[1] ✉
Visa Research, USA

William Knottenbelt ✉
Imperial College London, UK

Christian Cachin ✉
University of Bern, Switzerland

Abstract

On-chain mixers, such as Tornado Cash (TC), have become a popular privacy solution for many non-privacy-preserving blockchain users. These mixers enable users to deposit a fixed amount of coins and withdraw them to another address, while effectively reducing the linkability between these addresses and securely obscuring their transaction history. However, the high cost of interacting with existing on-chain mixer smart contracts prohibits standard users from using the mixer, mainly due to the use of computationally expensive cryptographic primitives. For instance, the deposit cost of TC on Ethereum is approximately $1.1m$ gas (i.e., 66 `USD` in June 2023), which is 53× higher than issuing a base transfer transaction.

In this work, we introduce the Merkle Pyramid Builder approach, to incrementally build the Merkle tree in an on-chain mixer and update the tree per batch of deposits, which can therefore decrease the overall cost of using the mixer. Our evaluation results highlight the effectiveness of this approach, showcasing a significant reduction of up to 7× in the amortized cost of depositing compared to state-of-the-art on-chain mixers. Importantly, these improvements are achieved without compromising users' privacy. Furthermore, we propose the utilization of verifiable computations to shift the responsibility of Merkle tree updates from on-chain smart contracts to off-chain clients, which can further reduce deposit costs. Additionally, our analysis demonstrates that our designs ensure fairness by distributing Merkle tree update costs among clients over time.

2012 ACM Subject Classification Security and privacy → Pseudonymity, anonymity and untraceability

Keywords and phrases Privacy, Blockchain, Mixers, Merkle Tree

Funding DVL has been supported by a grant from Protocol Labs to the University of Bern.

Acknowledgements The authors thank anonymous reviewers for helpful feedback.

1 Introduction

Permissionless blockchains, such as Bitcoin [30] and Ethereum [41], provide pseudonymity rather than complete anonymity. Each transaction is openly recorded in plaintext on the public ledger, revealing details such as the transaction amount, timestamp, and the addresses of the sender and recipient. While this information alone may not directly expose the

[1] The main part of the work was conducted while the author was at the University of Bern.

individuals involved, it poses a risk as malicious actors can exploit it to analyze and link addresses, potentially compromising the anonymity of users [16, 44, 38, 39].

To enhance the privacy on non-privacy-preserving blockchains, on-chain mixers [4, 6, 1, 24] have been proposed. On-chain mixers are Decentralized Applications running on a blockchain with smart contracts, in which users deposit a *fixed* amount of coins into a pool, and subsequently withdraw those coins to a different address. When being used properly, on-chain mixers enable *unlinkability* between the deposit and the withdrawal addresses. The most active on-chain mixer, Tornado Cash (TC), has accumulated more than $51k$ unique deposit addresses for its largest pool [39]. The set of deposit addresses is similar to k-anonymity, which allows a withdrawal to be hidden among a set of k other deposits. The larger anonymity set size is accumulated, the harder it can link a user's withdrawal address to the corresponding deposit address.

However, achieving unlinkability through on-chain mixers comes at the cost of expensive deposit operations, making it difficult for regular users to use them. In particular, on-chain mixers use Merkle tree with Snark-friendly hash functions such as MiMC [7] and Poseidon [19] to record deposit and withdrawal history. When a deposit happens, the new leaf is appended sequentially to the Merkle tree. This update operation is the most expensive operation for on-chain mixers. In particular, depositing into TC costs approximately $1.1m$ gas, equating to roughly 66 USD as of June 2023. Such high costs can potentially discourage users from utilizing the mixer, thereby impeding the growth rate of the anonymity set size. Consequently, it becomes crucial to address these expensive deposit costs to ensure wider adoption and encourage the continued expansion of the anonymity set size.

In this paper, we propose two approaches to reduce the overall deposit costs in on-chain mixers. Firstly, we introduce the Merkle Pyramid Builder approach, in which we batch the deposits together and renew the Merkle tree per batch. This approach provides a promising solution to the scalability and efficiency challenges of on-chain mixers. Secondly, we propose the utilization of off-chain verifiable computation to further minimize deposit costs. This approach empowers clients to perform Merkle tree updating computations locally and provide cryptographic proofs to validate the accuracy of these computations. By shifting the responsibility of updating the Merkle tree from on-chain smart contracts to off-chain clients, we can significantly reduce the associated costs, enhancing the overall system efficiency.

Our contributions can be summarized as follows.

1. Merkle Pyramid Builder. We propose the Merkle Pyramid Builder (MPB) approach, which batches the deposits in a queue and reduces the Merkle tree update times in an on-chain mixer. This approach decreases the average number of times to execute the expensive Merkle tree with smart contracts per deposit. We locally implement the improved mixer with MPB construction, and our evaluation results demonstrate its remarkable ability to reduce deposit costs by 7× compared to the widely adopted TC mixer.

2. Off-Chain Deposit Proof Generation. We employ Verifiable Computation (VC) techniques to further minimize the cost of updating the Merkle tree. This is accomplished by offloading computations to the off-chain environment, effectively eliminating the computational requirements imposed on the smart contract. Our empirical evaluation results indicate that this approach further reduces the cost of a single deposit update.

3. Formal On-Chain Mixer Analysis Framework. We provide a formal framework to analyze the on-chain mixers by considering the properties of *correctness*, *privacy*, *availability*, *efficiency*, and *fairness*. We prove that our deposit-cost-reduction approaches offer enhanced *efficiency* without deteriorating other properties. Particularly, our analysis shows that the improved mixers can also guarantee *fairness*, with which the clients' costs for interacting with the mixer can be amortized over time. Furthermore, our analysis framework is of independent interest and could be applied to analyze other mixer designs.

2 Preliminaries

2.1 On-chain Mixers

On non-privacy-preserving blockchains, transactions are recorded in plaintext on the public ledger. On-chain mixers, inspired by Zerocash [33], are one of the most widely-used privacy solutions for non-privacy-preserving blockchains. On-chain mixers are running on top of blockchains with smart contracts, e.g., Ethereum and Binance Smart Chain (BSC). Upon using a mixer, a user deposits a fixed denomination of coins into a pool and later withdraws these coins to another address [4, 5, 6, 1, 24]. When used properly, mixers can break the linkability between addresses, and thus enhance users' privacy. The largest on-chain mixer, TC, has accumulated over $3.54m$ ETH from more than $39k$ Ethereum addresses [39].

2.2 Cryptographic Primitives

Notation. We denote by 1^λ the security parameter and by $\mathsf{negl}(\lambda)$ a negligible function in λ. We express by $(\mathsf{pk}, \mathsf{sk})$ a pair of public and private keys. Moreover, we require that pk can always be efficiently and deterministically derived from sk, and denote $\text{EXTRACTPK}(\mathsf{sk}) = \mathsf{pk}$ to be the deterministic function to derive pk from sk. We denote $\mathbb{Z}_{\geq a}$ as the set of integers that are greater or equal a, $\{a, a+1, \dots\}$. We let PPT denote probabilistic polynomial time. We denote $st[a, b, c \dots]$ as an instance of the statement where $a, b, c \dots$ have fixed and public values. We use a shaded area i, j, k to denote the private inputs in the statement $st : \{(a, b, c; i, j, k) : f(a, b, c, x, y, z) = \text{TRUE}\}$.

Collision Resistant Hash Functions. A family H of hash functions is collision-resistant, iff for all PPT $\mathcal{A}$, given $h \overset{\$}{\leftarrow} H$, the probability that $\mathcal{A}$ finds x, x', such that $h(x) = h(x')$ is negligible. We refer to the cryptographic hash function h as a fixed function, $h : \{0,1\}^* \rightarrow \{0,1\}^\lambda$. For the formal definitions of the hash function family, we refer readers to [32].

zk-SNARK. A zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK) is a "succinct" non-interactive zero-knowledge proof (NIZK) for arithmetic circuit satisfiability. The construction of zk-SNARK is based on a field $\mathbb{F}$ and an arithmetic circuit C. An arithmetic circuit satisfiability problem of a circuit $C : \mathbb{F}^n \times \mathbb{F}^h \rightarrow \mathbb{F}^l$ is captured by the statement $st_C : \{(x, \mathsf{wit}) \in \mathbb{F}^n \times \mathbb{F}^h : C(x, \mathsf{wit}) = 0^l\}$, with the language $\mathcal{L}_C = \{x \in \mathbb{F}^n \mid \exists\ \mathsf{wit} \in \mathbb{F}^l\ s.t.\ C(x, \mathsf{wit}) = 0^l\}$.

▶ **Definition 1** (zk-SNARK). *zk-SNARK for arithmetic circuit satisfiability is a triple of efficient algorithms* (SETUP, PROVE, VERIFY)*:*

- $(\mathsf{ek}, \mathsf{vk}) \leftarrow \textsc{Setup}(1^\lambda, C)$ *takes as input the security parameter and the arithmetic circuit* C*, outputs an evaluation key* ek*, and a verification key* vk*.*
- $\pi \leftarrow \textsc{Prove}(\mathsf{ek}, x, \mathsf{wit})$ *takes as input the evaluation key* ek *and* $(x, \mathsf{wit}) \in st_C$*, outputs a proof* π *for the statement* $x \in \mathcal{L}_C$*.*
- $0/1 \leftarrow \textsc{Verify}(\mathsf{vk}, \pi, x)$ *takes as input the verification key* vk*, the proof* π*, the statement* x*, outputs* 1 *if* π *is valid proof for the statement* $x \in \mathcal{L}_C$*.*

Commitment Scheme. A commitment scheme allows an entity to commit to a value while keeping it hidden, with the option of later revealing the value. A commitment scheme contains two rounds: committing and revealing. During the committing round, a client commits to selected values while concealing them from others. The client can choose to reveal the committed value during the revealing round.

▶ **Definition 2** (Commitment Scheme). *A commitment scheme includes two algorithms:*

- $\mathsf{cm} \leftarrow \textsc{Commit}(m, r)$ *accepts a message m and a secret randomness r as inputs and returns the commitment string* cm.
- $0/1 \leftarrow \textsc{Verify}(m, r, \mathsf{cm})$ *accepts a message m, a commitment* cm *and a decommitment value r as inputs, and returns 1 if the commitment is opened correctly and 0 otherwise.*

A secure commitment scheme satisfies two requirements: *(i) Binding:* Except for a negligible probability, no adversary can efficiently create cm, (m_1, r_1) and (m_2, r_2) such that $\textsc{Verify}(m_1, r_1, \mathsf{cm}) = \textsc{Verify}(m_2, r_2, \mathsf{cm}) = 1$ and $m_1 \neq m_2$. *(ii) Hiding:* Except for a negligible probability, cm does not reveal any information about the committed data.

Authenticated Data Structure. An Authenticated Data Structure (ADS) is a data structure that not only stores information but also provides a cryptographic proof of the integrity and authenticity of its contents. It allows for efficient verification of data integrity without requiring the entire data structure to be transmitted or stored alongside the proof. In this work, we adopt the Merkle tree as an ADS for set membership proof.

