AuditPay: Anonymous Payments with Controlled Oversight

Elkana Tovey
Hebrew University
Jerusalem, Israel
elkana.tovey@mail.huji.ac.il

Yossi Gilad
Hebrew University
Jerusalem, Israel
yossigi@cs.huji.ac.il

Aviv Zohar
Hebrew University
Jerusalem, Israel
avivz@cs.huji.ac.il

Abstract

This paper introduces AuditPay, a novel mechanism for blockchain mixers that enables controlled oversight through an "auditing budget." Auditors may monitor up to a budgeted number of addresses per epoch (e.g., an hour or a day), without revealing to users which addresses are monitored. Unlike traditional approaches that require users to voluntarily disclose viewing keys to trusted gatekeepers, AuditPay cryptographically enforces auditing compliance while preserving privacy for non-monitored user addresses and without introducing additional trust assumptions. At the core of AuditPay 's design is a selective encryption mechanism coupled with an efficient audit-key encoding scheme, enforced by a lightweight zero-knowledge proof. Users encrypt audit-relevant information under an audit key, which permits decryption only if the payment is to (or, alternatively, from) an address selected by the auditor for monitoring. We implement AuditPay as an Ethereum-based payment mixer and show through experiments on our prototype that its performance and gas-cost overheads are modest, providing a practical solution for balancing payment oversight with user privacy.

CCS Concepts

• **Security and privacy** → **Pseudonymity, anonymity and untraceability**; **Privacy-preserving protocols**;

Keywords

Blockchain; payment mixers; auditing; privacy; zero knowledge

ACM Reference Format:
Elkana Tovey, Yossi Gilad, and Aviv Zohar. 2026. AuditPay: Anonymous Payments with Controlled Oversight. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3830454.3832694

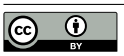
1 Introduction

Bitcoin [25] was initially lauded for its pseudonymous model, where value transfers occurred between addresses not directly tied to real-world identities. Over time, it became clear that the fully transparent ledger allowed chain-analysis techniques to link transactions and trace fund flows [31]. This gap between perceived privacy and actual traceability has inspired privacy solutions that break the linkage between a transaction's sender and recipient. Payment mixers such as Tornado Cash [29] on Ethereum form one common type of solution. The mixer lets senders deposit coins into a smart contract and lets recipients withdraw coins to a different address if they can prove in zero knowledge that they know a secret key corresponding to an unspent prior deposit. Thus, the mixer breaks the on-chain link between the deposit transaction and the subsequent withdrawal.

Governments often view such private payment solutions with suspicion because of their potential use in money laundering. For example, the U.S. Treasury sanctioned Tornado Cash after investigators found that a large share of its volume was used to launder stolen funds, a decision that was later overruled by the court [20, 22]. To mitigate compliance concerns and allow law enforcement to monitor suspicious activity, some privacy solutions, like Monero [26] and Zcash [19], introduced auditability features. This allows users to disclose a viewing key to an auditor, which reveals their transaction. This type of solution, however, falls short in two important ways. First, the auditor has to ask for the user's viewing key, which tells the user they are being tracked. This allows users to change their behavior or refuse to disclose their key (which may be possible if they are in a jurisdiction other than the auditor). Second, if the user concedes to disclose their viewing key, the auditor can see the user's transaction, even if it were benign, which reveals unnecessary information about them. Moreover, the auditor can ask for viewing keys from arbitrarily many users without publicly revealing that they are conducting large-scale monitoring, which allows it to completely circumvent the privacy guarantee.

In this work, we study the auditing problem for blockchain mixers to address the two problems above. We propose a solution called AuditPay that extends blockchain mixers to allow *restricted auditing* for payments. The auditor can commit to a set of addresses it wants to audit in every epoch (e.g., an hour or a day) and learn all about the payments involving those addresses either as a source or destination of funds. The auditor need not ask for users' consent, and its audit list is hidden from everyone else. Users, on the other hand, are guaranteed that the auditor can only track so many addresses per epoch (their "auditing budget"), and thus know that they are not likely to be monitored without reasonable suspicion (as the auditor needs to prioritize which addresses to track).

At the core of AuditPay is a selective encryption mechanism that forces users to encrypt data about their transactions in a way that the auditor can decrypt only if their address is being monitored. A key challenge in designing this mechanism is supporting auditing of multiple addresses in an efficient and private manner. For this purpose, AuditPay introduces an encoding scheme for audit keys that allows users to efficiently derive an encryption key, without learning whether the derived key corresponds to a monitored address or is merely a random value. To do so, we leverage the structure of cuckoo graphs [28] to obliviously encode keys associated with monitored addresses in a set of linear equations [15, 30] over an elliptic curve. This enables the retrieval



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832694

of a key via a small number of elliptic curve additions. From the user's perspective, the retrieved key is indistinguishable from a random group element, thereby revealing nothing about whether the address is currently being monitored. We discuss the design of this mechanism in detail in §6 and prove security properties in §7.

We implement AuditPay as an Ethereum smart contract that extends Tornado Cash and evaluate its performance and costs. We show that for users, the processing overhead of deposits is negligible, and that the processing time of withdrawals on standard consumer hardware is 10 - 15 seconds depending on system settings. The gas cost overhead for a deposit transaction is about 4%, and the withdrawal transaction costs about 1% less than Tornado Cash's. Lastly, for the auditor, we show how it can efficiently encode a set of audit keys for the addresses it monitors to fully utilize its auditing budget. For example, encoding a set of $40k$ addresses takes about an hour (for reference the OFAC list has about $18.5k$ records [27]).

In sum, this work makes the following contributions:

(1) A design of AuditPay, a mechanism for extending blockchain mixers to allow for restricted auditing.
(2) A novel selective encryption mechanism that enables efficient tracking of multiple addresses without revealing information about which addresses are being monitored.
(3) An implementation of AuditPay as an Ethereum smart contract along with performance benchmarks showing that the overheads are small compared to mixers without the auditing functionality.

2 Related work

Addressing auditing and compliance concerns in private blockchain solutions has recently become an active area of research interest. This interest was largely catalyzed by the inclusion of the Tornado Cash contract address on the OFAC Specially Designated Nationals (SDN) list for facilitating illicit activity. This effectively criminalized the use of Tornado Cash for US persons. There have been several types of proposed solutions to address these concerns, which we overview below. We conclude by contrasting attribute and identity-based encryption with AuditPay's selective encryption.

Viewing keys. This approach lets a user provide an auditor with a "viewing key" that reveals to the auditor all the details about the user's transactions [19, 35]. Viewing keys have two main drawbacks: first, the auditor has to ask for the viewing key, which reveals to the user that they are being tracked. The second is that without a mechanism to enforce or at least credibly monitor an auditing budget, the auditor may request viewing keys from arbitrarily many users without revealing that they are conducting large-scale monitoring. Arguably, viewing keys give the auditor too much power and circumvent the mixer's privacy guarantee.

User whitelists. Rather than revealing data to an auditor, some approaches focus on allowing access to a privacy service, like the mixer, only for pre-approved users. In many of these solutions, typically users must go through KYC, and are then limited to interacting only with other "approved" users. Some companies, such as Chainalysis and TRM Labs offer compliance-as-a-service solutions that check whether a given blockchain address is legitimate and allow automating user admission. Of course, this requires trusting those centralized compliance providers who can block addresses at will. Moreover, this approach only controls access to the mixer, it does not support an auditing functionality. A law enforcement agency cannot, for example, monitor activity that involves suspicious addresses.

User-provided proofs [4]. An alternative solution to having a centralized agency maintaining a user whitelist is to have users disclose critical information about their accounts as needed, particularly, which accounts funded their account. In this approach, users can decide whether and when to disclose information about their account (e.g., only when registering with an exchange to withdraw their funds). However, similarly to whitelist solutions, this approach does not allow for auditing. Specifically, users can decide to only access services that do not require revealing their information and never disclose it; they could also simply refuse to disclose it to a foreign auditor.

Accountable privacy [32]. This approach assigns each user a privacy budget. Once the user exceeds her budget, her future transactions are deanonymized. To enforce the privacy budget, this approach relies on a multiparty protocol between permissioned validators. This approach differs from AuditPay on two key properties. First, it relies on an honest majority across the (small set of) permissioned validators; if this assumption fails, then those validators could view all transactions regardless of the users' privacy budgets. AuditPay, in contrast, relies only on cryptographic assumptions to restrict the auditor. Second, accountable privacy lets users manage their privacy budget and potentially avoid monitoring by limiting themselves to a few transactions. In AuditPay, the auditor decides which addresses to monitor (within its budget), so monitoring is up to the auditor's decision rather than the user's transactions, which we believe is more suitable for law enforcement.

Anonymity budgets. PRCash [37] and Platypus [38] enforce compliance in central bank digital currency systems by assigning users a privacy budget that limits transaction volume; once exhausted, transactions are deanonymized. Both systems operate in permissioned blockchain settings with trusted validators. AuditPay takes the opposite approach: rather than limiting user transaction volume, it restricts the auditor to a budget of monitored addresses on a permissionless blockchain, with no trusted validators required.

Ex-post accountability. Garman et al. [16] propose an accountability mechanism in which an authority decrypts transaction data offline and publishes a proof that it adhered to a search limit. While this provides oversight transparency, it does not cryptographically prevent mass surveillance: there is no enforced limit on the number of accounts the authority may inspect. AuditPay, in contrast, bounds the auditor's monitoring scope cryptographically, and relies on lighter cryptographic tools than Garman et al.'s construction.

Committee-based auditing. Androulaki et al. [2] and PEReDi [21] achieve privacy with auditability in permissioned settings by distributing trust among committees of validators. Privacy relies on an honest majority within each committee; a colluding majority breaks all privacy guarantees. Additionally, Androulaki

et al. partition users into groups each assigned to a single auditor, exposing all group transactions to that auditor. AuditPay relies solely on cryptographic assumptions and does not require trustworthy committees or user partitioning.

Auditable payment channel hubs. AuditPCH [23] and AUPCH [24] add auditability to payment channel hubs by having users encrypt linkage metadata under an auditor's key. Both rely on a non-collusion assumption between the central payment hub and the auditing agent; a trust assumption AuditPay avoids. Furthermore, neither work bounds the auditor's monitoring scope: the auditor can inspect all transactions. In AuditPay, the auditor commits in advance to a budgeted set of (hidden) target addresses and can only unmask transactions involving those specific targets.

Related cryptographic techniques. Attribute-based and identity-based encryption [7, 9] are cryptographic schemes that allow fine-grained access control over encrypted data based on the decryptor's attributes or identity. Although these tools appear similar to our selective encryption scheme, they have two key differences. First, decryption success is based solely on attributes or identity of the auditor, whereas in AuditPay, revealing information depends on attributes of the users (the payer or payee). Moreover, those schemes rely on a centralized master authority that issues attribute or identity keys; a rogue authority could issue keys with increased privileges without detection.

3 Overview

AuditPay introduces controlled oversight for blockchain mixers. Traditional mixers, such as Tornado Cash [29], establish privacy by delinking deposits from withdrawals. Any user can deposit funds into the mixer from one address and later withdraw them to a different address, without revealing which deposit corresponds to said withdrawal. Thus, mixers offer a strong privacy guarantee to users by unlinking payers from payees.