▶ **Definition 3** (Merkle Tree). *A Merkle tree leverages a collision-resistant hash function h to construct the data structure. The four algorithms work as follows:*

- $\mathsf{root} \leftarrow \textsc{Init}(1^\lambda, X)$ *takes the security parameter and a list $X = (x_1, \ldots, x_n)$ as inputs, constructs a tree that stores $x_1, \ldots, x_n$ in the leaves, and finally outputs a root,* root.
- $\mathsf{path}_i \leftarrow \textsc{Prove}(i, x, X)$ *takes an element $x \in \{0,1\}^*, 1 \leq i \leq n$ and a list $X = (x_1, \ldots, x_n)$ as inputs, and outputs the proof* path_i*, which can prove that x is in X. The proof generation time is proportional to n, while the proof size grows logarithmically with n.*
- $0/1 \leftarrow \textsc{Verify}(i, x_i, \mathsf{root}, \mathsf{path}_i)$ *takes an element, $x_i \in \{0,1\}^*$, an index $1 \leq i \leq n, y \in \{0,1\}^\lambda$ and a proof π as inputs, and outputs 1 if π is correctly verified and 0 otherwise.*
- $y' \leftarrow \textsc{Update}(i, x, X)$ *takes an element $x \in \{0,1\}^*, 1 \leq i \leq n$ and X as inputs, and outputs $y' = \textsc{Init}(1^\lambda, X')$ where X' is X but $x_i \in X$ is replaced by x.*

A Merkle tree should satisfy *correctness* and *security.* For the formal definitions of these properties, we refer to the cryptography introduction book of Boneh and Shoup [13].

Cost for Appending to Merkle tree. Let T be a Merkle hash tree of size n and assume that all internal nodes are stored. If a single element is appended, the computational cost of updating a Merkle hash tree is $O(\log n)$ if the internal tree nodes are stored. This can be done by traversing the right-most path of the tree and modifying at most $O(\log n)$ internal nodes of the tree.

Inefficiency of Multiple Appending Operations in Existing On-Chain Mixers. We observe that despite the theoretically efficient append operation, the utilization of a SNARK-friendly hash function still leads to a $\log n$ cost of appending that amounts to around $1m$ gas. For k deposits, this operation is repeated k times, raising the cost to $O(k \cdot \log n)$.

However, in a conventional mixer, to assimilate within a sufficiently large anonymity set, users often wait several days before withdrawing funds [39, 24]. Consequently, this suggests that the Merkle tree update operation could be processed in a batch rather than individually each time. This adjustment could potentially decrease the cost from $O(k \log(n))$ to $O(k + \log(n))$. Further details on this improvement will be provided in Section 5.

Efficient Replace. The update algorithm described previously needs the entire set X to be able to recalculate the root. Nevertheless, it is feasible to update the root without knowing the entire set. Specifically, we can update the root in $O(\log(|X|))$ operations using only the information about the node membership that one wants to replace and the current root. This update will allow an efficient on-chain update of the Merkle tree.

- $\mathsf{root}'_{dep}/\bot \leftarrow \textsc{Replace}(i, x, \mathsf{root}_{dep}, \mathsf{path}_i, x')$: takes as input the index i, the old element x and its membership proof path_i, and the new element, x' that we want to put in the i-th position. The algorithm verifies the membership of both x in the old root_{dep} using path_i, abort otherwise. Once the verification returns 1, it recomputes the root root'_{dep} using x' and path_i.

This efficient update is needed for the construction using verifiable computation.

3 On-chain Mixer System

This section presents the on-chain mixer system's components and the algorithms for the setup phase, the client, and the smart contract.

3.1 System Components

The system consists of two components: the *client* and the *smart contract*. A client controls blockchain addresses to interact with the smart contract, which governs a *pool* of assets. A client can either deposit/withdraw coins into/from the pool. The smart contract manages both deposit and withdrawal actions. The contract keeps track of various data structures and parameters to verify the validity of transactions that are sent to the contract.

3.2 Contract Setup

In the setup phase, all public parameters and the mixer smart contract are generated. The contract will be initialized with different data structures to avoid double withdrawal.

Furthermore, the deposit amount is specified as a fixed deposit amount of coins, amt. The smart contract is set up with two empty lists: *(i)* $\mathsf{DepositList}$, which includes all commitments cm contained in depositing transactions; *(ii)* $\mathsf{NullifierList}$, which contains all unique identifiers (i.e., sn) appeared in withdrawal transactions. We refer to pp^h as the state of the contract at the block height h. The state includes all data structures of the contract, which were initialized in the setup phase. This state is implicitly provided to all client and contract algorithms. Finally, the smart contract is deployed on-chain in this phase.

3.3 Client Algorithm

A client can interact with the smart contract using the following algorithms. Note that each transaction is signed by the client using the private key associated with the blockchain address which issues the transaction.

- $(\mathsf{wit}, \mathsf{tx}_{dep}) \leftarrow \textsc{CreateDepositTx}(\mathsf{sk}, \mathsf{amt})$ takes a private key sk and the fixed amount amt as inputs, and outputs a witness wit and a deposit transaction tx_{dep}.
- $\mathsf{tx}_{wdr} \leftarrow \textsc{CreateWithdrawTx}(\mathsf{sk}', \mathsf{wit})$ takes as input a private key sk' (which can be different from sk) and the witness wit, and outputs a withdrawal transaction tx_{wdr}.

3.4 Smart Contract Algorithm

The smart contract handles mixer deposits and withdrawals, with the following algorithms:

- $0/1 \leftarrow \textsc{AcceptDeposit}(\mathsf{tx}_{dep})$ takes as an input the deposit transaction tx_{dep}, and outputs 1 if the transaction was successful and 0 otherwise.
- $0/1 \leftarrow \textsc{IssueWithdraw}(\mathsf{tx}_{wdr})$ takes the withdrawal transaction tx_{wdr} as the input, and outputs 1 and transfers amt coins to the sender $\mathsf{tx}_{wdr}.\mathsf{sender}$ if the transaction was successful. Otherwise, the algorithm outputs 0.

3.5 System Goals

A secure on-chain mixer aims to satisfy the following properties.
Privacy. An on-chain mixer can break the linkability between user addresses. Consider an adversary with access to the entire history of all deposit and withdrawal transactions made to the mixer contract. Given a client who deposits into and withdraws from the mixer, the system ensures that the adversary cannot *(i)* link the deposit and withdrawal transactions issued by the client, and *(ii)* link the deposit and withdrawal addresses used by the client.
Correctness. A client cannot withdraw more coins from the contract than the client deposits. Moreover, a client cannot withdraw coins from the mixer prior to the deposit. The property prevents a client from stealing coins from the contract or other clients.
Availability. Clients should always be able to use the mixer. No entity can prevent clients from depositing or withdrawing coins.
Efficiency. The system should efficiently update clients' deposits and withdrawals, without causing expensive costs.
Fairness. Given a time interval, the costs of clients who deposit into (resp. withdraw from) the mixer during the interval remain approximately equal. This property allows for the amortization of costs over time, contributing to a balanced and equitable user experience.

4 Basic On-chain Mixer

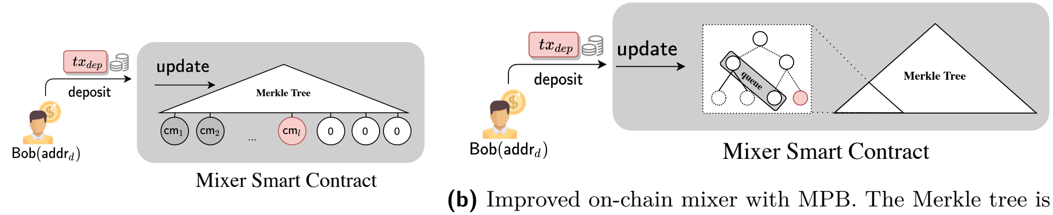
4.1 Cryptographic Building Blocks

In the following, we present the cryptographic building blocks to construct on-chain mixers.

Deposit Commitments. Let $(\textsc{Com}, \textsc{Verify})$ be a hiding and binding commitment scheme. To deposit into the contract, a client samples two randomnesses k_{dep}, r, and computes the commitment $\mathsf{cm} = \textsc{Com}(m, r)$ as a part of a deposit transaction. In practice, the commitment scheme can be realized using a secure hash function $H_p : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}$.

Merkle Tree for Deposits. In an on-chain mixer, the leaves of the Merkle tree are initialized with zero values. The mixer smart contract preserves the Merkle tree T_{dep} of all deposit commitments. When deposit transactions occur, the smart contract keeps track of the total number of deposit transactions and updates the tree using the $\textsc{AcceptDeposit}$ algorithm. We define the Merkle proof for commitment cm_i as path_i. In addition, we define the root of the Merkle tree at block h as $\mathsf{root}^{\mathsf{curr}}_{wdr}$. We also denote root_{dep}.blockheight as the height of the blockchain block at the moment when root_{dep} is updated. Figure 14 presents a visual illustration, and Definition 4 formally defines a deposit Merkle tree. Note that $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ is a collision-resistant hash function.

▶ **Definition 4** (Deposit Merkle Tree). *A deposit Merkle tree encompasses three essential algorithms, namely $T.\textsc{Init}()$, $T.\textsc{Prove}()$, and $T.\textsc{Verify}()$, as defined in Definition 3. Additionally, the tree can be efficiently updated in a batch using the following algorithm:*



(a) Basic on-chain mixer. The Merkle tree is updated for every successful deposit.

(b) Improved on-chain mixer with MPB. The Merkle tree is updated once the leaves in the subtree corresponding to the deposit queue are full.

Figure 1 Overview of basic and improved on-chain mixer systems.

- $\mathsf{root}_{new} \leftarrow T.\textsc{Update}(Q)$ *takes a list Q containing new hashes as input. The algorithm inserts the elements in Q into the existing Merkle tree, and thus the new root will be updated. The output is the new root,* root_{new}.

Withdrawal Proof. To withdraw coins from the smart contract, a client needs to satisfy:

1. The client has committed values for certain existing deposit commitments utilized to construct the tree root using zk-SNARK.
2. The nullifier in the withdrawal transaction has not been used to withdraw previously.
3. The private key used to issue the withdrawal transaction is known by the client.

In short, a client needs to provide a proof showing the following statement for a Merkle tree T_{dep} with the root root_{dep}:

$$\begin{aligned} st_{wdr} : \{(&\mathsf{pk}, \mathsf{sn}, \mathsf{root}_{dep}; \mathsf{sk}, k_{dep}, r, \mathsf{path}_i) : \mathsf{pk} = \textsc{extractPK}(\mathsf{sk}) \land \mathsf{sn} = H_p(k_{dep}, 0^\lambda) \land \\ &\mathsf{cm} = \textsc{Com}(k_{dep}, r) \land T.\textsc{Verify}(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}_i))\} \end{aligned} \tag{1}$$

where $\mathsf{pk}, \mathsf{sn}, \mathsf{root}_{dep}$ are public values and $\mathsf{sk}, k_{dep}, r, \mathsf{path}_i$ are private values[2].

4.2 Workflow of Basic On-chain Mixer

Fig. 1a shows a basic solution for an on-chain mixer system, which is adopted by existing on-chain mixers, e.g., TC [4], Typhoon Network (TN) [6], and Cyclone [1]. In a nutshell, the high-level workflow of a basic on-chain mixer is as follows:

1. For each deposit transaction, a client adopts an address to issue a transaction, which transfers a *fixed* amount of coins into the mixer smart contract and generates a deposit commitment. The contract adopts a Merkle tree to record the deposit commitments. Whenever a new deposit is made, the tree is updated, generating a new root. The roots are stored in a root list.
2. To withdraw, the client provides a withdrawal proof. The proof essentially shows that the client knows the secret to open a commitment in a deposit transaction and a membership proof by providing a path from the commitment to one root stored in the roots list (cf. Equation 1). The client will receive coins after the contract verifies the proof. The contract also records all withdrawal transactions' unique nullifiers.

[2] sk is not needed in the private values if the zk-SNARK has simulation extractability property [8].

4.3 System Goals of Basic On-chain Mixer

The basic mixers satisfy correctness, privacy, availability, and fairness when used properly:
Correctness: To issue a withdrawal, a client is required to present a proof associated with a unique nullifier. The mixer smart contract maintains a record of all unique nullifiers associated with withdrawals. Consequently, a client is restricted from issuing more withdrawals than the number of deposits they have previously conducted with the mixer.
Privacy: Considering the presence of n deposits within a mixer, a client's new deposit transaction becomes concealed amidst the n transactions sharing the same deposit amount. Furthermore, by employing separate deposit and withdrawal addresses, the probability of an adversary successfully linking the accurate deposit and withdrawal transactions is $\frac{1}{n}$.
Availability: As the mixer operates on a permissionless blockchain, which utilizes a global peer-to-peer network, adversaries are unable to impede clients from engaging with the mixer.
Fairness: The Merkle tree is updated for each deposit, resulting in equal gas fees for clients. When clients withdraw from the mixer, they incur costs for proof verification. These deposit and withdrawal costs are dependent on the gas price. Therefore, assuming the gas price does not fluctuate over a short timeframe, the mixer ensures fairness of costs.
Inefficiency: However, we contend that the existing basic on-chain mixer design lacks efficiency since clients are required to pay a high deposit cost for updating the entire Merkle tree. In typical basic mixers such as TC, whenever a deposit request is made, the new coin is sequentially inserted into the Merkle tree, necessitating an update to the entire tree. This update operation becomes even more costly due to the utilization of Snark-friendly hash functions. For instance, by analyzing the 156,466 deposit transactions in the TC 0.1, 1, 10, and 100 `ETH` pools from block 9,117,019 (December 16th, 2019) to 16,329,600 (January 3rd, 2023), we discovered that the average deposit cost for a TC `ETH` pool is approximately 1,111,030 gas, which is roughly 53 times higher than the Ethereum base fee of 21,000 gas.

5 Improving On-chain Mixers via Merkle Pyramid Builder

Existing on-chain mixers suffer from expensive deposit costs due to the frequent update of the Merkle tree for each deposit transaction. However, the one-deposit-one-update approach appears redundant since it is advisable to wait for other clients to deposit before initiating a withdrawal [39, 24]. To reduce the update time of the Merkle tree, we draw inspiration from the concept of Merkle tree mountain range [31, 34] (refer to Definition 15). Accordingly, we propose a novel method called MPB.

Deposit-Queuing. To optimize the cost of deposits, we introduce a deposit-queuing method that batches transactions, resulting in less frequent updates of the Merkle tree.

▶ **Definition 5** (Deposit-Queuing). *A deposit-queuing method consists of three algorithms:*

- $q_{empty} \leftarrow \textsc{CreateQueue}(l)$ *takes as input an integer l, which specifies the number of deposit transactions to be batched. It returns an empty queue q_{empty} with a size of l, designed to store the nodes in a subtree with a height of $\log_2 l$. A deposit queue is considered full when all the leaves in the corresponding subtree are occupied.*
- $q' \leftarrow \textsc{Enqueue}(q, \mathsf{cm})$ *takes as input a queue q containing internal nodes of the subtree that can be used as helpers for an efficient update and a new commitment cm. As shown in Fig. 2, this procedure resembles the storage of internal nodes in the Merkle Mountain Range [34] and accumulator construction outlined in [31]. The function produces a new output queue q' containing internal nodes that enable the subtree to perform efficient updates during subsequent deposits.*

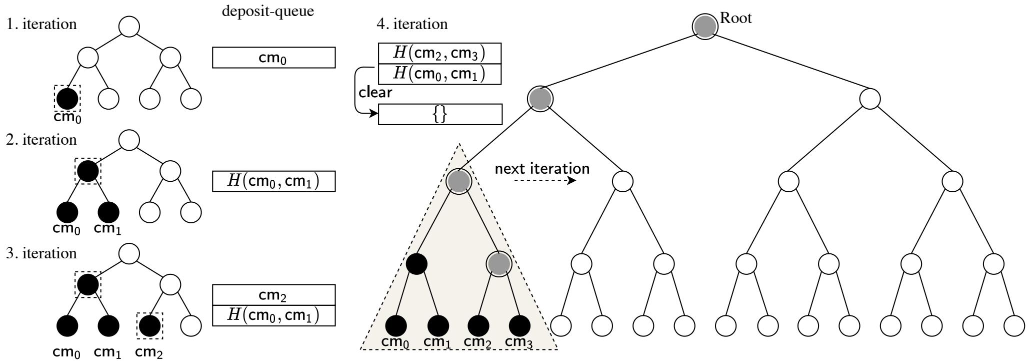


Figure 2 Graphical illustration of the Merkle Pyramid Builder approach with a deposit queue size of four. cm_i represents the deposit commitment. In the first iteration, a deposit is made and included in the deposit queue. For subsequent deposits, the client combines the new deposit with the previously stored deposit, generating a new value that replaces the previous deposit in the queue. The third deposit is appended to the queue. Upon the fourth deposit, all hashes, including the root, are computed using all the values in the deposit queue. Finally, the deposit queue is cleared, and the process restarts from the beginning for the next deposit.

- $q_{empty} \leftarrow \textsc{ClearQueue}(q)$ *takes as input a queue q. The algorithm returns an empty queue corresponding to a new subtree.*

Merkle Pyramid Builder Approach. As shown in Fig. 1b, the key concept is to avoid frequent updates of the Merkle tree by aggregating deposit transactions and performing collective updates. In this approach, a deposit queue of size l corresponds to a subtree with a height of $\log_2 l$, as illustrated in Fig. 2. Notably, every even deposit in the sequence incurs no additional computational cost. However, each odd deposit requires hashing all the hashes up to the tree until no values remain on the left side within the same subtree. Finally, every l-th deposit necessitates computing all the hashes up to the root.

5.1 Contract Setup

The contract setup phase is to generate all the cryptographic parameters used in the protocol. In the setup phase (cf. Fig. 3), the algorithm $\textsc{ContractSetUp}$ samples two secure hash functions H_p and H_{2p} from the collision-resistant hash families. The procedure further initializes amt as the fixed amount of coins that can be deposited into the mixer contract.

Setup Merkle Tree. We denote T as the deposit commitment Merkle tree with depth d. The algorithm $T.\textsc{Init}$ initializes T as described in Section 4.1. The algorithm also initiates $\mathsf{RootList}_{wdr,k}$ to be the list of k most recent roots of T, which helps address the concurrency issue for withdrawal transactions.

Setup zk-SNARK Parameters. We initialize a zk-SNARK instance Π with the algorithm $\Pi.\textsc{Setup}$ with C_{wdr} as input, which can output two keys $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep})$.

Setup Commitments and Nullifier Lists. The contract initializes two empty lists: *(i)* $\mathsf{DepositList}$, which contains all cm included in deposit transactions; *(ii)* $\mathsf{NullifierList}$, which contains all unique nullifiers sn committed in withdrawal transactions.

Setup Deposit Queue. The $\mathsf{DepositQueue}$ is initialized to track the number of deposit transactions in the queue and the number of transactions that have been hashed. This information determines whether the subsequent client needs to bear the cost of updating the Merkle tree. Note that the queue size $\log_2 l$ is the height of the subtree.

```
ContractSetUp($1^\lambda$)
Sample $H_p : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}$ and $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$
Choose $\mathsf{amt} \in \mathbb{Z}_{>0}$ to be a fixed deposit amount
Choose $d \in \mathbb{Z}_{>0}$, Let $X = \{x_1, \ldots, x_{2^d}\}$ where $x_i = 0$ for all $x_i \in X$
Initialize an empty tree $\mathsf{root}_{dep} = T.\textsc{Init}(1^\lambda, X)$,
Choose $k \in \mathbb{Z}_{>0}$, set $\mathsf{RootList}_{wdr,k}[i] = \mathsf{root}_{dep}$, for $1 \leq i \leq k$
Construct $C_{wdr}$ for statement described in Equation 1
Let $\Pi$ be the zk-SNARK instance. Run $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}) \leftarrow \Pi.\textsc{Setup}(1^\lambda, C_{wdr})$
Initialize: $\mathsf{DepositList} = \{\}, \mathsf{NullifierList} = \{\}, \mathsf{TotalFee} = 0$
Initialize: $\mathsf{DepositQueue} \leftarrow \textsc{CreateQueue}(l)$
Deploy smart contract with parameters :
$\mathsf{pp} = (\mathbb{F}, H_p, H_{2p}, \mathsf{amt}, \mathsf{fee}_d, T, \mathsf{index}, \mathsf{RootList}_{wdr,k}, \mathsf{DespositQueue},$
    $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}), \mathsf{DepositList}, \mathsf{NullifierList}, \mathsf{TotalFee})$
```

Figure 3 Pseudocode for the smart contract setup in a mixer with MPB.

Functionality of $\mathsf{RootList}_{wdr,k}$. Similar to existing on-chain mixers [4, 24], our mixer contract maintains a list of the k most recent roots, denoted as $\mathsf{RootList}_{wdr,k}$. This list serves as an "AllowList" [17] to maintain a list of authorized parties (i.e., roots), which can address the concurrency issue.

Consider a scenario where user Bob attempts to withdraw using the most recently updated root $root_{k-1}$ from the $\mathsf{RootList}_{wdr,k}$. If an attacker, Alice, manages to make k deposits prior to Bob's withdrawal, these deposits would prompt k updates to the Merkle tree, effectively removing $root_{k-1}$ from the refreshed $\mathsf{RootList}_{wdr,k}$ and invalidating it for withdrawal.

However, as demonstrated in [24], this attack's cost is at least $k \times (\mathsf{amt} + \mathsf{fee})$, where amt represents the number of coins supported by the mixer pool (e.g., 0.1, 1, 10, and 100 for the TC `ETH` pools), and fee denotes the deposit fee. Moreover, the cost is higher in our improved mixer with MPB, as it typically requires $k \times l$ deposits to trigger k Merkle tree updates.

Importantly, the $\mathsf{RootList}_{wdr,k}$ presence does not introduce any vulnerability to double withdrawals. The only downside is that users might experience a marginally smaller anonymity set than the actual one. For instance, a user might use the root $root_0$ (corresponding to $|\mathsf{CmpSet}^\mathsf{h}|$ deposits), but due to concurrency, the latest root available may be $root_{k-1}$ (corresponding to $|\mathsf{CmpSet}^\mathsf{h}| + k$ deposits). In reality, $k \ll |\mathsf{CmpSet}^\mathsf{h}|$ (e.g., in TC, k is set to 100^3, while there are more than $26,000$ deposits in each `ETH` pool [39]), rendering the difference in the anonymity set negligible.

5.2 Deposit Interaction

Fig. 4 shows the deposit process of a client in the mixer. This step allows the client to deposit coins into the mixer pool and obtain the witness for future withdrawal.

Client. A client deposits coins into the contract using the algorithm CreateDepositTx. The client first randomly selects two parameters k_{dep} and r, which are used to construct the commitment cm. The lengths of k_{dep} and r are defined and fixed during the initial setup.