In many scenarios, it is desirable to allow a regulatory authority to track transactions involving a limited set of suspicious payer or payee addresses, while preserving the privacy of all other users. For example, a government agency may wish to monitor addresses associated with money laundering, while preserving the privacy of all other users. Crucially, effective monitoring requires that monitored users do not learn they are being tracked, as this may cause them to change their behavior and evade detection.

In AuditPay, an *auditor* can link deposit-withdrawal pairs when either the originating depositor address or the withdrawal's recipient address belongs to a secret, committed, *audit set* S which holds a monitored set of addresses. For all users that do not interact with the monitored addresses, AuditPay gives exactly the same privacy guarantees as traditional mixers. Figure 1 illustrates AuditPay's key components and operation.

In AuditPay, the auditor chooses S by publishing the *auditkey projection vector* V and committing to it (Figure 1 steps (1a), (1b)). The vector is an encoding of a set of *audit public keys* related to the addresses it chooses to monitor. Each key is constructed so that the auditor can only decrypt information encrypted with it if the encryptor owns the corresponding monitored address. The auditor can update S over time, at predetermined epoch boundaries, but any deposit-withdrawal pair is monitored against the audit set current at withdrawal time.

Although the audit set is kept secret, its maximal size $|S|$ is known to everyone since it is bounded by the size of the vector the auditor publishes. Ensuring that users know the bound on $|S|$ is important, since it restricts the auditor's power. It guarantees that an overreaching auditor cannot simply ask to monitor all addresses and thus break privacy for everyone; instead, the auditor has to prioritize monitoring transactions to/from suspect addresses they care about.

Linking deposit-withdrawal pairs happens in two steps. The first step happens when funds are deposited in the mixer contract, and the second is when they are withdrawn. As illustrated in Figure 1 step (2), to deposit funds, the depositor creates a *commitment note* and deposits it along with the funds into the mixer. To allow tracking of the depositor's address at withdrawal time, the mixer appends a cryptographic commitment to the depositor's address to the commitment note when accumulating the note into its internal Merkle commitment tree.

To transfer funds (Figure 1 steps (3) & (4)), the depositor "spends" the commitment note created at deposit time, and also runs a procedure designed to reveal the deposit address of this transaction to the auditor if either the depositor's or the recipient's address is in S. The user encrypts metadata that links the transaction under an audit public key that belongs to the committed audit set, and proves in zero-knowledge that this (published) encryption is well-formed (relative to the data encrypted and the addresses encrypted against). The purpose of the commitment to the depositor's address at deposit time is to enable this zero-knowledge proof at withdrawal time. If neither the depositor's nor the recipient's address is in S, decryption under the said public key yields no information about the transaction (Figure 1 step (5)).

Limitations. Auditing open payment mixers (that allow addresses without KYC) comes with inherent challenges when monitoring *recipient* addresses. A user of interest to the auditor who wishes to evade scrutiny can create a fresh unseen address for each incoming transaction. Still, even for such open systems, users may sometimes be forced to publicly list an address to receive funds, e.g., in the case of a darknet market vendor that must list a receiving address to accept payments or an illicit organization asking for donations. In contrast, monitoring *depositor* addresses does not have this issue. Since the auditor sees the depositor's address moving the funds to the mixer, the auditor can decide to monitor where those funds go next at that time. If the mixer enforces KYC, users cannot evade monitoring by creating fresh addresses, as each address is bound to a verified identity, making AuditPay's auditing guarantees strictly stronger.

Another limitation that we do not handle is linking addresses in retrospect. AuditPay does not support an auditor who tries to find the source of funds after they were withdrawn to an unmonitored address. At this point, the users may have departed and are no longer interacting with the mixer. Thus, to support such retrospective linking, the mixer would have to retain additional information about past transactions and selectively reveal it to the auditor. We leave this extension to future work.

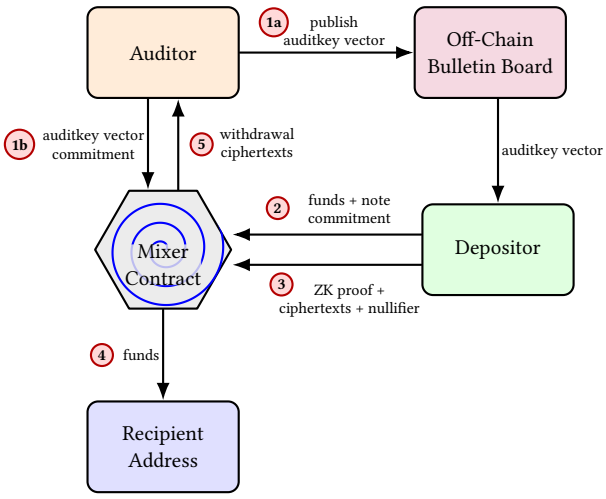


Figure 1: AuditPay protocol as five steps: (1) auditor publishes the auditkey projection vector off-chain and commits to it on-chain via the mixer contract. (2) The depositor deposits commitment note and funds into the mixer. (3) Some time later, the depositor transfers the funds to a recipient address by submitting a ZK proof along with ciphertexts and a nullifier. The transfer cannot be linked to the initial deposit unless the depositor or recipient address is in the audit set. (4) Upon verification of the proof, the mixer sends funds to the recipient address. (5) The auditor reads the withdrawal ciphertexts and attempts to decrypt them. They successfully decrypt and link the depositor and recipient if either of their addresses is in the audit set.

3.1 Threat Model

We divide AuditPay's assumptions into two categories: cryptographic assumptions about the primitives we use, and trust assumptions about the entities involved in the protocol. We also assume the depositor and recipient coordinate the payment off-chain through a metadata-private communication channel. Under its assumptions, which we detail below, AuditPay should allow an auditor to trace transactions involving monitored addresses, while preserving the privacy of all other users. Moreover, it should ensure that users learn nothing about the set of addresses the auditor monitors. We outline these security goals below, and formalize and prove AuditPay meets them in §7.

Cryptographic assumptions. We assume the existence of a collision- and preimage-resistant hash function, which we model as a random oracle. We also assume the existence and security of a zero-knowledge proof system. Finally, we assume the hardness of the discrete logarithm problem over elliptic curves, and moreover, the hardness of decisional Diffie-Hellman over that curve.

Trust assumptions. We assume that the auditor may collude with corrupt users to attempt to deanonymize transactions that do not contact monitored addresses and may try to exceed its monitoring budget. We assume that users may collude with each other to try to circumvent detection or learn about address membership in S.

3.2 Security Goals

AuditPay builds over traditional mixers, such as Tornado Cash, and extends their unlinkability property between deposits and withdrawals. Its first security properties are inherited from the underlying mixer, that is, that funds cannot be double-spent (payment validity) and that an attacker cannot link between deposits and withdrawals (payment privacy). In §4 we describe the mechanism that achieves these properties, which AuditPay builds upon. In addition to these, AuditPay introduces the following three security goals to capture payment auditing.

Selective linkability. The mixer provides unlinkability for all deposit-withdrawal pairs that do not involve any address in S. For these pairs, the auditor's view is computationally indistinguishable from its view in the underlying mixing protocol. For all deposit-withdrawal pairs where either the depositor or the recipient address is in S, the auditor can link the deposit to the withdrawal.

In AuditPay, a deposit-withdrawal pair remains unlinkable even if the recipient of a transaction colludes with the auditor. This arises as the recipient does not learn any information about the deposit address at any point in the protocol, and thus cannot leak this information to the auditor. This is because the depositor, not the recipient, generates and submits the withdrawal proof.

Restricted monitoring. The auditor is restricted to a public monitoring budget $|V|$ per fixed monitoring epoch. The monitored set S ($|S| \leq |V|$) is committed at the start of a monitor epoch and cannot be changed during its duration. Any deposit-withdrawal pair is monitored relative to the set S that is fixed at withdrawal time. The auditor provides unlinkability for all deposit-withdrawal pairs that do not involve an address that was in S at withdrawal-time (or alternatively at deposit-time).

Membership hiding. A user cannot distinguish between a candidate audit set and the true audit set S: the published auditkey vector V reveals nothing about S. Moreover, users cannot craft a set of up to $|V|/2$ addresses that the auditor is unable to monitor together.

3.3 Efficiency Goal

AuditPay aims to achieve the above security goals with only a modest overhead compared to the standard mixer baseline. Its chain gas costs should be comparable to the baseline mixer, with only a small constant overhead. Off-chain computational overhead for the client (proof generation, encryption, etc.) should be logarithmic in the size of the audit set, and off-chain setup costs for the auditor should be polynomial in the size of the audit set. We empirically show that AuditPay is efficient through evaluation (§9).

4 Background on Mixers

We briefly review blockchain mixers (also known as privacy pools) implemented as smart contracts that support deposit and withdrawal operations of fixed denominations. AuditPay builds on this model to add restricted auditing functionality.

A mixer (such as Tornado Cash) maintains an append-only Merkle tree of note commitments. To *deposit*, a user submits funds and a *note commitment*. To *withdraw*, the user proves in

zero knowledge that they own one of the previously deposited (unspent) notes without revealing which one. Unlinkability stems from proving membership in the set of commitments rather than referencing a specific deposit.

Deposits. Let $\mathbb{B} = \{0,1\}$ denote the Boolean alphabet, and let $\mathbb{B}^\lambda$ denote bitstrings of length λ. To deposit, a user selects $\kappa, \rho \in \mathbb{B}^{248}$ and computes the note commitment $C = h(\kappa||\rho)$. Here, k is a secret used later to derive a *nullifier* - a string used to prevent double spends, ρ is a *randomness*, and h is a cryptographic hash function such as defined in [1]. The depositor submits C and some fixed sum to the contract. The contract receives the sum and appends C to an append-only Merkle tree.

Withdrawals. To withdraw, the user must know the opening of an unspent note commitment (κ, ρ) and its position ℓ in the Merkle tree (knowledge of (κ, ρ) implies knowledge of ℓ). A withdrawal transaction includes a withdrawal address, a nullifier hash $null = h(\kappa)$, and a proof of knowledge of (κ, ρ) along with its Merkle path relative to the current root.

The contract completes the transaction if the proof is valid and *null* has not yet been published. We detail the mixer smart contract in Algorithm 1. Changes to the standard mixer implementation to support AuditPay's selective auditability appear in blue. The contract maintains a Merkle root of committed nodes and a set of spent nullifiers to prevent double spends. The functionality we add is mainly an auditor-controlled commitment to a set of monitored addresses, and logic to advance the epoch and update the set that requires the auditor to notify of changes at least Δ_{min} time units before the current epoch ends.