[3] https://etherscan.io/address/0x910...dbf#code

Client(sk, amt)

```
CreateDepositTx(sk, amt) :
1 :  Sample $(k_{dep}, r) \xleftarrow{\$} \{0,1\}^{\lambda}$
2 :  Compute $\mathsf{cm} = H_p(k_{dep}, r)$
3 :  return $\mathsf{tx}_{dep} = (\mathsf{amt}, \mathsf{cm}, \mathsf{fee}_d)$,
         $\mathsf{wit} = (k_{dep}, r)$
```

$\xrightarrow{\mathsf{tx}_{dep}}$

Smart Contract

```
AcceptDeposit($\mathsf{tx}_{dep}$)
1 :  Parse $\mathsf{tx}_{dep} = (\mathsf{amt}', \mathsf{cm}, \mathsf{fee}'_d)$
2 :  Require: $\mathsf{amt}{=}\mathsf{amt}', \mathsf{fee}_d = \mathsf{fee}'_d, \mathsf{index} < 2^d$
3 :  Append cm to DepositList
4 :  Increment $\mathsf{index} = \mathsf{index} + 1$
5 :  $\mathsf{DepositQueue} = \textsc{Enqueue}(\mathsf{DepositQueue}, \mathsf{cm})$
6 :  $\mathsf{TotalFee} = \mathsf{TotalFee} + \mathsf{fee}_d$
     // The subtree is full
7 :  if DepositQueue is full:
8 :    Compute $\mathsf{root}_{new}$ from DespositQueue
9 :    ClearQueue(DespositQueue)
10 :   Append $\mathsf{root}_{new}$ to $\mathsf{RootList}_{wdr,k}$
       // Compensate the tree updater
11 :   Do $\mathtt{tx}_{dep}$.sender.transfer(TotalFee)
12 :   $\mathsf{TotalFee} = 0$
13 : else :
14 :   Enqueue(DespositQueue, cm)
15 : return 1
```

Figure 4 Deposit interactions between the client and the smart contract in a mixer with MPB. The computation of root_{new} (in line 8) can be done via the efficient replace function Def.2.2

The deposit transaction tx_{dep} consists of the commitment cm and the coins amt that the client wants to deposit. Moreover, the client needs to specify the deposit fee fee_d in the transaction. To issue the transaction tx_{dep}, the client must use their private key sk to sign it. In addition, a witness wit is issued, which will be used to withdraw the coins in the future.

Contract. Upon receiving a deposit transaction tx_{dep}, the smart contract verifies that *(i)* the amount of coins and fees are as requested, and *(ii)* there is available space in the Merkle tree. If both conditions are met, the commitment in tx_{dep} is added to the DepositList. If the deposit queue is not full, the deposit fee will be added to the current total fee TotalFee. Otherwise, the total fee will be transferred to the last client and be reset as 0. With the algorithm Enqueue, the list DespositQueue will be updated. Moreover, if this deposit queue is full, the Merkle tree will be updated using the algorithm T.Update, and the smart contract will clear the deposit queue and finally add the new root to the list $\mathsf{RootList}_{wdr,k}$.

Note that the last client in a deposit queue will pay the gas cost for updating the Merkle tree. To guarantee fairness, the previous clients in the queue need to transfer additional deposit fees fee_d to the smart contract, which will be accumulated and serve as the reimbursement for the last client. A more comprehensive analysis of the deposit fee design will be presented in Section 6.4.

5.3 Withdrawal Interaction

Fig. 5 shows the mixer withdrawal interaction. This step allows the client to use the secret witness to generate a proof which is used to withdraw the corresponding deposited coins.

Client. A client needs to create a withdrawal proof π_{wdr} with the secret witness wit, and the private key sk', to withdraw amt to the public key pk'. The proof π_{wdr} should be able to

Client(sk)

```
CreateWithdrawTx$(\mathsf{sk}, \mathsf{wit})$ :
    Parse $\mathsf{wit} = (k_{dep}, r)$
    Obtain $\mathsf{pp}^h$ from the contract
    Compute $\mathsf{sn}_{wdr} = H_p(k_{dep}, 0^\lambda)$
    Compute $\mathsf{cm} = H_p(k_{dep}, r)$
    Get index $i$ of $\mathsf{cm}$ from $\mathsf{DepositList}^h$
    Choose $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
    Compute $\mathsf{path}^h_{dep,i}$ s.t.:
    $T.\textsc{Verify}(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}^h_{dep,i}) = 1$
    Form $\mathsf{wit}_{dep} = (\mathsf{sk}, k_{dep}, r, \mathsf{path}^h_{dep,i})$
    $\pi_{wdr} \leftarrow \Pi.\textsc{Prove}(\mathsf{ek}_{dep}, \mathsf{wit}_{dep},$
        $st[\mathsf{pk}, \mathsf{sn}_{wdr}, \mathsf{root}_{dep}])$
    return $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$
```

$\xrightarrow{\mathsf{tx}_{wdr}}$

Smart Contract

```
IssueWithdraw$(\mathsf{tx}_{wdr})$ :
    Parse: $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$
    Require:
        $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
        $\mathsf{sn}_{wdr} \notin \mathsf{NullifierList}$
        $\Pi.\textsc{Verify}(\mathsf{vk}_{dep}, \pi_{wdr}, st[\mathtt{tx}_{wdr}.\mathsf{sender},$
        $\mathsf{sn}_{wdr}, \mathsf{sn}_{wdr}, \mathsf{root}_{dep}]) = 1$
    Append $\mathsf{sn}_{wdr}$ to NullifierList
    // Send the original deposit back
    Do $\mathtt{tx}_{wdr}$.sender.transfer(amt)
    return 1
```

Figure 5 Withdrawal interactions between the client and the smart contract in a mixer with MPB. The state of the contract at block height h is denoted by pp^h. The withdrawal transaction tx_{wdr} contains the proof π_{wdr} that proves the client's knowledge of $\mathsf{cm} = H_p(k_{dep}, r)$ which is a valid member of the Merkle tree with the root root_{wdr}.

prove the three conditions mentioned in Section 4.1. The client then issues the withdrawal transaction tx_{wdr}, which consists of the nullifier sn_{wdr}, the root root_{dep}, and the proof π_{wdr}.

Contract. When receiving a withdrawal transaction $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$, the contract checks the proof, π_{wdr}, and confirms that the nullifier sn_{wdr} is not in the NullifierList. The contract then appends sn_{wdr} to NullifierList to avoid future double withdrawals. Finally, the smart contract transfers amt coins to the withdrawal address specified by the client.

5.4 Further Improvement with Verifiable Computation Techniques

In this section, we employ verifiable computation techniques [18, 23] to enhance the efficiency of deposit costs. In an on-chain mixer utilizing VC, the key idea is to conduct the computation off-chain. In this approach, the client evaluates the Merkle tree root and transmits both the result and a proof of correct computation to the smart contract. The contract subsequently verifies the validity of the proof to ensure the accuracy of the computation.

Verifiable Computation Scheme. In a VC scheme (cf. Definition 6), the verifier selects a function and an input to send to the prover. The prover evaluates the function on the input and returns the result, along with proof attesting to the result's validity. The verifier then verifies that the output provided by the prover indeed corresponds to the result of the function evaluated on the given input. The objective is to achieve efficient verification, significantly faster than the actual computation of the function itself.

▶ **Definition 6** (Verifiable Computation Scheme). *Let f be a function, expressed as an arithmetic circuit over a finite field $\mathbb{F}$, and let λ be a security parameter.*

- $(\mathsf{ek}, \mathsf{vk}) \leftarrow \textsc{VcInit}(1^\lambda, f)$ *takes a security parameter and an arithmetic circuit as input and generates two public keys: an evaluation key* ek *and a verification key* vk.

```
$\textsc{ContractSetUp}(1^\lambda)$

Sample $H_p : \{0,1\}^* \to \mathbb{F}$ and $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$
Choose $\mathsf{amt} \in \mathbb{Z}_{>0}$ to be a fixed deposit amount
Choose $d \in \mathbb{Z}_{>0}$, Let $X = \{x_1, \ldots, x_{2^d}\}$ where $x_i = 0^\lambda$ for all $x_i \in X$
Initialize an empty tree $\mathsf{root}_{dep} = T.\textsc{Init}(1^\lambda, X)$,
Choose $k \in \mathbb{Z}_{>0}$, set $\mathsf{RootList}_{wdr,k}[i] = \mathsf{root}_{dep}$, for $1 \le i \le k$
Construct $C_{wdr}$ for statement described in Equation 1
Construct $C_{dep}$ for statement described in Equation 2
Let $\Pi_{dep}$ be the Verifiable Computation instance.
    – Run $(\mathsf{ek}^{vc}_{dep}, \mathsf{vk}^{vc}_{dep}) \leftarrow \Pi_{dep}.\textsc{VcInit}(1^\lambda, C_{dep})$
Let $\Pi_{wdr}$ be the zk-SNARK instance.
    – Run $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}) \leftarrow \Pi_{wdr}.\textsc{Setup}(1^\lambda, C_{wdr})$
Initialize: $\mathsf{DepositList} = \{\}, \mathsf{NullifierList} = \{\}, \mathsf{TotalFee}$
Initialize: $\mathsf{DepositQueue} \leftarrow \textsc{CreateQueue}(l)$
Deploy smart contract with parameters :
$\mathsf{pp} = (\mathbb{F}, H_p, H_{2p}, \mathsf{amt}, \mathsf{fee}_d, T, \mathsf{index}, \mathsf{RootList}_{wdr,k}, \mathsf{DespositQueue},$
        $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}), (\mathsf{ek}^{vc}_{dep}, \mathsf{vk}^{vc}_{dep}), \mathsf{DepositList}, \mathsf{NullifierList}, \mathsf{TotalFee})$
```

Figure 6 Pseudocode for the smart contract setup in a mixer with verifiable computations.

- $(y, \pi) \leftarrow \textsc{VcProve}(\mathsf{ek}, x)$ *takes as input an element* x *and the evaluation key* ek *and computes* $y = f(x)$ *and a proof* π *that* y *has been correctly computed.*
- $0/1 \leftarrow \textsc{VcVerify}(\mathsf{vk}, x, y, \pi)$ *takes the verification key* vk*, the input/output* (x, y) *of the computation* f *and the proof* π *and outputs* 1 *if* $y = f(x)$ *and* 0 *otherwise.*

Any SNARK instance (e.g., Groth16 [20]) can be used to construct a VC scheme.

Deposit Proof. To deposit coins to the smart contract, a client needs to give a proof showing the following relation for a Merkle tree T with a root root_{dep}:

$$st_{dep} : \{\mathsf{pk}, \mathsf{cm}, \mathsf{root}^{old}_{dep}, \mathsf{root}^{new}_{dep}, \mathsf{path}_i : \mathsf{root}^{new}_{dep} = T.\textsc{Replace}(\mathsf{index}, 0, \mathsf{root}^{old}_{dep}, \mathsf{path}_i, \mathsf{cm})\} \tag{2}$$

In this construction, the function f that we want to verify is the replacement algorithm (i.e., Replace) defined for the Merkle tree.

Concurrent Updating Operations. During the construction process using VC, it is possible to encounter a situation where two users are simultaneously updating the tree. However, this does not present a problem. Both deposits will be processed since the smart contract only accepts the VC proof when the queue is full. Consequently, only one user (the faster one) is required to pay and is subsequently compensated for the computation cost. Meanwhile, the other deposit will be placed as the first element of the queue.