Algorithm 1 Mixer Smart Contract. Changes to AuditPay vs. the reference mixer implementation appear in blue.

```
1:  const AMNT ← fixed deposit/withdrawal amount
2:  spent_nullifiers ← ∅
3:  root ← empty Merkle tree root
4:  auditor ← set at contract deployment
5:  next_com ← commit to initial auditkey vector at deployment
6:  next_epoch ← 0
7:
8:  procedure DEPOSIT(amount, sender, C_mixer)
9:      require amount = AMNT
10:     root ← UPDATEMERKLETREE(root, h(C_mixer, sender))
11: end procedure
12:
13: procedure WITHDRAW(recipient, π, null, ct)
14:     ADVANCEEPOCH()
15:     require null ∉ spent_nullifiers
16:     require VERIFYZKP(π, (null, root, ct, active_com, recipient))
17:     spent_nullifiers ← spent_nullifiers ∪ {null}
18:     TRANSFER(AMNT, recipient)
19:     EMITEVENT(Withdraw(recipient, ct))
20: end procedure
21:
22: procedure SETNEXTAUDITKEYVECTOR(com)
23:     require caller = auditor
24:     ADVANCEEPOCH()
25:     next_com ← com
26:     next_epoch ← CURRENTEPOCH() + 1
27:     while TIMETOEPOCH(next_epoch) < Δ_min do
28:         next_epoch ← next_epoch + 1
29: end procedure
30:
31: procedure ADVANCEEPOCH()
32:     if CURRENTEPOCH() ≥ next_epoch
33:         active_com ← next_com
34: end procedure
```

5 Payment Auditing Building Blocks

We introduce two mechanisms for building AuditPay's auditing protocol. The first is a selective encryption scheme (§5.1) based on ECDH. This scheme allows encrypting linkage metadata about the depositor to an *audit public key* derived from an address. The auditor can decrypt this metadata only if the address for key derivation matches one of the addresses the auditor monitors.

However, selective encryption alone gives an inefficient solution. Naively requiring users to encrypt the linkage metadata to all audit keys for tracked addresses would lead to a linear growth in transaction size and gas costs with respect to the number of tracked addresses. The second mechanism addresses this efficiency issue (§5.2). It creates an oblivious auditkey vector V mapping tracked addresses to audit public keys. The vector is oblivious in the sense that it does not leak information about the addresses whose audit keys it encodes: Computing a key for some address using V returns the corresponding audit public key if the address is tracked; otherwise, the result is a pseudorandom value computationally indistinguishable from a valid audit public key.

5.1 Selective Encryption

Algorithm 2 presents AuditPay's selective encryption scheme. We design it to simultaneously meet two goals. First, we want successful decryption to be possible only when the encryptor's address matches one committed by the auditor. Second, we want it to be difficult to know which address is actually linked to the encryption key used.

Selective encryption uses the following cryptographic primitives, all instantiated with a security parameter λ (e.g., for achieving 128-bit security):

- An elliptic curve group $\mathbb{G}$ of prime order q with generator G, where the Decisional Diffie-Hellman (DDH) assumption holds.
- A hash-to-curve function $H : \{0,1\}^* \to \mathbb{G}$ modeled as a random oracle.
- A Key Derivation Function $\mathsf{KDF} : \mathbb{G} \to \mathcal{K}$, where $\mathcal{K}$ is the key space for a symmetric cipher, also modeled as a random oracle.
- A symmetric encryption scheme (SymEnc, SymDec) that is IND-CPA secure.

For an address a, one can derive an *audit key* $\mathsf{AK_a}$ by running AuditKey with input a. To selectively-encrypt data m under audit key $\mathsf{AK_a}$, a sender with address a' runs Encrypt with inputs $\mathsf{AK_a}$, a', and m.

Algorithm 2 Selective Encryption Scheme

procedure KeyGen()
  $sk \in_R \mathbb{Z}_q$
  $PK \leftarrow sk \cdot G$
  return (sk, PK)
end procedure

procedure AuditKey(a)
  $(sk, PK) \leftarrow$ KeyGen()
  $AK_a \leftarrow PK + H(a)$
  return (sk, AK_a)
end procedure

procedure Encrypt(AK_a, a', m)
  $r \in_R \mathbb{Z}_q$
  $R \leftarrow r \cdot G$
  $K \leftarrow r \cdot (AK_a - H(a'))$
  $K_{\text{sym}} \leftarrow \text{KDF}(K)$
  return $ct \leftarrow (\text{SymEnc}_{K_{\text{sym}}}(m), R)$
end procedure

procedure Decrypt(sk, ct)
  Parse: $(c', R) \leftarrow ct$
  $K' \leftarrow sk \cdot R$
  $K'_{\text{sym}} \leftarrow \text{KDF}(K')$
  $m, error \leftarrow \text{SymDec}_{K'_{\text{sym}}}(c')$
  if $error$ = nil
    return m
  else
    return $\perp$
end procedure

Correctness. If $a = a'$, then we subtract from $\mathsf{AK_a}$ the same point $H(a)$ that was added to the public key when creating the audit key, so encryption is performed with PK that corresponds to the secret key sk. In this case, decryption reduces to ECDH, allowing the holder of sk to recover m by running Decrypt and the resulting ciphertext as inputs.

Security. If $a \neq a'$, then encryption was done with a random key K (since H is modeled as a random oracle). In this case, decryption returns an error, and the ciphertext is indistinguishable from random. We formally define and prove the security property in §7.

5.2 Encoding Audit Keys for Multiple Addresses

Lookups in the auditkey vector V must be efficient to perform by users and verify in the mixer's smart contract. Additionally, since V is public, it must not leak information about S, the set of addresses that the auditor tracks (beyond its size limit). We now discuss how to encode the audit keys for all addresses in S into the auditkey vector V while meeting the above requirements. We begin by describing a simple cuckoo-hash-based approach that constructs a table T of addresses, describe its security flaw and then present AuditPay's construction, which also maps addresses to public keys using hash functions, but addresses this issue.

5.2.1 Cuckoo-Hashing-based Construction and Its Flaws. As a first attempt, one might consider using a standard cuckoo hashing approach to construct a table of audit keys. In this construction, a table T consists of $|T| \geq |S|$ entries, each containing an elliptic curve point. An address a is hashed to k candidate locations $\{h_1(a), \ldots, h_k(a)\}$. If a is tracked, its associated audit public key, AK_a, is stored in one of these locations. When a user withdraws coins from the mixer's smart contract, they selectively encrypt (Algorithm 2) the linkage metadata with the keys found at *all* of the k locations. If $a \in S$, then one of these encryptions will be to a key that the auditor can decrypt.

This construction is simple to implement, since it uses standard techniques for building cuckoo hash tables. However, it suffers from several drawbacks. The first, and most critical, is security. This construction leaks information about S. In particular, users can try hashing different addresses to find small sets of addresses (smaller than $|S|$) that they know cannot be fully accommodated by T. The second issue is efficiency. The on-chain gas costs for withdrawal grow linearly with k, as the user must prove encryption to all candidate table locations. Additionally, cuckoo hashing does not allow full utilization of the table T for an independently chosen set of keys. Namely, there is some gap between the advertised tracking budget $|T|$ and the budget the auditor is actually able to use. Increasing utilization requires increasing the number of hash functions (k), which in turn increases the on-chain withdrawal cost for all users. A very high number of hash functions (k in the hundreds) mitigates the security concern too [39], but proportionally increases the on-chain costs, making this solution impractical.

To illustrate these problems, consider the example for $k = 2$ and a user who calculates the two hashes for each of multiple addresses. Each address is hashed to two locations in the table, so the chance that the two addresses collide in *both* locations is $|T|^{-2}$. Given the birthday paradox, the user will only need to enumerate $O(|T|)$ addresses in expectation until it finds two addresses that cannot be monitored together. If one of the addresses in the pair is a priori likely to be tracked (e.g., is known to belong to a bad actor), then the user can be (almost) sure the other address is not tracked.

5.2.2 Constructing an Oblivious AuditKey Projection Vector. Inspired by the cuckoo-hashing approach and motivated by its drawbacks, we now present AuditPay's mechanism for encoding the set S into an auditkey projection vector V, illustrated in Algorithm 3. The encoding scheme is also based on hashing addresses into entries in a vector. We solve both security and efficiency issues of the cuckoo-hashing approach by mapping each address into a linear combination of the elliptic curve points that we then use as the address's audit public key. Figure 2 illustrates AuditPay's vector construction technique, which we now detail.

Instead of directly storing the audit key AK_a at one of these locations, a defines a linear combination of the values stored at these locations. Our vector construction procedure (`AuditKeyVecGen` in Algorithm 3) uses $2k$ hash functions as follows. Hash functions $h_1, \ldots, h_k$ map each address to k locations in V, and hash functions $c_1, \ldots, c_k$ determine the weights in the linear combination for each vector entry (see Figure 2). The auditor solves a linear equation to find the elements in $\mathbb{G}$ for which the linear combinations induced

Algorithm 3 AuditKey Vector Encoding and Retrieval

procedure AuditKeyVecGen(SE, S, $|V|$, $\{h_i\}_{i=1}^k$, $\{c_i\}_{i=1}^k$)
  Require: $|S| \le |V|$, S in priority order
  while $|S| < |V|$ **do** S.Append(rand addr) ▷ Pad S
  Equations $\leftarrow$ []
  $AKs \leftarrow$ []
  for $a \in S$ **do**
    idx $\leftarrow [h_i(a) \mod |V| \textbf{for} i \in [k]]$
    coef $\leftarrow [c_i(a) \textbf{for} i \in [k]]$
    if a's equation is linearly dependent on Equations
      continue ▷ Drop a
      ▷ SE is the suite of procedures from Algorithm 2
    $sk_a, AK_a \leftarrow$ SE.AuditKey(a)
    $AKs.Append(\langle sk_a, AK_a \rangle)$
    Equations.$Append(\sum_{i=1}^k \text{coef}[i] \cdot x_{\text{idx}[i]} = AK_a)$
  end for
  ▷ Solve the linear equation system for variables x_i over $\mathbb{G}$
  $V \leftarrow$ Solve(Equations) ▷ Uniformly random solution
  return V, AKs
end procedure

procedure RetrieveKey(SE, V, a)
  $AK \leftarrow 0_{\mathbb{G}}$ ▷ Neutral element of Group $\mathbb{G}$
  for $i \in [k]$ **do**
    idx $\leftarrow h_i(a) \mod |V|$
    $AK \leftarrow AK + c_i(a) \cdot V[\text{idx}]$
  end for
  return AK
end procedure

by each address in the monitored set equals the audit key for that address. The auditor processes the addresses in order of priority and drops any address whose linear combination is determined by those before it. If addresses were dropped, the auditor picks a solution uniformly at random.

The auditor uses the solution vector as the auditkey vector. When a user retrieves the audit key for address a, they use the k hash functions to derive the vector indices $\{h_1(a), \ldots, h_k(a)\}$, and k hash functions to derive their weights $c_1(a), \ldots, c_k(a)$. These define the linear combination that is the audit key:

$$\mathsf{AK}_a = c_1 \cdot V[h_1(a)] + c_2 \cdot V[h_2(a)] + \ldots + c_k \cdot V[h_k(a)]$$

where the operations are elliptic curve point-addition and scalar multiplication (see Figure 2).

The probability that a set of different addresses cannot be all tracked, given V reduces to the probability that the system of equations contains a linear dependency. In §7, we show that this probability, for any set of size at most $|V|/2$, even adversarially chosen with knowledge of the hash functions, can be made negligibly small by choosing a higher parameter k (number of hash functions), addressing the security problem. For addresses chosen independently of the hash functions the same holds for sets of size up to $|V|$. A larger k means that generating the proof that a withdrawal transaction uses the correct audit key is computationally more expensive (growing linearly in k). However,

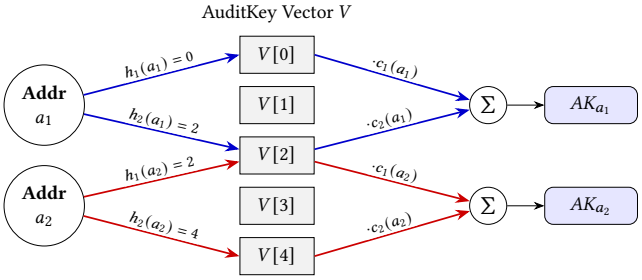


Figure 2: AuditKey Vector encoding example with $|V| = 5$ and $k = 2$ hash functions. Up to $|S| = 5$ addresses can be tracked, but only a_1 and a_2 are shown for clarity. The auditor chooses audit keys AK_{a_1}, AK_{a_2} and populates V, for monitored addresses $S = \{a_1, a2\}$ by solving the linear equation system shown at the bottom for $V[\cdot]$. Each tracked address a_i defines an equation relating its audit key AK_{a_i} to a weighted sum of its vector entries at its k mapped locations. In this example, despite the collision at $V[2]$, the auditor can find a solution and construct an auditkey vector to monitor both addresses.

we show that the computational cost is reasonable (§9.2) and importantly, no matter how large k is, each address maps to just one audit key, which mitigates the on-chain costs.

6 Selective and Restricted Auditing

AuditPay works in discrete epochs. At the beginning of each epoch, the auditor updates the set of tracked addresses S. Algorithm 1 illustrates how we adapt a standard mixer smart contract to support AuditPay's auditing functionality. We now describe its auditing protocol, following the steps in Figure 1. In §7 we formally analyze its security and in §9 we empirically evaluate its performance.

The system is parameterized by the auditor's budget, which is also the size of the vector V, the hash functions for deriving audit keys for addresses (Algorithm 3), and the cryptographic primitives (see §5.1).

Each epoch begins with the auditor updating the set of addresses S. For each address $a \in S$, the auditor generates a per-address *audit key pair* $(sk_a, \mathsf{AK_a})$ through the AuditKey procedure from Algorithm 2. Then, the auditor builds V using the linear equation-based vector construction described in Algorithm 3. The auditor publishes V, and posts its Merkle root com to the mixer smart contract, which commits to the vector. See steps (1a) and (1b) in Figure 1 and the `SetNextAuditKeyVector` procedure in Algorithm 1.

6.1 Deposits

In AuditPay, deposits (step (2) in Figure 1) work as in vanilla (non-auditable) mixers with one extension to the deposited note commitment structure. To facilitate auditing, the depositor address *sender* must be bound to the note commitment so that an auditor

can later (at withdrawal time) learn the correct linkage between *sender* and the recipient's address.

To accommodate this change, AuditPay's smart contract modifies the deposited note commitment to also bind the depositor address *sender* to the note prior to appending to the note Merkle tree. This is done by hashing the original commitment together with *sender*, as shown in the `Deposit` procedure from Algorithm 1.

$$C_{\text{mixer}} \triangleq h(\kappa \,\|\, \rho), \qquad C \triangleq \mathrm{h}(C_{\text{mixer}} \,\|\, sender), \qquad null \triangleq H(\kappa).$$

We refer to C_{mixer} as the *internal commitment*.

6.2 Withdrawals

The withdrawal step (Figure 1 step 3) is when AuditPay's selective auditability is enforced. To withdraw the funds, the client derives the audit keys for both the sender and receiver addresses from V (see the RetrieveKey procedure in Algorithm 3). The client then encrypts the linkage metadata, which is the opening (κ, ρ) of the internal commitment from §6.1, under both keys. Finally, it generates a ZK proof of correct key derivation, linkage data encryption, and knowledge of an unspent deposit's nullifier in the mixer. The withdrawal request to the mixer includes the two ciphertexts, for selectively encrypting the linkage metadata to the auditor's committed keys for both the depositor's *sender* address and recipient's address respectively. If either address is monitored, then the auditor can link the sender and recipient addresses. We now give the ZK relation details for the SNARK attached to the withdrawal.

6.2.1 Proof of Correct Encryption of Linkage Data. AuditPay enforces the correctness of the linkage metadata ciphertexts included with the withdrawal by extending the mixer's ZK proof. This proof is checked by the smart contract's `Withdraw` procedure (Algorithm 1). To allow checking correctness relative to the committed vector V, we include in the proof's private inputs the Merkle paths authenticating the slots $\{V[h_i(sender)]\}_{i=1}^{k}$, $\{V[h_i(recipient)]\}_{i=1}^{k}$. The proof binds the committed com, the ciphertexts, and encryption nonces, and does not reveal whether decryption will succeed. We next describe the details of the relations demonstrated by the ZK proof.

Public inputs. *null*, *recipient*, root (a commitment to the note commitments Merkle tree maintained by the mixer's smart contract); com - this commits to the auditkey vector contents and size; and the two ciphertexts with ephemeral public keys R:

$$(R_{sender}, \mathrm{ct}_{sender}), \quad (R_{recipient}, \mathrm{ct}_{recipient}).$$

Private inputs. κ, ρ, the opening of the internal commitment C, *sender*; Merkle path for C; per-hash function Merkle paths authenticating the slot values $V[\cdot]$ at indices $h_i(sender), h_i(recipient)$ under the current auditkey vector commitment com; ephemeral private keys r_{sender} and $r_{recipient}$.

Relation. Accept iff all the following hold:

(i) **Valid spend:** C is the address-bound commitment, $null = h(\kappa)$, and $\mathsf{MerkleVerify}(C, \mathsf{root}, \mathsf{path}) = \mathbf{True}$.

(ii) **Vector-anchored encryption correctness:** For each $a \in \{sender, recipient\}$, the circuit:

(1) Recomputes positions $\mathrm{pos}_{a,i} = h_i(a)$ from the committed parameters, and verifies Merkle openings of $V_{a,i} = V[\mathrm{pos}_{a,i}]$ under the auditkey vector commitment com.

(2) Uses exactly these opened values $V_{a,i}$ and verifies the public ct_a is a correct selective encryption of the committed $m = (\kappa, \rho)$ under the following keys:

$$R_a = r_a \cdot G, \qquad K_a = r_a \cdot \left(\sum_{i=1}^{k} \left(c_{a,i} \cdot V_{a,i} \right) - H(a) \right),$$

If these checks pass, the mixer smart contract accepts the withdrawal and moves the funds to the recipient address (step 4 in Figure 1).

6.3 Auditing Payments

The auditor monitors the blockchain for withdrawal transactions from the mixer contract (step 5 in Figure 1). For each withdrawal, the auditor attempts to decrypt each of the ciphertexts published as part of the withdrawal, using their secret audit keys sk_a for all $a \in S$, following the `Decrypt` procedure from Algorithm 2. If the ciphertext was encrypted under an address in S, one of the decryption attempts succeeds and reveals the linkage metadata (κ, ρ) to the auditor. This allows the auditor to derive the nullifier $H(k)$ and the internal commitment C. By matching these derived values against the public nullifier revealed in the withdrawal transaction and the commitment stored in the deposit tree, the auditor definitively links the withdrawal to its original deposit. Conversely, if the address is not in S, the ciphertext remains computationally indistinguishable from random, yielding no information to the auditor.

6.4 AuditKey Vector Availability

Our design has users prove that their withdrawal is valid against a commitment to the auditkey vector that the auditor posts on-chain. A rogue auditor may post this commitment, but refuse to publish the underlying vector. Such an attack would lock all user funds in the mixer. AuditPay addresses this issue by extending the ZKP above. Instead of requiring that relation (ii) in the ZKP above must hold, which requires proving correct encryption with respect to the auditkey vector, we require that either relation (ii) holds or (iii) that the recipient's address matches the sender's committed address. This allows a user to always withdraw their funds back to their account.

Orthogonal to this solution, an instantiation of AuditPay can integrate data availability solutions. One option is to post the vector itself to the mixer smart contract. Since each vector entry contains just one elliptic curve point (typically 32B for 128-bit security), directly posting a vector with 1K entries (32KB) may be feasible for a resourceful auditor. Alternatively, AuditPay could integrate with data availability solutions where a committee of validators makes sure that the public can recover data committed on chain [40].

6.5 Mitigating Sybil Attacks

AuditPay's monitoring does not explicitly protect against Sybil attacks [14], where an adversarial user creates multiple addresses in the hopes that some of those addresses will not be monitored. Spamming accounts is a common problem in blockchain

systems [10]. In our context, a simple mitigation may be to require an address registration fee before a user can use that address with the mixer (to withdraw or deposit funds) following the method described in [8]. This ensures that interacting with the system via multiple addresses comes at a substantial cost.

We also argue that in practice, it is sufficient for the auditor to trace a single payment between a depositor and a withdrawer to link between them. Often, a single linkage is sufficient to decipher the full transaction graph between two users in context [31]. AuditPay's security properties ensure that users cannot learn which addresses are being traced at every epoch, so (corrupt) users cannot request payments to addresses that they know are unmonitored. Thus, even if the auditor only monitors a subset of a user's addresses and links only a subset of the user's transactions, this can still provide significant value in tracing the user's overall transaction graph.

New addresses waiting-period. A related challenge is that an adversarial user may register a new address, before the epoch boundary, when the auditor has a chance to include it in its monitor list. This issue can be addressed by imposing a delay between the time an address registers and when it can be used with the mixer smart contract.

6.6 Monitoring by Deposit-Time Address Sets

We presented AuditPay with auditing performed based on the auditkey vector state at the time of withdrawal. We next discuss extending AuditPay to support monitoring based on the auditkey vector state at deposit time. A naive way to do this is to require selective encryption (and a corresponding ZK proof of correctness) of linkage metadata at deposit time relative to the deposit address. However, this would require adding a full zero-knowledge proof at deposit time, which is relatively expensive in gas costs.

However, this goal can be achieved more efficiently by delaying verification of the deposit-time monitoring until withdrawal time. When making a deposit, AuditPay's smart contract adds to the deposit note a commitment to the deposit-time audit set. This can be done by hashing the current auditkey vector root into the deposit note commitment, thus binding the deposit to the current audit set. At withdrawal time, the zero-knowledge proof ensures the deposit address-related proof of encryption to be relative to the deposit time root.

7 Analysis

We now present the security games and theorems for AuditPay, based on its security goals (§3.2). We present the proofs for all theorems in the full version of the paper [33]; machine-checked Lean proofs of the theorems are available online [34].

7.1 Selective Linkability

Selective linkability requires that an auditor can only learn a note associated with a deposit-withdrawal pair (and thus link the pair) if at least one of the depositor/withdrawer addresses was in the active monitored set S at withdrawal time (§3.2).

We formally define selective linkability via two cryptographic games. We first define the *Key Indistinguishability* game (denoted IND-Key), which captures the security of the selective encryption scheme (Algorithm 2). This definition models an *adaptive adversary* who may query the system for a polynomial number of audit keys and hash values before choosing a specific target pair to attack. Informally, this game captures the requirement that an auditor cannot distinguish the encryption key from line 16 in Algorithm 2 from random when the address used as input for AuditKey (line 7) does not match the address used as input for Encrypt (line 13). This property implies that the auditor learns nothing about the plaintext from the ciphertext since the symmetric encryption procedure AuditPay uses is IND-CPA-secure.