5.4.1 Algorithms for On-Chain Mixer with VC

Contract Setup. The ContractSetUp algorithm of a mixer with VC differs from the MPB method (cf. Section 5.1): An additional instance must be initialized for the verifiable

Client$(\mathsf{sk}, \mathsf{amt})$

```
CreateDepositTx(sk, amt) :
Sample $(k_{dep}, r) \xleftarrow{\$} \{0,1\}^{\lambda}$
Compute $\mathsf{cm} = H_p(k_{dep}, r)$
Read $\mathsf{DepositQueue}, \mathsf{RootList}_{wdr,k}$
$\mathsf{root}^{old}_{dep} = \mathsf{RootList}_{wdr,k}[-1]$
Compute $\mathsf{path}_{\mathsf{index}}$ from $\mathsf{DepositQueue}$
$\mathsf{input} := (\mathsf{index}, 0, \mathsf{root}^{old}_{dep}, \mathsf{path}_{\mathsf{index}}, \mathsf{cm})$
$\mathsf{root}_{new}, \pi_{dep} \leftarrow$ VcProve$(\mathsf{ek}^{vc}_{dep}, \mathsf{input})$
return $\mathsf{wit} = (k_{dep}, r)$,
    $\mathsf{tx}_{dep} = (\mathsf{amt}, \mathsf{root}_{new}, \pi_{dep}, \mathsf{cm}, \mathsf{fee}_d)$
```

$\xrightarrow{\mathsf{tx}_{dep}}$

Smart Contract

```
AcceptDeposit($\mathsf{tx}_{dep}$)
$\mathsf{tx}_{dep} = (\mathsf{amt}', \mathsf{root}_{new}, \pi_{dep}, \mathsf{cm}, \mathsf{fee}'_d)$
Require:
  $\mathsf{amt} = \mathsf{amt}', \mathsf{fee}_d = \mathsf{fee}'_d, \mathsf{index} < 2^d$
Append $\mathsf{cm}$ to $\mathsf{DepositList}$
$\mathsf{index} = \mathsf{index} + 1$
$\mathsf{TotalFee} = \mathsf{TotalFee} + \mathsf{fee}_d$
// The subtree is full
if $\mathsf{DepositQueue}$ is full:
  $\mathsf{root}^{old}_{dep} = \mathsf{RootList}_{wdr,k}[-1]$
  Compute $\mathsf{path}_{\mathsf{index}}$ from $\mathsf{DepositQueue}$
  $\mathsf{input} := (\mathsf{index}, 0, \mathsf{root}^{old}_{dep}, \mathsf{path}_{\mathsf{index}}, \mathsf{cm})$
  Require VcVerify$(\mathsf{vk}^{vc}_{dep}, \mathsf{input},$
  $\mathsf{root}_{new}, \pi_{dep}) = 1$
  Append $\mathsf{root}_{new}$ to $\mathsf{RootList}_{wdr,k}$
  // Compensate the tree updater
  Do $\mathtt{tx}_{dep}$.sender.transfer(TotalFee)
  $\mathsf{TotalFee} = 0$
  ClearQueue(DespositQueue)
else :
  Enqueue(DespositQueue, cm)
return 1
```

Figure 7 Deposit interactions between the client and the smart contract in a mixer with VC.

computation (cf. Lines 8 and 9 in Fig. 6). The rest remains unchanged: We initialize a verifiable computation instance Π_{dep} with the algorithm Π_{dep}.VcInit with C_{dep} as input. We obtain two keys $(\mathsf{ek}^{vc}_{dep}, \mathsf{vk}^{vc}_{dep})$.

Deposit Interaction. Fig. 4 shows the deposit interaction between a client and the mixer with VC, in which the client computes the updated root when depositing into the mixer.

Client. The only difference between a mixer with the MPB and the one with VC is that the last client must compute and prove the valid computation of the new root in addition to cm and amt. The rest remains unchanged.

Contract. In contrast to MPB, the smart contract now performs fewer calculations. The smart contract checks the client's proof of the validity of the new root. If the proof is valid, the root of the contract is modified to match the client's root.

Withdraw Interaction. Because the verifiable computation only alters the deposit method, the way to withdraw coins in a mixer stays the same as in Section 5.1.

6 System Analysis

In this section, we prove that our improved mixers with MPB and VC both can guarantee *privacy*, *correctness*, *availability*, and *fairness*, while they achieve different degrees of *efficiency*.

6.1 Privacy

Our improved on-chain mixers achieve the privacy goals as follows.

6.1.1 Linking deposit and withdrawal transactions

Using a similar definition proposed in the AMR system [24], we first investigate the potential for an adversary to establish a link between a withdrawal transaction and its associated deposit transaction. We denote h as the height of the blockchain. Given a block height h, we define $\mathsf{CmpSet}^{\mathsf{h}}$ as the set of commitments of deposits made by honest users within a mixer pool. Within this context, for a given $\mathsf{cm} \in \mathsf{CmpSet}^{\mathsf{h}}$, we denote $\mathsf{tx}_{dep}(\mathsf{cm})$ as the deposit transaction which includes cm. Given a deposit transaction tx_{dep} including the commitment cm, and a withdrawal transaction tx_{wdr} with the nullifier sn, we say sn is originated from cm (denoted as $\mathsf{sn} \overset{origin}{\leftarrow} \mathsf{cm}$) if $\exists (k, r) \in \{0,1\}^{\lambda}, \text{s.t.}, \mathsf{cm} = \textsc{Com}(k, r) \wedge \mathsf{sn}_{wdr} = H_p(k, 0^{\lambda})$. Therefore, the adversarial advantage of linking the withdrawal transaction tx_{wdr} to its corresponding deposit transaction tx_{dep}, can be quantified as the probability that an adversary correctly guesses the commitment that originates the nullifier value in tx_{wdr} (cf. Definition 7).

▶ **Definition 7.** *(Adversarial Advantage in Transaction Linking) Let $\mathcal{A}$ be a PPT adversary, and $\mathsf{tx}^{\mathsf{h}}_{wdr}$ be a valid withdrawal transaction issued at block h by an honest user. Let $\mathsf{sn}^{\mathsf{h}}_{wdr}$ be the nullifier included in $\mathsf{tx}^{\mathsf{h}}_{wdr}$. We define the adversarial advantage as follows:*

$$\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},tx} = \Pr[\mathcal{A}(\mathsf{tx}^{\mathsf{h}}_{wdr}) \rightarrow \mathsf{tx}_{dep}(\mathsf{cm}),\ \ s.t.\ \ \mathsf{cm} \in \mathsf{CmpSet}^{\mathsf{h}} \wedge \mathsf{sn}^{\mathsf{h}}_{wdr} \overset{origin}{\leftarrow} \mathsf{cm}]$$

6.1.2 Linking deposit and withdrawal addresses

In practice, a user may utilize the same address to make multiple deposits or withdrawals within a mixer pool [39]. Rather than linking individual deposit and withdrawal transactions, the adversary may choose to target the linkability between addresses, specifically linking deposit and withdrawal addresses controlled by the same user.

Given a block height h, we denote $\mathsf{DepAddrSet}^{\mathsf{h}}$ as the set of addresses that are used to deposit coins into a mixer pool from honest users, and $\mathsf{WdrAddrSet}^{\mathsf{h}}$ as the set of withdrawal addresses of honest users.

Given a deposit address $\mathsf{addr}_d \in \mathsf{DepAddrSet}^{\mathsf{h}}$ and a withdrawal address $\mathsf{addr}_w \in \mathsf{WdrAddrSet}^{\mathsf{h}}$, we say addr_d and addr_w are linked if they belong to the same user, i.e., the user controls both the private keys of addr_d and addr_w. Denote as $\mathsf{addr}_d \overset{link}{\leftrightarrow} \mathsf{addr}_w$.

Therefore, the adversarial advantage of linking the withdrawal address addr_w to the corresponding deposit address addr_d, is the probability that an adversary can correctly determine if they are controlled by the same user (cf. Definition 8).

▶ **Definition 8.** *(Adversarial Advantage in Addresses Linking) Let $\mathcal{A}$ be a PPT adversary, and $\mathsf{addr}_w \in \mathsf{WdrAddrSet}^{\mathsf{h}}$ be an address that has been previously used to withdraw coins from the mixer pool before the block height h. We define the adversarial advantage as follows:*

$$\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},addr} = \Pr[\mathcal{A}(\mathsf{addr}_w) \rightarrow \mathsf{addr}_d,\ \ s.t.\ \ \mathsf{addr}_d \in \mathsf{DepAddrSet}^{\mathsf{h}} \wedge \mathsf{addr}_d \overset{link}{\leftrightarrow} \mathsf{addr}_w]$$

6.1.3 Privacy Analysis

We present the following claims to analyze the adversarial advantages of linking transactions and addresses in our improved mixers. For detailed proofs, we refer the reader to Appendix C.

▷ Claim 9. Under the assumption that all underlying cryptographic primitives are secure, the adversarial advantage in linking a withdrawal transaction at block height h to the corresponding deposit transaction (cf. Definition 7) satisfies: $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},tx} \leq \frac{1}{|\mathsf{CmpSet}^{\mathsf{h}}|} + \mathsf{negl}(\lambda)$.

▷ Claim 10. Under the assumption that all underlying cryptographic primitives are secure, the adversarial advantage in linking a withdrawal address at block height h to the corresponding deposit address (cf. Definition 8) satisfies: $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},addr} \leq \frac{1}{|\mathsf{DepAddrSet}^{\mathsf{h}}|} + \mathsf{negl}(\lambda)$.

Remark. Note that the increase in deposit transactions will not always decrease the adversarial advantage in linking deposit and withdrawal addresses, because an address can be used to deposit multiple times in a mixer pool. We also remark that the deposits in the queue are not considered in our privacy analysis, because they are not finalized and do not contribute to the set of commitments $\mathsf{CmpSet}^{\mathsf{h}}$ and the set of deposit addresses $\mathsf{DepAddrSet}^{\mathsf{h}}$.

6.2 Correctness

In the following, we prove that our system can guarantee correctness by demonstrating that the probability that an adversary can withdraw more times than deposits is negligible.

Consider an adversary who deposits into the mixer via a transaction $\mathsf{tx}^{\mathsf{h}}_{dep}$ at block h, and the transaction includes the commitment cm. We define the adversarial advantage of double withdrawing as the probability that the adversary can generate two withdrawal transactions linking to $\mathsf{tx}^{\mathsf{h}}_{dep}$ (cf. Definition 11).

▶ **Definition 11.** *(Adversarial Advantage in Double Withdrawing) Let $\mathcal{A}$ be a PPT adversary, which issues a deposit transaction tx^{h}_{dep} at block h. Let cm be the commitment included in tx^{h}_{dep}. We define the adversarial advantage as follows:*

$$\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},ww} = \Pr[\mathcal{A}(tx^{\mathsf{h}}_{dep}(\mathsf{cm})) \rightarrow \left(tx^{0}_{wdr}\left(\mathsf{sn}^{\mathsf{h}_0}_{wdr}\right), tx^{1}_{wdr}\left(\mathsf{sn}^{\mathsf{h}_1}_{wdr}\right)\right)$$
$$s.t.\ \mathsf{sn}^{\mathsf{h}_0}_{wdr} \overset{origin}{\longleftarrow} \mathsf{cm} \wedge \mathsf{sn}^{\mathsf{h}_1}_{wdr} \overset{origin}{\longleftarrow} \mathsf{cm} \wedge \mathsf{h}_0 > \mathsf{h} \wedge \mathsf{h}_1 > \mathsf{h} \wedge \mathsf{sn}^{\mathsf{h}_0}_{wdr} \neq \mathsf{sn}^{\mathsf{h}_1}_{wdr}]$$

Intuitively, double withdrawals occur when the adversary manages to generate two different nullifiers, $\mathsf{sn}^{\mathsf{h}0}_{wdr}$ and $\mathsf{sn}^{\mathsf{h}_1}_{wdr}$, both originating from the same commitment cm. However, the probability of this event is negligible if the underlying cryptographic primitives are secure (cf. Claim 12 and its proof in Appendix C).