We then define the *Selective Linkability* game (denoted SL-IND), which captures AuditPay corresponding security goal. It builds on the IND-Key game to capture that when both addresses are outside the monitored set, the auditor learns nothing about the deposit-withdrawal linkage, as the shared secrets used for encryption are indistinguishable from random. We then prove that AuditPay meets SL-IND security under the DDH assumption in the random oracle model. Our proof builds on the base mixer's unlinkability property (§4), which unlinks deposits from withdrawals.

Definition 1 (IND-Key security). *The IND-Key game is played between a challenger and a PPT adversary $\mathcal{A}$:*

Setup: *The challenger generates public parameters pp tuple, which consists of the instantiation of the basic cryptographic primitives described in §5.1 with security parameter λ. It then gives pp to the adversary $\mathcal{A}$.*

Query: *$\mathcal{A}$ may adaptively query the random oracle H (from pp) that hashes addresses to the curve $\mathbb{G}$. Finally, $\mathcal{A}$ submits to the challenger two addresses a, a^* such that $a^* \neq a$.*

Challenge: *The challenger responds with a challenge generated as follows (running the shared key derivation steps from Selective Encrypt function, with the address a^* that mismatches the audit key's address a):*

1. *The challenger samples an audit key for a: $sk, AK_a \leftarrow$ $\texttt{AuditKey}(a)$*
2. *The challenger samples an ephemeral scalar $r \in_R \mathbb{Z}_q$, and computes the corresponding commitment $R = r \cdot G$*
3. *The challenger computes the "shared" encryption key $K_0 = r \cdot (AK_a - H(a^*))$. This is an execution of lines 13- 16 from Algorithm 2.*
4. *The challenger samples a random key $K_1 \leftarrow_R \mathbb{G}$.*
5. *The challenger flips a fair coin $b \in \{0, 1\}$, computes $R = r \cdot G$ and gives the challenge tuple (sk, AK_a, R, K_b) to $\mathcal{A}$*

Guess: *$\mathcal{A}$ outputs b'.*

Winning Condition: *$\mathcal{A}$ wins the game if $b' = b$.*

The scheme is IND-Key secure if the advantage of $\mathcal{A}$ over random guessing the bit is negligible in the security parameter λ:

$$\mathbf{Adv}_{\mathcal{A}}^{\textit{IND-Key}}(\lambda) = \left| \Pr[\mathcal{A}(pp, sk, AK_a, R, K_b) = b] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)$$

where $\mathsf{negl}(\cdot)$ *denotes a function that diminishes faster than the inverse of any polynomial.*

Theorem 1. Let $\mathbb{G}$ be a group where DDH holds. The selective encryption scheme defined in Algorithm 2 is adaptively IND-Key secure in the Random Oracle Model over $\mathbb{G}$.

Corollary 1.1. Under the random oracle model and the hardness of DDH assumption, Algorithm 2's shared secret K is computationally

indistinguishable from random when the audit key is created for an address that does not match the address encrypted against.

We now formally define the *Selective Linkability* (SL-IND) game. This game captures the requirement that an auditor may link a deposit-withdrawal pair only if at least one of the involved addresses (sender or recipient) is in the monitored set used to construct the auditkey vector at withdrawal time. (We note that by construction of AuditPay's selective encryption, if an address was encoded in V then any transaction related to it is linkable.)

Definition 2 (SL-IND security). *The SL-IND game is played between a challenger and a PPT adversary $\mathcal{A}$ taking the role of the auditor:*

Setup: *The challenger generates public parameters pp with security parameter λ (described in §5.1) and gives pp to the adversary $\mathcal{A}$.*

Commitment: *(1) $\mathcal{A}$ selects a list of monitored addresses S, in priority order. The challenger generates an auditkey vector V by running* `AuditKeyVecGen` *from Algorithm 3 on S and gives its output, and the random coins it used, to $\mathcal{A}$.*

(2) $\mathcal{A}$ chooses two distinct deposit addresses d_0, d_1 and a different withdrawal address w that `AuditKeyVecGen` *did not encode in V. Since these addresses are not monitored, $\mathcal{A}$ should not be able to distinguish encryptions to audit keys derived from V for these addresses from random values.*

(3) $\mathcal{A}$ gives d_0, d_1, w to the challenger.

Challenge: *The challenger creates deposit transactions originating from addresses d_0 and d_1 and gives those deposit transactions, dtx_0, dtx_1 to $\mathcal{A}$. The challenger then flips a fair coin $b \in \{0,1\}$ and creates a* withdrawal *transaction from address d_b to address w:*

(1) The withdrawal transaction includes two ciphertexts $ct_{sender}, ct_{recipient}$ for the audit keys derived from the auditkey vector V (via the `RetrieveKey` *from Algorithm 3). It also includes the zero-knowledge proof π proving correctness of the encryption.*

(2) The challenger returns the withdrawal transaction, wtx_b, to $\mathcal{A}$.

Guess: *$\mathcal{A}$ outputs a guess b' as to the originating address of the withdrawal.*

Winning Condition: *The adversary wins the game if $b' = b$.*

A system is SL-IND secure if the advantage of $\mathcal{A}$ is negligible in the security parameter λ:

$$\mathbf{Adv}^{SL\text{-}IND}_{\mathcal{A}}(\lambda) = \left| \Pr[\mathcal{A}(pp, S, V, d_0, d_1, w, dtx_0, dtx_1, wtx_b) = b] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)$$

Theorem 2. AuditPay is SL-IND secure in the Random Oracle Model, assuming that the ZK-SNARK is zero-knowledge, the base mixer is unlinkable, the symmetric encryption scheme is IND-CPA secure, and DDH holds over $\mathbb{G}$.

7.2 Restricted Monitoring

We formally define the Restricted Monitoring property via a game. This game captures the guarantee that given an auditkey vector V of size $|V|$, an auditor can monitor at most $|V|$ addresses, and that this set of addresses is determined once V is published. The game asks the auditor for the shared encryption keys: an auditor that decrypts a ciphertext must query the KDF (a random oracle) on its shared key.

Definition 3 (Restricted Monitoring). *The Restricted Monitoring game is played by a PPT adversary $\mathcal{A}$ against a challenger:*

Setup: *The challenger generates public parameters pp with security parameter λ as described in §5.1 and gives them to $\mathcal{A}$.*

Commitment: *$\mathcal{A}$ outputs a public vector V of size $|V|$ and a target set of addresses $S^* = \{a_1, \ldots, a_{|V|+1}\}$.*

Challenge: *For each address $a \in S^*$, the challenger samples $r_a \in_R \mathbb{Z}_q$ and gives $R_a = r_a \cdot G$ to $\mathcal{A}$, as in Algorithm 2's* `Encrypt` *procedure.*

Response: *$\mathcal{A}$ returns to the challenger a key K_a for each address $a \in S^*$.*

Win Condition: *$\mathcal{A}$ wins if every K_a is the shared encryption key for a (line 16 in Algorithm 2), i.e., $K_a = r_a \cdot (\mathsf{RetrieveKey}(V, a) - H(a))$.*

AuditPay satisfies Restricted Monitoring if the probability of $\mathcal{A}$ winning is negligible in λ.

Theorem 3. In the Random Oracle Model, if DDH holds in $\mathbb{G}$, AuditPay satisfies Restricted Monitoring as defined in Definition 3.

7.3 Membership Hiding

We define Membership Hiding via two games. The first captures the inability of a user to learn from the auditkey projection vector V anything about the addresses used to generate it, and the second the inability of users to craft addresses that the auditor cannot monitor together.

Definition 4 (Membership Hiding). *The Membership Hiding game is played by a challenger and a PPT adversary $\mathcal{A}$:*

Setup: *The challenger generates public parameters pp as described in 5.1 (including access to the random oracles) and gives them to $\mathcal{A}$.*

Query: *$\mathcal{A}$ may adaptively query the random oracles (the hash-to-curve function H and the hash functions h_i, c_i of Algorithm 3) on a polynomial number of addresses a_i.*

Commitment: *The adversary $\mathcal{A}$ selects an auditkey vector size $|V| \geq 1$ and two candidate lists of addresses S_0 and S_1, each in priority order and with $|S_0|, |S_1| \leq |V|$, and submits them to the challenger along with $|V|$.*

Challenge: *The challenger flips a fair coin $b \in \{0,1\}$ and generates a vector V of size $|V|$ by running Algorithm 3's* `AuditKeyVecGen` *procedure on S_b. The vector V is given to $\mathcal{A}$.*

Guess: *$\mathcal{A}$ outputs a guess b' as to which list of addresses was used to create V.*

Win Condition: *$\mathcal{A}$ wins if $b' = b$.*

A system is Membership Hiding if $\left|\Pr[b' = b] - \frac{1}{2}\right| \leq negl(\lambda)$.

Definition 5 (Monitoring Coverage). *The Monitoring Coverage game is played by a challenger and a PPT adversary $\mathcal{A}$, with the Setup and Query phases of Definition 4:*

Output: *$\mathcal{A}$ outputs an auditkey vector size $|V| \geq 1$ and a list S of at most $|V|$ addresses, in priority order.*

Win Condition: *$\mathcal{A}$ wins if* `AuditKeyVecGen`*, run on S, drops one of the first $\lfloor |V|/2 \rfloor$ addresses of S.*

A system provides Monitoring Coverage if the probability of $\mathcal{A}$ winning is negligible in λ.

Theorem 4. In the Random Oracle Model:

(1) AuditPay is Membership Hiding. Moreover, for every list S the vector V that AuditKeyVecGen outputs is uniformly distributed over $\mathbb{G}^{|V|}$, so the advantage of $\mathcal{A}$ is zero.
(2) AuditPay provides Monitoring Coverage for $k = \omega(\log \lambda)$. We give the concrete bound on the winning probability of $\mathcal{A}$ in §7.4.

7.4 The Probability of Dropping a Monitored Address

AuditPay's auditing mechanism relies on `AuditKeyVecGen` encoding the addresses that the auditor chooses. An address is dropped only when its equation is linearly dependent on the equations of the addresses before it. Let us consider the matrix M whose rows are the equations of the addresses in S and evaluate the probability that its rows are linearly dependent. We do so in two settings, which give two guarantees, and use them to inform the parameter choice for AuditPay's implementation (§8).

Each row i of M is filled in by randomly selecting k positions (induced by the address $a_i \in S$) and assigning each position a random coefficient from $\mathbb{Z}_q$ (since AuditPay uses k position and k coefficient hash functions, modeled as random oracles).

Addresses chosen independently of the hash functions. We first consider addresses that were not crafted using the hash functions. We assume $|S| = |V|$, since this case imposes the most restrictions, so M is a random k-sparse square matrix over $\mathbb{Z}_q$. The probability of such a matrix being singular has been studied in prior work [11, 12], and depends on two types of singularities:

structural singularities, where some t rows have all their positions in fewer than t columns.

algebraic singularities, where rows are linearly dependent although the matrix has no structural singularity.