▷ Claim 12. Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in successfully generating two distinct withdrawal transactions which correspond to the same deposit transaction (cf. Definition 11) satisfies: $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},ww} \leq \mathsf{negl}(\lambda)$.

6.3 Availability

Our system, comprising both MPB and VC, ensures availability. Similar to existing on-chain mixers such as TC [4] on Ethereum and TN [6] on BSC, our improved mixer can operate autonomously on smart-contract-enabled blockchains. It should be noted that a centralized regulator could affect the availability of an on-chain mixer. For instance, the regulator can impose sanctions [37] on on-chain mixers and require decentralized application frontends or Front-running as a Service (e.g., Flashbots MEV-boost relays) to censor mixer-related transactions [15]. However, users still have the option to utilize the command line interface or intermediary addresses to bypass the censorship and interact with the on-chain mixer smart contracts [40].

Table 1 Deposit cost calibration over time. The first $l-1$ clients' cost in the l_i-th deposit queue is calibrated by the total cost in the previous queue.

deposit queue		l_{i-1}	l_i
Cost	first $l-1$ clients	d_{i-1}^0	$d_i^0 = \frac{C_{i-1}}{l}$
	last client	d_{i-1}^1	d_i^1
	total	$C_{i-1} = (l-1) \cdot d_{i-1}^0 + d_{i-1}^1$	$C_i = (l-1) \cdot d_i^0 + d_i^1$

6.4 Fairness

To ensure fairness, our improved mixer employs a mechanism that ensures nearly equal fees are paid by all l clients in the same deposit queue when updating the Merkle tree. The approach involves each client in the queue, except the last one, paying an additional gas fee of d^0 in their deposit transactions to the smart contract. The contract then keeps track of the accumulated fees, totaling $(l-1) \cdot d^0$. When it is the turn of the l-th client to update the Merkle tree and clear the queue, the client will be responsible for paying the remaining gas fees, denoted as d^1.

Note that gas prices can vary over time, which means that the cost for each client needs to be adjusted accordingly. To achieve this, the payment fees for clients in the l_i-th deposit queue are calibrated based on the total cost of the l_{i-1}-th queue, where $i \geq 1$. Specifically, if the total update cost of the l_{i-1}-th queue is C_{i-1}, then the average cost for the previous $l-1$ clients in the l_i-th deposit queue is calculated as $\frac{C_{i-1}}{l}$. Table 1 illustrates the deposit costs for clients in a queue. It is worth noting that while the cost of the last client technically differs slightly from that of the remaining $l-1$ clients, we can prove that the improved mixer achieves fairness through the following analysis.

We first provide the definition of fairness for an on-chain mixer.

▶ **Definition 13.** *(ϵ-Fairness) Given two blocks b_0, b_1, and their corresponding gas prices $\text{PRICE}(b_0)$ and $\text{PRICE}(b_1)$, we say a mixer achieves ϵ-fairness if the deposit costs $\text{COST}(b_0)$ and $\text{COST}(b_1)$ in blocks b_0 and b_1 satisfy $|\text{COST}(b_0) - \text{COST}(b_1)| \leq \epsilon \cdot |\text{PRICE}(b_0) - \text{PRICE}(b_1)|$.*

We proceed to prove that our improved mixer can achieve ϵ-fairness when adopting the deposit cost calibration design in Table 1.

▷ Claim 14. Given a timeframe T covers at least two deposit queues l_{i-1} and l_i, whose final deposits occur in block $b_{l_{i-1}}$ and b_{l_i} respectively. Assuming that the deposit gas for updating the Merkle tree in different queues is constant, denoted as gas_{update}, then our improved mixer can achieve ϵ-fairness in the timeframe, where $\epsilon = gas_{update}$.

Proof Sketch. The total deposit costs in the two queues l_{i-1} and l_i satisfy: $|C_{i-1} - C_i| = gas_{update} \cdot \left|\text{PRICE}(b_{l_{i-1}}) - \text{PRICE}(b_{l_i})\right|$.

We further calculate the deposit cost differences of clients in the same queue. Note that the first $l-1$ clients' cost in the l_i-th queue is determined by the total cost in the previous queue. We thus have: $|d_i^0 - d_i^1| = \left|\frac{C_{i-1}}{l} - d_i^1\right| = \left|C_{i-1} - \frac{(l-1) \cdot C_{i-1}}{l} - d_i^1\right| = \left|C_{i-1} - (l-1) \cdot d_i^0 - d_i^1\right| = |C_{i-1} - C_i| = gas_{update} \cdot \left|\text{PRICE}(b_{l_{i-1}}) - \text{PRICE}(b_{l_i})\right|$.

Therefore, similar to the basic design (cf. Section 4), assuming that the gas price does not change in a short timeframe, i.e., $\left|\text{PRICE}(b_{l_{i-1}}) - \text{PRICE}(b_{l_i})\right| \approx 0$, the difference between the last client's deposit cost and the first $l-1$ clients' is approximately zero, i.e., $|d_i^0 - d_i^1| \approx 0$. Our improved mixer can thus guarantee fairness. We will provide a more detailed empirical analysis in Section 7.3.

6.5 Efficiency

6.5.1 Efficiency for On-Chain Mixer with MPB

Compared to the basic on-chain mixer design (cf. Section 4), our improved mixer system offers enhanced efficiency. We consider a deposit Merkle tree with a size of n. In a basic on-chain mixer, the computation cost for l deposits is $l \cdot O\left(\log(n)\right)$ as the entire Merkle tree for each deposit needs to be updated. However, in our improved mixer design with an l-length deposit queue, the Merkle tree only needs to be updated once per l deposits. Therefore, the computation cost for l deposits is $l + O\left(\log(n)\right)$.

We further define the ratio as $\gamma = \frac{l+O(\log(n))}{l \cdot O(\log(n))}$ to quantify the times of the saving deposit cost in our improved mixer compared to the basic mixer solution. Note that the deposit queue size l is much smaller than the Merkle tree size n; therefore, $\gamma = \frac{l}{O(\log(n))} + \frac{O(\log(n))}{l} \approx \frac{O(\log(n))}{l}$. The larger the queue size l, the more deposit cost clients can save. However, the finalization time of the deposit is also increasing over l. We should properly choose the size l to deal with the trade-off between the deposit cost and finalization time. We will provide the detailed evaluation results in Section 7.2.

6.5.2 Efficiency for On-Chain Mixer with VC

Thanks to VC, the deposit cost in a mixer can further be reduced. Consider that the computation cost of verifying an updated Merkle tree root is costVc, and the cost of generating an updated Merkle tree root is costGc. Therefore, in a basic on-chain mixer, the computation cost for l deposits is $l \cdot O\left(\log(n)\right) \cdot \mathsf{costGc}$ as the Mekle tree is updated per deposit. The computation cost for l deposits in an improved mixer with VC is $l + O\left(\log(n)\right) \cdot \mathsf{costVc} + \mathsf{costGc}$. The deposit cost saving is $\gamma^{vc} = \frac{l+O(\log(n))\cdot\mathsf{costVc}+\mathsf{costGc}}{l\cdot O(\log(n))\cdot\mathsf{costGc}} = \frac{l}{O(\log(n))\cdot\mathsf{costGc}} + \frac{\mathsf{costVc}+\frac{\mathsf{costGc}}{O(\log(n))}}{l\cdot\mathsf{costGc}} \approx \frac{\mathsf{costVc}+\frac{\mathsf{costGc}}{O(\log(n))}}{l\cdot\mathsf{costGc}}$. Note that in verifiable computations, the computation cost of verification costVc is inferior to the cost of proof generation costGc. Therefore, an on-chain mixer with verifiable computations can reduce deposit costs more than one with MPB. Section 7.4 will provide more quantification results.

7 Evaluation

In this section, we implement our improved mixer designs and evaluate their performance.

Cryptographic Primitives. We adopt Groth's zk-SNARK, Groth16 [20], as our instance of zk-SNARK owing to its efficiency in terms of its proof size and the calculations required by the verifier. We note that although the original Groth16 has malleable proofs, it can be implemented to achieve weak simulation extractability in practice [8]. We will elaborate more on the discussion in Section 9. We employ the Pedersen hash function [27] for H_p and the MiMC hash function [7] for H_{2p}, as cryptographic hash functions. Compared to arithmetic circuits that rely on other hash functions, such as Jubjub [3], arithmetic circuits that employ MiMC hash can produce a smaller number of constraints and operations. In addition to being created exclusively for SNARK applications, MiMC hash functions are also very gas-efficient for Ethereum smart contract applications.

Software Setup. For the arithmetic circuit design, we leverage the Circom library [9] to build the withdrawal circuit, C_{wdr}, for the relation specified in Equation 1. We utilize Groth16[20] proof system implemented by the snarkjs package [10] to construct the client's algorithms.. We establish a trusted environment for evaluating the mixer smart contract

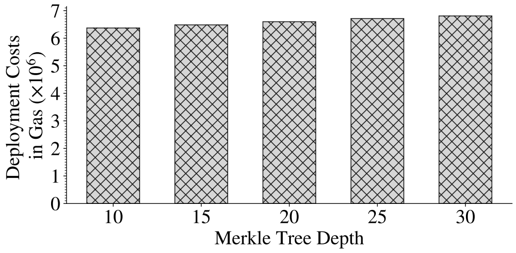


Figure 8 Cost of deploying the MPB smart contract with different Merkle tree sizes.

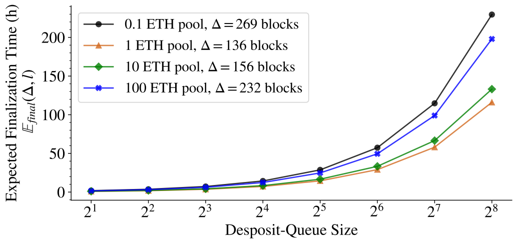


Figure 9 Expected deposit finalization time in TC ETH pools with various deposit queue sizes.

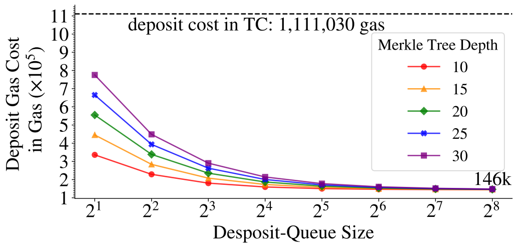


Figure 10 Average deposit costs per client for various queue sizes and Merkle tree depths. The deposit cost in TC (dashed line) $\approx 1.1m$ gas.

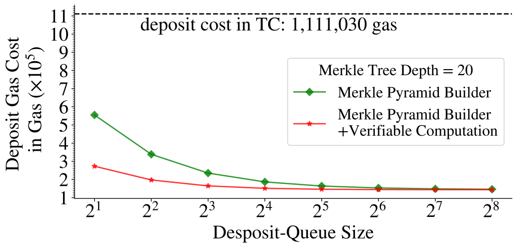


Figure 11 Average on-chain deposit costs per client for various deposit queue sizes. The VC approach can further reduce the deposit cost.

and clients. We deploy the on-chain mixer system on the EVM ganache [2]. **Hardware Setup.** We perform our experiment on a standard desktop system with an 11th Gen Intel(R) Core(TM) i7-11800H with 2.30G CPU and 16GB RAM in configuration.

7.1 Evaluating Merkle Pyramid Builder Costs

We evaluate the performance of the MPB system using different configurations of Merkle tree depths (i.e., $d = 10, 15, 20, 25, 30$) and deposit queue lengths ($l = 2, 4, 8, 16, 32, 64, 128$).