Algebraic singularities happen with probability at most $|V|/q$ [34]. Structural singularities are dominated by the case of an empty column, which happens with probability $|V|e^{-k}$. Solving for k such that $|V|e^{-k} \le 2^{-128}$ gives us a lower bound on k. For example, for $|V| = 10^6$:

$$\begin{aligned} 10^6 \cdot e^{-k} &\le 2^{-128} \\ \ln(10^6) - k &\le -128 \ln(2) \\ k &\ge \ln(10^6) + 128 \ln(2) \ge 103 \end{aligned}$$

Meaning that for an auditkey projection vector of size $|V| = 10^6$, using $k = 103$ hash functions, the auditor can use its whole budget: all $|V|$ addresses are encoded except with probability about $2^{-128} + 10^6/q \approx 2^{-128}$ (since q, the order of $\mathbb{G}$, is about 251 bits).

Addresses chosen adversarially. Users can evaluate the hash functions, so colluding users may search for addresses whose equations are dependent, so that the auditor cannot monitor all of them. For sets of size close to $|V|$ this is easy: for example, users can pick $|V|$ addresses none of which maps to the first entry of V, and at most $|V| - 1$ of them can be encoded. For smaller sets it is hard: for t addresses to have a structural dependency all their positions must fall in fewer than t entries of V, and an address has all its k positions in a given half of V with probability 2^{-k}. Formally, for an adversary that evaluates the hash functions on at most Q addresses, the probability that some set of at most s of these addresses has linearly dependent equations is at most [34]

$$\sum_{t=1}^{s} \binom{Q}{t} \left[\binom{|V|}{t-1} \left(\tfrac{t-1}{|V|}\right)^{kt} + \sum_{u=t}^{|V|} \binom{|V|}{u} \left(\tfrac{u}{|V|}\right)^{kt} q^{t-1-u} \right],$$

where the first term accounts for structural dependencies and the second for algebraic ones. For $|V| = 10^6$, $k = 103$ and $s = |V|/2$ this bound is below 2^{-128} for an adversary that evaluates up to 2^{118} addresses. The bound degrades quickly for larger sets, which is why Monitoring Coverage (Definition 5) is stated for the first $|V|/2$ addresses.

The auditor may list up to $|V|$ addresses. Those after the first $|V|/2$ are encoded with probability about $1 - 2^{-128}$ when no listed address was crafted, but may be dropped if earlier addresses were crafted, hence the priority order.

8 Implementation

We implemented AuditPay as an extension of Tornado Cash v2.0.0, a well-known mixer [29] written in 2020. Our implementation has three main components. The first is the mixer smart contract written in Solidity. As part of our implementation, we also migrated the Tornado tool-chain to be current.

The second is the client-side, which creates deposit and withdrawal transactions (written in Node-JS), where the withdrawal transaction carries a zero-knowledge proof that demonstrates the withdrawal is valid. AuditPay extends Tornado Cash's ZK proof (written in Circom) to include the relations discussed in §6. It additionally modifies the deposit and withdrawal functions to handle the new commitment structures necessary at deposit and additional fields sent during withdrawal.

Finally, we implemented in C++ the auditor's protocol, which takes a set of monitored addresses, encodes audit keys for them in an auditkey vector (§5.2) and posts the commitment to that vector to the mixer's smart contract at every epoch.

Auditkey vector encoding. The audit keys encoded in the auditkey projection vector are ECC points over the BabyJubJub curve. We use the analysis from §7.4 to inform our implementation's parameter choice. We see that for $k = 100$ hash functions, the encoding scheme has over 110 bits of security against users crafting a set of addresses, of up to half the vector's size, that cannot be monitored together, for auditkey vectors of up to 1M entries. Our implementation uses the Poseidon hash function to instantiate those hash functions; this allows the user withdrawing from the mixer to efficiently prove in zero knowledge that they encrypt to the correct audit key (identified by a linear combination of points induced by the hash functions).

We implemented LU-decomposition [13, Chapter 28] over the BabyJubJub curve field [36], and used it solve the linear equations induced by the monitored address set, the audit keys, and the hash functions we picked. This allowed for significant parallelization when building the auditkey projection vector, allowing AuditPay to support auditors with relatively high computational resources.

8.1 Efficient zero knowledge proofs

Our zero-knowledge proof circuit implementation uses Groth16 [18] with the BabyJubJub elliptic curve. We use Poseidon [17] as the hash function throughout: for the Merkle tree holding the deposit commitments, for encoding the auditkey vector (described above), and for hashing scalars to the BabyJubJub curve. Poseidon is efficient to verify in zero knowledge, and for the on-chain deposit tree it is also cheaper in gas than the MiMC hash [1] used by Tornado Cash, since each tree level takes one call to the hash function instead of two: a Tornado Cash deposit costs $809K$ gas with the Poseidon tree against $1.06M$ with MiMC. We assume preimage resistance and collision resistance of the Poseidon hash function, and model the Poseidon sponge construction as a random oracle (due to the hash-to-curve use). These assumptions are supported by the security analysis in [17]. We use rapid-snark [3] to generate zk-SNARKs and their public verification keys. We use Circom [5] to write the arithmetic circuits for the ZKPs.

Folding public inputs. Verifying a Groth16 proof on-chain costs one elliptic-curve scalar multiplication per public input, and AuditPay's withdrawal proof naturally has 14: Tornado Cash's Merkle root, nullifier hash, recipient and relayer data, plus the auditkey vector commitment, the nonce key, and the two ciphertexts (three field elements each) with their ephemeral public keys. We fold the 11 values other than the root, the nullifier hash and the auditkey vector commitment into a single SHA-256 digest, which the circuit computes and exposes as one public input and the contract recomputes from the calldata, leaving four public inputs. This cuts the withdrawal gas cost from 370K to 306K. The price is paid by the prover: the in-circuit SHA-256 adds about 190K constraints (2.08M to 2.28M for $k = 100$ and an 8k auditkey vector), or about 3 seconds of proving time (9.7s to 12.5s). Soundness of the fold additionally assumes collision resistance of SHA-256.

We also implement symmetric encryption using Poseidon with a secret input as key, as the underlying permutation, following the construction in [6]. This lets us efficiently prove correct symmetric encryption of the linkage data under the audit key in zero knowledge.

9 Evaluation

We evaluate AuditPay using our prototype implementation. We run client-side code on a laptop machine with an Intel Core Ultra 7 155U CPU @ 2.70GHz and 32GB of RAM allocated on WSL2. We run the auditor's code on an AMD EPYC 9555 CPU @ 3.2GHz machine with 100 cores and 800GB RAM allocated, since we assume the auditor affords relatively high-end hardware.

We use Ganache as the blockchain back-end when evaluating gas costs for using the mixer smart contract. Our Tornado Cash baseline runs the Tornado Cash smart contract code with the same library versions as our prototype and, like AuditPay, uses a Poseidon hash for the deposit Merkle tree in place of Tornado Cash's MiMC hash (§8), so that the measured overhead isolates the cost of auditing rather than the choice of hash function.

Our evaluation quantitatively answers the following questions:

(1) What are the gas costs of using AuditPay to deposit and withdraw coins to/from the mixer, how do these costs compare to the base Tornado mixer, and how do they scale with the auditing budget?

(2) What is the proving cost for creating withdrawal transactions, and how does it scale with the auditing budget? (Computational cost for users.)

(3) What is the cost of creating AuditPay's auditkey vector? (Computational cost for the auditor.)

(4) How do AuditPay's costs compare to naive alternatives for implementing selective encryption?

9.1 Gas Costs

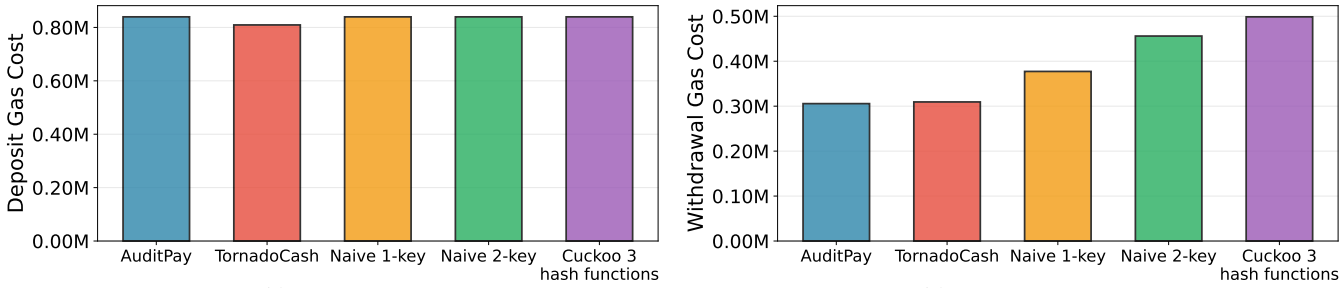
Our proof system uses one public input for a commitment to the whole auditkey vector and the hash functions used to construct it. This means that the deposit and withdrawal gas costs are independent of the size of the auditkey vector or the number of hash functions. Figure 3 compares these costs against the Tornado Cash baseline described above for evaluating AuditPay's overhead. We next analyze AuditPay's cost overhead.

Deposit. AuditPay's deposits are identical to Tornado Cash deposits with one additional Poseidon hash computation. The additional Poseidon hash is used to relate the depositor's address to the note commitment being deposited, as discussed in §6.1. The deposit gas cost is the Tornado Cash gas cost ($809K$) plus the cost of hashing the depositor's address together with the note commitment using the Poseidon hash (approximately $30K$ gas), for a total of $840K$ gas (3.7% overhead, Figure 3a). The above statement holds for all audit set sizes and cuckoo hashing parameters, as deposits do not depend on these parameters.

Withdrawal. At withdrawal time, AuditPay extends Tornado Cash's ZK proof and parameters. ZKP verification costs comprise a fixed cost for verifying a SNARK plus a variable cost that scales linearly with the number of public inputs to the SNARK; AuditPay folds its public inputs into a single hash (§8), so it has no more public inputs than Tornado Cash. Tornado Cash's baseline cost is 309K gas while AuditPay's cost is 306K gas (about 1% less, Figure 3b).

Comparison with naive alternatives. We compare AuditPay's gas costs with two simple alternative schemes for auditing multiple addresses. The first is the baseline approach that encrypts each withdrawal transaction linkage metadata to all audit keys. We consider monitoring with either one or two audit keys. In our evaluation, we can see that AuditPay's cost is comparable to this naive approach with a single audit key (i.e., an auditing budget of one address): 306K versus 377K gas per withdrawal, where most of the difference is due to AuditPay folding its public inputs into a single hash (§8). The baseline withdrawal costs increase linearly with the number of audit keys; each new key induces a 79k gas cost. Thus, the naive method does not scale well to support thousands of monitored addresses. For example, to support an auditing budget of just 1000 addresses, the baseline withdrawal gas cost would be approximately $79M$ gas, about 255× higher than Tornado Cash.

The second alternative we consider is the cuckoo hash-based technique (§5.2.1). It addresses the gas cost issue of the baseline, as



(a) Deposit gas **(b) Withdrawal gas**

Figure 3: Deposit and withdrawal gas cost comparison

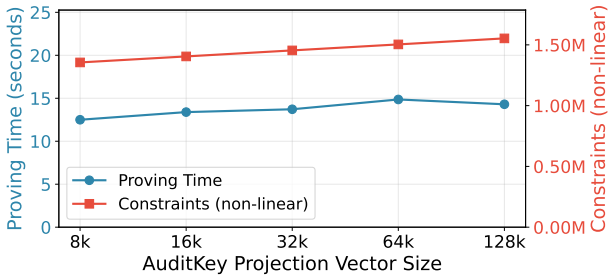


Figure 4: Withdrawal proving time and ZKP constraints as a function of the auditor's budget.

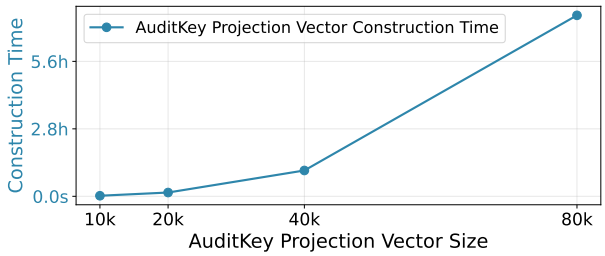


Figure 5: AuditKey vector computation time.

it always encrypts an address to a fixed number of audit keys (3 in our experiment), but it does not satisfy the membership hiding property. Still, we see that it has a higher cost than AuditPay (which only encrypts the linking metadata to a single key).

9.2 Withdrawal Transaction Proving Cost

At withdrawal time, the user needs to generate a zero-knowledge proof that their transaction is valid (spends an unspent deposit), and that they encrypt the correct linkage metadata using the correct audit key (§6.2.1). The former part is similar to Tornado Cash, while the latter is new to AuditPay and grows with the auditor's budget.

In Figure 4 we compare the proving cost, both in terms of time (on our laptop machine, see above) and number of ZKP constraints. We focus on non-linear constraints since those dominate the proof cost. We see that proving overhead grows logarithmically with the number of addresses the auditor can track (the size of the auditkey vector); this allows the cost of generating withdrawal proofs to scale gracefully even for large auditing budgets. The logarithmic growth is due to the multiple Merkle proofs of membership in AuditPay's auditkey vector, whose cost increases logarithmically with the vector size. Each new level in this Merkle tree adds about 0.5s to the proving time, which concludes in under 15s, even when the auditor's budget exceeds a hundred thousand addresses (for comparison, Ethereum's block time is about 12s, which is the same order of magnitude).

9.3 AuditKey Vector Creation Compute Cost

The auditor needs to solve a system of linear equations to encode its audit keys into the auditkey vector. Our implementation uses the LU-decomposition technique. While this operation has sequential dependencies, it is amenable to block-based parallelization. We assume that an auditor can exploit this parallelism in practice by running this computation on a machine with many processing cores. Thus, our experiments use a high-end machine, see above (albeit, just one instance of it).

We observe that the auditor's computation grows roughly cubically with the auditing budget (Figure 5). An auditor monitoring $40k$ addresses needs about an hour to generate the auditkey vector. For reference, the OFAC list [27] of addresses (controlled by the US government) contains about $18.5k$ records.

10 Conclusion

This paper presented AuditPay, the first mixer that supports a restricted auditor, giving important privacy guarantees to both sides: users know that the auditor has a limited budget, and the auditor knows that its set of monitored addresses is hidden from the users. We introduce a novel technique for selective encryption that efficiently encodes audit keys for monitored addresses in an auditkey vector. Experiments on a prototype implementation of AuditPay show that it has small overhead costs over a baseline mixer, making AuditPay a practical solution for balancing payment privacy and auditability.

Acknowledgments

We thank the CCS'26 reviewers for their comments, suggestions and feedback, which have helped improve this paper. We also thank Yoav Moran for his helpful comments on implementation and overhead of matrix operations. This work was supported by Israel Science Foundation grants no. 1104/23 and 1443/21. Aviv Zohar was also supported by DeCenter at Princeton, whose hospitality is gratefully acknowledged. The paper was edited for grammar using Gemini.

References

[1] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology – ASIACRYPT 2016, Part I (Lecture Notes in Computer Science, Vol. 10031)*. 191–219. https://doi.org/10.1007/978-3-662-53887-6_7

[2] Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. 2020. Privacy-preserving auditable token payments in a permissioned blockchain system. In *2nd ACM Conference on Advances in Financial Technologies, AFT 2020*. 255–267. https://doi.org/10.1145/3419614.3423259

[3] Jordi Baylina and iden3 Team. 2021. Rapidsnark: Fast zkSNARK prover written in C++ and Assembly. https://github.com/iden3/rapidsnark.

[4] Josh Beal and Ben Fisch. 2024. Derecho: privacy pools with proof-carrying disclosures. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3197–3211.

[5] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Access* 10 (2022), 101–118. https://doi.org/10.1109/ACCESS.2021.3138865

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. Duplexing the sponge: single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography (SAC 2011)*. Springer, 320–337.

[7] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-policy attribute-based encryption. In *2007 IEEE symposium on security and privacy (SP'07)*. IEEE, 321–334.

[8] George Bissias, A Pinar Ozisik, Brian N Levine, and Marc Liberatore. 2014. Sybil-resistant mixing for bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. 149–158.

[9] Dan Boneh and Matt Franklin. 2001. Identity-based encryption from the Weil pairing. In *Annual international cryptology conference*. Springer, 213–229.

[10] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. 2015. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE symposium on security and privacy*. IEEE, 104–121.

[11] Neil J Calkin. 1996. Dependent sets of constant weight vectors in GF (q). *Random Structures & Algorithms* 9, 1-2 (1996), 49–53.

[12] Colin Cooper. 2000. On the rank of random matrices. *Random Structures & Algorithms* 16, 2 (2000), 209–232.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (4th ed.). MIT Press.

[14] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.

[15] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious Key-Value Stores and Amplification for Private Set Intersection. In *Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II*. 395–425. https://doi.org/10.1007/978-3-030-84245-1_14

[16] Christina Garman, Matthew Green, and Ian Miers. 2016. Accountable Privacy for Decentralized Anonymous Payments. In *Financial Cryptography and Data Security, FC 2016*. 81–98. https://doi.org/10.1007/978-3-662-54970-4_5

[17] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. 2023. Poseidon2: A Faster Version of the Poseidon Hash Function. In *Progress in Cryptology - AFRICACRYPT 2023: 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19–21, 2023, Proceedings* (Sousse, Tunisia). 177–203. https://doi.org/10.1007/978-3-031-37679-5_8

[18] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). 305–326.

[19] Daira Hopwood, Sean Bowe, Taylor Hornby, Nathan Wilcox, et al. 2016. Zcash protocol specification. *GitHub: San Francisco, CA, USA* 4, 220 (2016), 32.

[20] S. Horowitz. 2024. U.S. Court Overrules Tornado Cash Sanctions. https://s-horowitz.com/u-s-court-overrules-tornado-cash-sanctions.

[21] Aggelos Kiayias, Markulf Kohlweiss, and Amirreza Sarencheh. 2022. PEReDi: Privacy-Enhanced, Regulated and Distributed Central Bank Digital Currencies. In *ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*. 1739–1752. https://doi.org/10.1145/3548606.3560707

[22] Hannah Lang. 2025. US scraps sanctions on Tornado Cash, crypto 'mixer' accused of laundering North Korea money. https://www.reuters.com/business/finance/us-scraps-sanctions-tornado-cash-crypto-mixer-accused-laundering-north-korea-2025-03-21.

[23] Yuxian Li, Jian Weng, Junzuo Lai, Yingjiu Li, Jiahe Wu, Ming Li, Jianfei Sun, Pengfei Wu, and Robert Deng. 2024. AuditPCH: Auditable Payment Channel Hub with Privacy Protection. *IEEE Transactions on Information Forensics and Security* 20 (2024), 1251–1261.

[24] Pedro Moreno-Sanchez, Mohsen Minaei, Srinivasan Raghuraman, Panagiotis Chatzigiannis, and Duc V. Le. 2025. AUPCH: Auditable Unlinkable Payment Channel Hubs. Cryptology ePrint Archive, Paper 2025/1524. https://eprint.iacr.org/2025/1524

[25] Satoshi Nakamoto and A Bitcoin. 2008. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf* 4, 2 (2008), 15.

[26] Shen Noether. 2015. Ring SIgnature Confidential Transactions for Monero. *IACR Cryptology ePrint Archive* 2015 (2015), 1098.

[27] Office of Foreign Assets Control. 2026. Specially Designated Nationals and Blocked Persons List (SDN). https://sanctionslist.ofac.treas.gov/Home/SdnList. Accessed: 2026-01-12.

[28] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 121–133.

[29] Alexey Pertsev, Roman Semenov, and Roman Storm. 2019. Tornado cash privacy solution version 1.4. *Tornado cash privacy solution version* 1, 6 (2019).

[30] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: Fast, Malicious Private Set Intersection. In *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II* (Zagreb, Croatia). 739–767. https://doi.org/10.1007/978-3-030-45724-2_25

[31] Dorit Ron and Adi Shamir. 2013. Quantitative analysis of the full bitcoin transaction graph. In *International conference on financial cryptography and data security*. Springer, 6–24.

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

[33] Elkana Tovey, Yossi Gilad, and Aviv Zohar. 2026. AuditPay: Anonymous Payments with Controlled Oversight. Cryptology ePrint Archive, Paper 2026/1118. https://eprint.iacr.org/2026/1118

[34] Elkana Tovey, Yossi Gilad, and Aviv Zohar. 2026. AuditPay: Machine-Checked Security Proofs in Lean. GitHub repository. https://github.com/elkanatovey/auditpay_proofs

[35] Nicolas Van Saberhagen. 2013. CryptoNote v2.0. (2013). "Viewing keys for Monero".

[36] Barry WhiteHat, Jordi Baylina, and Marta Bellés. 2020. Baby Jubjub elliptic curve. *Ethereum Improvement Proposal, EIP-2494* 29 (2020), 27.

[37] Karl Wüst, Kari Kostiainen, Vedran Capkun, and Srdjan Capkun. 2019. PRCash: Fast, Private and Regulated Transactions for Digital Currencies. In *Financial Cryptography and Data Security, FC 2019*. 158–178. https://doi.org/10.1007/978-3-030-32101-7_11

[38] Karl Wüst, Kari Kostiainen, Noah Delius, and Srdjan Capkun. 2022. Platypus: A Central Bank Digital Currency with Unlinkable Transactions and Privacy-Preserving Regulation. In *ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*. 2947–2960. https://doi.org/10.1145/3548606.3560617

[39] Kevin Yeo. 2023. Cuckoo Hashing in Cryptography: Optimal Parameters, Robustness and Applications. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV* (Santa Barbara, CA, USA). 197–230. https://doi.org/10.1007/978-3-031-38551-3_7

[40] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. 2020. Coded merkle tree: Solving data availability attacks in blockchains. In *International Conference on Financial Cryptography and Data Security*. Springer, 114–134.

A Security Proofs

We next prove the theorems from §7. Machine-checked Lean proofs of these theorems are available online [34].

A.1 Proof of Theorem 1

Proof. We show that for any PPT adversary $\mathcal{A}$ achieving advantage ϵ in the IND-Key game, there exists a PPT algorithm $\mathcal{B}$ that solves the DDH problem with advantage at least $\epsilon - 1/q$.