On-chain Deployment Costs. Fig. 8 shows the expense associated with deploying smart contracts. The Merkle tree depth does not have a huge influence on the deployment costs: the total cost is always between $6m$ and $7m$ gas. However, the deployment cost is a one-time expense that can be amortized over the duration of the contract.

On-chain Deposit Costs. Figure 10 provides a visualization of the cost reduction achieved by increasing the size of the deposit queue. We observe that after reducing or raising the depth of the Merkle tree, the cost decreases or increases, accordingly. Note that reducing the depth of the Merkle tree reduces the number of users, while raising the depth significantly increases the time required to compute the withdrawal proof. We can also observe that for a deposit queue of length 128, the gas costs are around $146k$, which is consistent amongst the various Merkle tree depths. Figure 10 also shows that this cost is merely $\frac{1}{7}$ of the cost of depositing in TC. Note that the costs associated with the deposit queue with a size of one, would roughly correspond to those of TC.

7.2 Evaluating Deposit Finalization Time

The deposit queue sizes will affect the deposit finalization time, i.e., the time that users need to wait for their deposits to be updated in the Merkle tree. To quantify the waiting time,

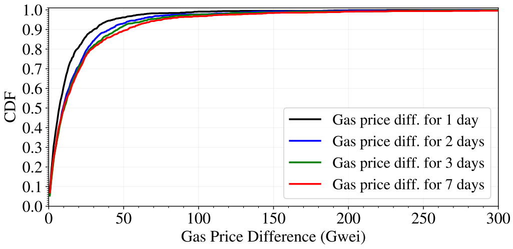


Figure 12 Distribution of gas price differences for TC deposit transactions.

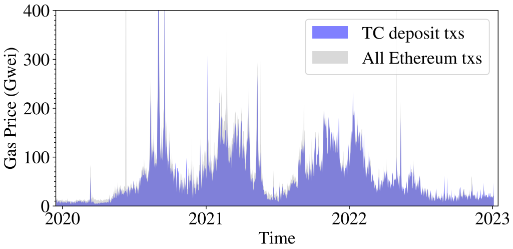


Figure 13 Average gas price of TC deposit transactions over time.

we crawl the historical deposits in the TC four ETH pools (i.e., 0.1, 1, 10, and 100 ETH pools) over time. As shown in Fig. 15 in Appendix B, from October 1st, 2020 to August 8th, 2022 (i.e., the date when Office of Foreign Assets Control (OFAC) announced the sanctions against TC), the average number of daily deposits in the four pools is 51 ± 25. Even after the OFAC sanctions' announcement, the average number is still 9 ± 8. Therefore, when the deposit queue size is set as 32, users merely need to wait for less than 4 days on average to ensure their deposits are finalized in a TC pool adopting our MPB method.

To further quantify the expected finalization time, we define Δ as the average timeframe between two deposits. Therefore, the expected finalization time $\mathbb{E}_{final}(\Delta, l)$ of a client in a deposit queue with the size l is $\mathbb{E}_{final}(\Delta, l) = \sum_{i=1}^{l} \frac{1}{l} \cdot (l - i) \cdot \Delta = \frac{(l-1)\cdot\Delta}{2}$.

Based on our empirical data, we can calculate that the average timeframes between two deposits in the TC 0.1, 1, 10, and 100 are 269, 136, 156, and 232 blocks respectively. Fig. 9 shows the expectation of the deposit finalization time when choosing different deposit queue sizes for TC ETH pools. Note that we set the block timeframe as $12s$, which corresponds to the slot time in the post-merge Ethereum. The results indicate that the expected deposit finalization time increases linearly over the deposit queue size.

7.3 Evaluating Deposit Gas Prices

In the following, we leverage the gas prices of TC historical deposit transactions to quantify our improved mixer's fairness.

Fig. 13 shows the average gas price of TC deposit transactions and all Ethereum transactions over time. We observe that generally, the TC deposit transactions have almost the same average gas price as other Ethereum transactions. To further measure the variance of gas price, we calculate the differences between sequential daily deposits in TC ETH pools. As shown in Fig. 12, we plot the distribution of the gas price differences for the deposits in sequential n days, where $n = 1, 2, 3$ and 7. We observe that more than 80% gas price differences in $n(n \leq 7)$ days are inferior to 30 GWei. Therefore, if we set the expected deposit finalization time as less than 7 days, the deposit cost differences between the last client and other clients in the same queue are inferior to 30 GWei × gas_{update}. Recall that gas_{update} is the total gas consumed for updating the Merkle tree in an on-chain mixer (cf. Section 6.4).

When setting the deposit queue size as 2^5, and the Merkle tree depth as 20, we have $gas_{update} = 2^5 \times 1.78 \times 10^5 = 5.69m$ gas (cf. Fig. 10). We can further calculate that the cost difference is $5.69 \times 10^6 \times 30 \times 10^9$ Wei $= 1.7 \times 10^{17}$ Wei $= 0.17$ ETH.

7.4 Evaluating Verifiable Computation Costs

TC supports anonymity mining [36] to incentivize users to user their ETH mixer pool. In order to claim the anonymity mining rewards, users can provide the Merkle tree roots to show their deposit and withdrawal transactions have already been successfully performed. This design is similar to our deposit with off-chain proof generation. Therefore, to evaluate the deposit cost of VC, we adopt a similar circuit of TC anonymity mining[4] to implement the VC deposit contract and test it locally. Our evaluation shows that the average deposit cost of VC is approximately $436k$. Thus, the saving deposit cost ratio of a client in a deposit queue of size l can be estimated as $\frac{1m}{436k \cdot l} = \frac{2.29}{l}$.

Note that in the improved mixer with VC, the smaller the queue size (i.e., l) is, the more cost per client can save (cf. Fig. 11). Therefore, when combining VC and MPB approaches, our improved mixer enables a significant improvement in the cost with a small deposit queue size (e.g., 4 or 8), which allows a short waiting time for deposit finalization.

8 Related Work

Blockchain Add-On Privacy Solutions. To enhance the privacy of non-privacy-preserving blockchains such as Bitcoin and Ethereum, numerous mixers are proposed in academic works or deployed in practice. For instance, Meiklejohn *et al.* [26] propose a smart-contract-based mixer named Möbius, which adopts linkable ring signatures and stealth addresses [29] to hide the addresses of transaction senders and recipients. However, the anonymity set size of Möbius is limited by the ring size, and the withdrawal cost increases linearly with the ring size. CoinJoin [25] is a Bitcoin mixer, which enables a user can collaborate with others to merge multiple transactions, thereby breaking the linkability between addresses.

State-of-the-art on-chain mixers [4, 6, 5, 1] are inspired by Zerocash [33] and are running on smart-contract-enabled blockchains to obfuscate the link between the users' deposit and withdrawal using zero-knowledge proof. Le *et al.* propose an on-chain mixer design [24], which incentivizes users to participate in a mixer. Shortly after, TC follows by adding anonymity mining as a deposit reward scheme for attracting users [36].

In addition to blockchain add-on privacy solutions, several existing works have proposed privacy-enhanced and regulated solutions for Central Bank Digital Currencies (CBDCs). These solutions include UTT [35], PEReDi [22], and Platypus [43]. Despite their theoretical promise, the practicality of implementing and running these solutions efficiently on the Ethereum platform remains unclear.

Blockchain Mixer Analysis. Although mixers can break the linkability among user addresses by design, in practice, mixers are not being used properly. Wu *et al.* [42] propose a generic abstraction model for Bitcoin mixers. They identify two mixing mechanisms, i.e., swapping and obfuscating, and present a method to reveal mixing transactions that leverage the obfuscating mechanism. Through analyzing the mixing activities in TC and TN, Wang *et al.* [39] proposes five heuristics to link the deposit and withdrawal addresses of on-chain mixers. Their heuristics can reduce a mixer's anonymity set size by more than 34.18%. Moreover, Wang *et al.*'s measurement work [39] also indicates that the reward mechanism of on-chain mixers tends to attract profit-driven but privacy-ignorant users, who do not contribute to the anonymity set size.

[4] https://github.com/tornadocash/tornado-anonymity-mining

9 Discussion

Trade-Off Between Latency and Cost. In our MPB mixer design, typically, the last depositor in a deposit queue initiates the Merkle tree update when the queue is full. However, we recognize the potential for including an optional function that allows any user to pay and trigger this update. This functionality would enable users to achieve faster-verified deposits by updating the tree without waiting for the queue to be filled. In such cases, the tree updater must strike a balance between latency and cost since the accumulated fees in an unfilled queue may not fully compensate for the updater's expenses.

Trade-Off Between Storage and Computation. In MPB, the subtree is updated with every deposit to optimize on-chain storage costs while maintaining a balance between storage and update efficiency. Specifically, in MPB, the total computation cost of updating a queue with l deposits is $l + O(\log(n))$, and the storage cost is $O(\log l)$. An alternative approach is to use a more "naive" batch update, where the mixer contract keeps a record of all l deposits in the queue and updates the subtree per queue. In this case, the total computation cost is still $l + O(\log(n))$, and the storage cost is $O(l)$.

Comparison with Other Data Structures. In addition to MPB, we have also conducted tests on other data-authenticated structures, such as dynamic RSA accumulators [11, 28, 12]. Accumulators offer the capability to generate proofs that demonstrate the membership of potential items in a specific set. One notable advantage of RSA accumulators is their efficiency in adding new items to an existing set. However, we observe that they do not effectively reduce deposit appending costs, as the mixer contract is required to perform expensive primality testing to identify prime numbers [21] as commitments on-chain. Furthermore, the process of proving membership in zero knowledge will incur significantly higher expenses.

Weak Simulation Extractability of Groth16. It is well-known that Groth16 zk-SNARK is malleable. Hence, one needs to use a deterministic nullifier to prevent double withdrawing. Also, in our design (refer to Eq. 1), it is necessary for a user to supply the associated private key sk as part of the private input (i.e., witness). This measure is taken to mitigate the risk that an adversary can replace a public key with his public key. However, in practice, we only need to use the public as a parameter to the public input. This is considered secure, given that Groth16 is recently proven to have the weak simulation extractability property [8]. This means that an adversary cannot produce a valid proof for a different input instance. Adopting this approach can notably reduce the proof generation costs for provers.

10 Conclusion

This paper investigates how to reduce the deposit cost of on-chain mixers. We first propose a design named MPB, which batches deposits in a queue and updates the Merkle tree per batch. This methodology reduces the Merkle tree update times and, thus, decreases the deposit cost. Specifically, our evaluation results show that MPB can achieve 7× fewer costs for depositing than state-of-the-art on-chain mixers. Moreover, we leverage off-chain verification to reduce the cost further. We also prove that our improved on-chain mixer designs can guarantee *correctness*, *privacy*, *availability*, *efficiency*, and *fairness*. We hope our work can engender further research into more secure and user-friendly on-chain mixers.

References

1 Cyclone. Available at: https://cyclone.xyz/bsc.

2 Ganache. Available at: https://trufflesuite.com/ganache/.

3 Jubjub. Available at: https://z.cash/technology/jubjub/.

4 Tornado cash. Available at: https://tornado.cash/, before August 8th, 2022.

5 Typhoon.cash. Available at: https://typhoon.cash/.

6 Typhoon.network. Available at: https://app.typhoon.network/.

7 Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge 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.

8 Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of groth's zk-snark. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*, pages 457–475. Springer, 2021.

9 Jordi Baylina, Kobi Gurkan, Roman Semenov, Alexey Pertsev, adria0, Ehud Ben-Reuven, arnaucube, Eduard S., and Marta Bellés. circomlib, 2020. Available at: https://github.com/tornadocash/circomlib#c372f14d324d57339c88451834bf2824e73bbdbc.

10 Jordi Baylina, Kobi Gurkan, Roman Semenov, Alexey Pertsev, adria0, Ehud Ben-Reuven, arnaucube, Eduard S., and Marta Bellés. snarkjs, 2020. Available at: https://github.com/tornadocash/snarkjs#869181cfaf7526fe8972073d31655493a04326d5.

11 Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology—EUROCRYPT'93: Workshop on the Theory and Application of Cryptographic Techniques Lofthus, Norway, May 23–27, 1993 Proceedings 12*, pages 274–285. Springer, 1994.

12 Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

13 Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020.

14 Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 928–946. IEEE, 2020.

15 Chainalysis. Understanding tornado cash, its sanctions implications, and key compliance questions, 2022. Available at: https://blog.chainalysis.com/reports/tornado-cash-sanctions-challenges/.

16 Dmitry Ermilov, Maxim Panov, and Yury Yanovich. Automatic bitcoin address clustering. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 461–466. IEEE, 2017.

17 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: The case of allowlist and denylist. *arXiv preprint arXiv:2302.06344*, 2023.

18 Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*, pages 465–482. Springer, 2010.

19 Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, volume 2021, 2021.

20 Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

21 Joe Hurd. Verification of the miller–rabin probabilistic primality test. *The Journal of Logic and Algebraic Programming*, 56(1-2):3–21, 2003.

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

23 Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

24 Duc V Le and Arthur Gervais. Amr: Autonomous coin mixer with privacy preserving reward distribution. *ACM Conference on Advances in Financial Technologies (AFT'21)*, 2021.

25 Greg Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin forum*, 2013.

26 Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(2):105–121, 2018.

27 Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 80–91. IEEE Computer Society, 2003. doi:10.1109/SFCS.2003.1238183.

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

29 Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, et al. An empirical analysis of traceability in the monero blockchain. *Proceedings on Privacy Enhancing Technologies*, 2018(3):143–163, 2018.

30 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available at: https://bitcoin.org/bitcoin.pdf.

31 Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed pki. In *Security and Cryptography for Networks: 10th International Conference, SCN 2016, Amalfi, Italy, August 31–September 2, 2016, Proceedings 10*, pages 292–309. Springer, 2016.

32 Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.

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

34 Peter Todd. Merkle mountain ranges, 2018. Available at: https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

35 Alin Tomescu, Adithya Bhat, Benny Applebaum, Ittai Abraham, Guy Gueta, Benny Pinkas, and Avishay Yanai. Utt: Decentralized ecash with accountable privacy. *Cryptology ePrint Archive*, 2022.

36 TornadoCash. Tornado.cash governance proposal, 2020. Available at: https://tornado-cash.medium.com/tornado-cash-governance-proposal-a55c5c7d0703.

37 U.S. DEPARTMENT OF THE TREASURY. U.s. treasury sanctions notorious virtual currency mixer tornado cash, 2022. Available at: https://home.treasury.gov/news/press-releases/jy0916.

38 Friedhelm Victor. Address clustering heuristics for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 617–633. Springer, 2020.

39 Zhipeng Wang, Stefanos Chaliasos, Kaihua Qin, Liyi Zhou, Lifeng Gao, Pascal Berrang, Benjamin Livshits, and Arthur Gervais. On how zero-knowledge proof blockchain mixers improve, and worsen user privacy. In *Proceedings of the ACM Web Conference 2023*, pages 2022–2032, 2023.

40 Zhipeng Wang, Xihan Xiong, and William J Knottenbelt. Blockchain transaction censorship: (in)secure and (in)efficient? *Cryptology ePrint Archive*, 2023.

41 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

42 Lei Wu, Yufeng Hu, Yajin Zhou, Haoyu Wang, Xiapu Luo, Zhi Wang, Fan Zhang, and Kui Ren. Towards understanding and demystifying bitcoin mixing services. In *Proceedings of the Web Conference 2021*, pages 33–44, 2021.

43 Karl Wüst, Kari Kostiainen, Noah Delius, and Srdjan Capkun. Platypus: a central bank digital currency with unlinkable transactions and privacy-preserving regulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2947–2960, 2022.

44 Haaroon Yousaf, George Kappos, and Sarah Meiklejohn. Tracing transactions across cryptocurrency ledgers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 837–850, 2019.

A Additional Building Blocks

Zero-Knowledge Proof. Zero-knowledge proof (ZKP) is a cryptographic primitive that allows a prover to convince the verifier about the correctness of some assertions without providing any meaningful information to the verifier. A zero-knowledge proof of some statement satisfies the following three properties [20]:

- *Completeness:* If *st* is true, an honest verifier will always be convinced by an honest prover.
- *Soundness:* For false statements, a prover cannot convince the verifier (even if the prover cheats and deviates from the protocol)
- *Zero-knowledge:* No verifier learns anything other than the fact that the statement *st* is valid if it is true. In other words, knowing the statement, but not the secret, is enough to construct a scenario in which the prover knows the secret.

Apart from *correctness, soundness,* and *zero-knowledge* properties of ZKPs, a zk-SNARK requires two additional properties, i.e., *succinctness* and *simulation extractability* [8].

Deposit Merkle Tree. Fig. 14 shows an illustrative example of the deposit Merkle tree.

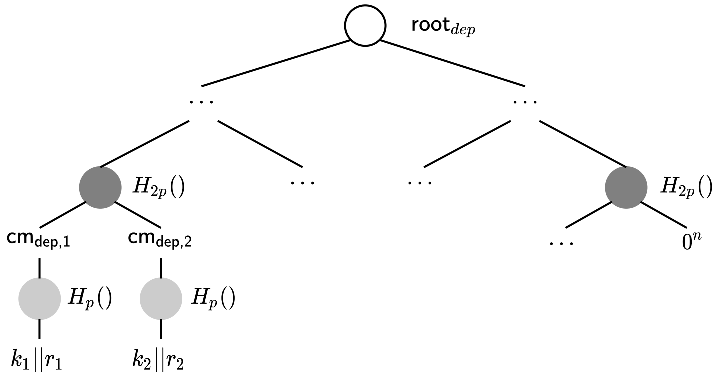


Figure 14 Illustrative example of the deposit Merkle tree, T_{dep}. The tree keeps track of commitments from clients' deposit transactions. The root of the tree, root_{dep} is used to verify the NIZK proofs from withdrawing transactions.

Merkle Mountain Range. Merkle Mountain Range (MMR) is a unique variety of Merkle tree which provides an efficient append function to maintain a balanced binary tree. Appending a new element to a MMR with n leaves requires traversing the right-most path of the tree and creating or modifying at most $O(\log(n))$ nodes in the tree. We adopt the definition from [14] to formally define a Merkle Mountain Range.

▶ **Definition 15** (Merkle Mountain Range (MMR)). *A Merkle Mountain Range (MMR), M, is a tree with a $n(n > 1)$ leaves, a root r, and the following properties:*

- *M is a binary Merkle hash tree.*
- *M has a depth of $\lceil \log_2 n \rceil$.*
- *Let $n = 2^i + j$, where $i = \lfloor \log_2 (n-1) \rfloor$, then* (i) *the subtree in r's left is a MMR with 2^i leaves, and* (ii) *the subtree in r's right is a MMR with j leaves.*

B Additional Evaluation Results

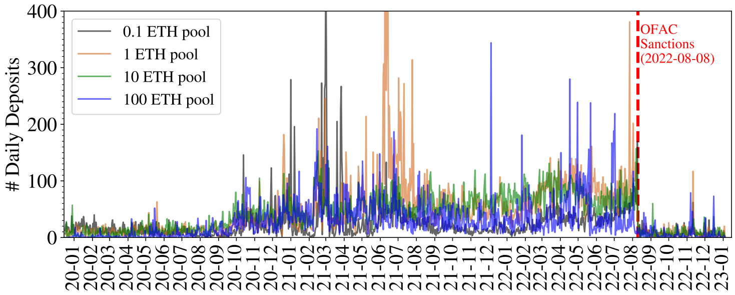


Figure 15 TC daily deposits over time. The average daily deposit number is 51 ± 25 during October 1st, 2020 and August 8th, 2022. The number decreases to 9 ± 8 due to the OFAC sanctions.

We provide the additional TC measurement results in Fig. 15, which can help us evaluate our improved mixers.

C Proof of Claims

The following two claims show the adversarial advantages in linking transactions and addresses in our improved mixers.

▷ Claim 16. Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in linking a withdrawal transaction at block height h to the corresponding deposit transaction (cf. Definition 7) satisfies: $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},tx} \leq \frac{1}{|\mathsf{CmpSet}^{\mathsf{h}}|} + \mathsf{negl}(\lambda)$.

Proof Sketch. Because we assume that the underlying cryptographic primitives are secure, the adversarial advantage in guessing correctly by breaking those primitives is negligible, i.e., $\mathsf{negl}(\lambda)$. Moreover, since each deposit transaction in the mixer pool adds a leaf to the Merkle tree, the probability of guessing a correct leaf is equal to the number of Merkle leaves that are not controlled by the adversary, i.e., $1/|\mathsf{CmpSet}^{\mathsf{h}}|$). Therefore, the adversarial advantage is $\mathsf{Adv}^{h}_{\mathcal{A},wdr} \leq 1/|\mathsf{CmpSet}^{\mathsf{h}}| + \mathsf{negl}(\lambda)$. Intuitively, the more deposits in the mixer pool, the harder the adversary can correctly link the deposit and withdrawal transactions.

▷ Claim 17. Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in linking a withdrawal address at block height h to the corresponding deposit address (cf. Definition 8) satisfies: $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},addr} \leq \frac{1}{|\mathsf{DepAddrSet}^{\mathsf{h}}|} + \mathsf{negl}(\lambda)$.

Proof Sketch. Analogously, the adversarial advantage in breaking the underlying secure cryptographic primitive is negligible $\mathsf{negl}(\lambda)$. Furthermore, each new deposit address will increase the size of the set $\mathsf{DepAddrSet}^{\mathsf{h}}$. Given a withdrawal address addr_w at block height h, the probability that the adversary can guess its corresponding deposit address is $1/|\mathsf{DepAddrSet}^{\mathsf{h}}|$. Therefore, the overall adversarial advantage is $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},addr} \leq 1/|\mathsf{DepAddrSet}^{\mathsf{h}}| + \mathsf{negl}(\lambda)$.

The following claim shows the correctness of our improved mixers.

▷ Claim 18. Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in successfully generating two distinct withdrawal transactions which correspond to the same deposit transaction (cf. Definition 11) satisfies: $\mathsf{Adv}^{\mathsf{h}}_{\mathcal{A},ww} \leq \mathsf{negl}(\lambda)$.

Proof Sketch. Consider a deposit transaction tx_{dep} including the commitment cm. If the adversary can find two distinct nullifiers $\mathsf{sn}^{\mathsf{h}_0}_{wdr}$ and $\mathsf{sn}^{\mathsf{h}_1}_{wdr}$ that are both originated from cm, then there exist two tuples (k_0, r_0) and (k_1, r_1) satisfying $H_p(k_0, r_0) = H_p(k_1, r_1) = \mathsf{cm}$, and $H_p(k_0, 0^\lambda) = \mathsf{sn}^{\mathsf{h}_0}_{wdr} \neq H_p(k_1, 0^\lambda) = \mathsf{sn}^{\mathsf{h}_1}_{wdr}$. This result breaks the preimage-resistant security of the hash function H_p (Note that k_0 and k_1 have the same length, and r_0 and r_1 have the same length). Inasmuch as the underlying cryptographic primitives are secure, the probability of breaking H_p's security is negligible.