We prove this via a reduction to the DDH problem in the group $\mathbb{G}$. Suppose there exists a PPT adversary $\mathcal{A}$ that wins the IND-Key game with non-negligible advantage ϵ. We construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to solve the DDH problem. Given a DDH challenge (G, X, Y, Z), $\mathcal{B}$ simulates the IND-Key challenger for $\mathcal{A}$ as follows:

Simulation. $\mathcal{B}$ programs H to embed the DDH challenge into all of the oracle responses. For every address x that $\mathcal{A}$ queries, $\mathcal{B}$ samples $u_x, t_x \leftarrow \mathbb{Z}_q$ and sets $H(x) = u_x \cdot G + t_x \cdot Y$. In particular,

$\mathcal{B}$ does not need to guess which addresses $\mathcal{A}$ will select for the challenge.

$\mathcal{B}$ now simulates the challenger in the IND-Key game for $\mathcal{A}$, using the programmed random oracle defined above. It first passes the public parameters pp to $\mathcal{A}$. When $\mathcal{A}$ submits the addresses a, a^*, $\mathcal{B}$ samples $sk \leftarrow \mathbb{Z}_q$, and sets $AK_a = sk \cdot G + H(a)$, $R = X$ and

$$K = sk \cdot X + (u_a - u_{a^*}) \cdot X + (t_a - t_{a^*}) \cdot Z,$$

sending (sk, AK_a, R, K) to $\mathcal{A}$.

Finally, when $\mathcal{A}$ outputs its guess b', $\mathcal{B}$ outputs the same guess as its own DDH guess.

Analysis. We argue that $\mathcal{B}$ simulates the IND-Key game for $\mathcal{A}$. In the real IND-Key game H's outputs are uniformly random in $\mathbb{G}$. In the simulation, $H(x) = u_x \cdot G + t_x \cdot Y$ is uniformly random for every address x, and moreover reveals nothing about t_x (since u_x is a random scalar, i.e., $u_x \cdot G$ acts as a one time pad). All other cryptographic primitives and procedures are simulated exactly as in the real IND-Key game.

We now show that the key K computed in the simulation embeds the DDH challenge. In the real game, $K_0 = r \cdot (AK_a - H(a^*))$. Let us denote the discrete logs of X and Y by α, β respectively (both are unknown to $\mathcal{B}$). Namely, $\mathcal{B}$ needs to decide whether $Z = \alpha\beta \cdot G$. In the simulation, we implicitly have $r = \alpha$, the discrete log of X (since $R = X = rG$). Substituting the programmed oracle values:

$$\begin{aligned} K_0 &= r \cdot (AK_a - H(a^*)) \\ &= r \cdot (sk \cdot G + H(a) - H(a^*)) \\ &= r \cdot (sk \cdot G + (u_a - u_{a^*}) \cdot G + (t_a - t_{a^*}) \cdot Y) \\ &= sk \cdot X + (u_a - u_{a^*}) \cdot X + (t_a - t_{a^*}) \cdot \alpha\beta \cdot G \end{aligned}$$

The above shows that if $Z = \alpha\beta \cdot G$, then $K = K_0$ and $\mathcal{A}$ receives the correct shared secret. If Z is random, then K differs from K_0 by $(t_a - t_{a^*}) \cdot (Z - \alpha\beta \cdot G)$, so $\mathcal{A}$ receives a random element in $\mathbb{G}$ whenever $t_a \neq t_{a^*}$. Since $a \neq a^*$ and the oracle responses reveal nothing about t_a, t_{a^*}, we have $t_a = t_{a^*}$ with probability $1/q$.

Thus, except with probability $1/q$, $\mathcal{B}$ simulates the IND-Key game perfectly for $\mathcal{A}$. Therefore, $\mathcal{A}$'s advantage ϵ in the IND-Key game translates into an advantage of at least $\epsilon - 1/q$ for $\mathcal{B}$ in solving the DDH problem. □

A.2 Proof of Theorem 2

Proof. We define a sequence of hybrid games Game 0, …, Game 3. Let E_i denote the event that $\mathcal{A}$ wins in Game i.

Game 0: The real SL-IND game.

Game 1: Identical to Game 0, except the ZK proof π is replaced by a simulated proof π_{sim} generated by the ZK Zero-Knowledge Simulator. By the zero-knowledge property of the underlying ZK proof system, such a simulator exists and $\mathcal{A}$ cannot distinguish π from π_{sim}. Thus, $|\Pr[E_1] - \Pr[E_0]| \leq \mathrm{negl}_{ZK}(\lambda)$.

Game 2: Identical to Game 1, except the shared secrets (line 16 in Algorithm 2) used for the ciphertexts $(ct_{sender}, ct_{recipient})$ from line 1 are replaced by independent, random elements in $\mathbb{G}$. We show that $\mathcal{A}$ cannot distinguish the two games under the DDH assumption, replacing one shared secret at a time. This extends the argument of Theorem 1: there, the audit key was created for a single address, while here the key retrieved for a challenge address is a linear combination of entries of V.

Given a DDH challenge (G, X, Y, Z), $\mathcal{B}$ programs $H(x) = u_x \cdot G + t_x \cdot Y$ for every address x, as in the proof of Theorem 1, and runs `AuditKeyVecGen` on S as the challenger does. Every audit key that `AuditKeyVecGen` creates is of the form $sk_x \cdot G + H(x) = (sk_x + u_x) \cdot G + t_x \cdot Y$, and the entries of V are a linear combination of these audit keys and of random elements that $\mathcal{B}$ picks as multiples of G. Thus, for a challenge address a, $\mathcal{B}$ knows scalars μ, ν such that

$$\mathsf{RetrieveKey}(V, a) - H(a) = \mu \cdot G + \nu \cdot Y,$$

where ν is a linear combination of the t_x of the addresses encoded in V, minus t_a. Since a was not encoded in V, t_a does not appear in V, and (as H reveals nothing about t_a) we have $\nu = 0$ with probability $1/q$. $\mathcal{B}$ sets $R = X$ for the ciphertext of a and uses $\mu \cdot X + \nu \cdot Z$ as its shared secret. If $Z = \alpha\beta \cdot G$ this is the correct shared secret, and if Z is random and $\nu \neq 0$ it is a random element in $\mathbb{G}$. Applying this to the sender's and then to the recipient's ciphertext, $|\Pr[E_2] - \Pr[E_1]| \leq 2 \cdot (\mathrm{negl}_{DDH}(\lambda) + 1/q)$.

Game 3: Identical to Game 2, except the ciphertexts are replaced by encryptions of random strings under independent, random symmetric keys $K^*_{sender}, K^*_{recipient}$. In Game 2 the shared secrets are random elements in $\mathbb{G}$ that $\mathcal{A}$ does not receive. Since the KDF is modeled as a random oracle, the derived symmetric keys are random unless $\mathcal{A}$ queries the KDF on one of the shared secrets, which happens with negligible probability. Since the symmetric encryption scheme is IND-CPA secure, the Game 2 ciphertexts are then indistinguishable from the random (Game 3) ciphertexts. Thus, $|\Pr[E_3] - \Pr[E_2]| \leq \mathrm{negl}_{Sym}(\lambda)$.

In Game 3, the transcript $\mathcal{T}$ consists of a simulated proof and random ciphertexts, neither of which depends on the bit b. The only remaining dependency on b may be in the underlying mixer protocol. As such, this game's security reduces to the security of the base mixer protocol. Under the assumption that the base mixer provides unlinkability (as described in §3 and satisfied by the mixer described in §4), $\mathcal{A}$'s advantage in linking the withdrawal w to the specific deposit d_b is negligible. Thus, $|\Pr[E_3] - 1/2| \leq \mathrm{negl}_{Mix}(\lambda)$.

By the triangle inequality, the adversary's advantage in the real game is bounded by the sum of the advantages in the distinguishing steps:

$$\begin{aligned} \left|\Pr[E_0] - \frac{1}{2}\right| \leq\ & |\Pr[E_0] - \Pr[E_1]| \\ &+ |\Pr[E_1] - \Pr[E_2]| \\ &+ |\Pr[E_2] - \Pr[E_3]| \\ &+ \left|\Pr[E_3] - \frac{1}{2}\right| \end{aligned}$$

Substituting the bounds established in the previous games:

$$\begin{aligned} \mathbf{Adv}^{\text{SL-IND}}_{\mathcal{A}} \leq\ & \mathrm{negl}_{ZK}(\lambda) + 2 \cdot (\mathrm{negl}_{DDH}(\lambda) + 1/q) \\ &+ \mathrm{negl}_{Sym}(\lambda) + \mathrm{negl}_{Mix}(\lambda) \end{aligned}$$

Since the sum of negligible functions is itself negligible, we conclude that $\mathcal{A}$'s advantage in the original experiment is negligible. □

A.3 Proof of Theorem 3

Proof. We show that for any PPT adversary $\mathcal{A}$ that wins the Restricted Monitoring game with probability ϵ, there exists a PPT algorithm $\mathcal{B}$ that solves the computational Diffie-Hellman (CDH) problem in $\mathbb{G}$ with probability at least $\epsilon - 1/q$. Since CDH is hard in any group where DDH holds, ϵ is negligible in the security parameter λ. We note that the proof holds for any vector V that $\mathcal{A}$ outputs, whether or not it was created by `AuditKeyVecGen`.

Given a CDH challenge (G, X, Y), where $X = \alpha \cdot G$ and $Y = \beta \cdot G$, $\mathcal{B}$ needs to output $\alpha\beta \cdot G$. $\mathcal{B}$ simulates the Restricted Monitoring challenger for $\mathcal{A}$ as follows:

Simulation. $\mathcal{B}$ programs $H(x) = u_x \cdot G + t_x \cdot Y$ for every address x, with $u_x, t_x \leftarrow \mathbb{Z}_q$, as in the proof of Theorem 1. When $\mathcal{A}$ outputs V and S^*, $\mathcal{B}$ samples $\sigma_a \leftarrow \mathbb{Z}_q$ for each address $a \in S^*$ and gives $R_a = X + \sigma_a \cdot G$ to $\mathcal{A}$. This implicitly sets $r_a = \alpha + \sigma_a$, so the R_a are distributed as in the real game. $\mathcal{A}$ returns a key K_a for each $a \in S^*$.

Analysis. For an address $a \in S^*$, let $P_a = \mathsf{RetrieveKey}(V, a) - H(a)$, which $\mathcal{B}$ can compute. RetrieveKey (Algorithm 3) returns the linear relation $\sum_{j=1}^{k} c_j(a) \cdot V[h_j(a)]$, so each address in S^* corresponds to a vector of $|V|$ coefficients in $\mathbb{Z}_q$, one for each slot in the vector V. Now, since $|S^*| > |V|$, there are more addresses than slots. This means that the coefficient vectors are linearly dependent. Namely, there exists a non-zero scalar vector $\vec{\gamma}$ (which $\mathcal{B}$ can find by Gaussian elimination) such that:

$$\sum_{a \in S^*} \gamma_a \cdot \mathsf{RetrieveKey}(V, a) = 0_{\mathbb{G}}$$

Now, let us substitute in the above equation the retrieved key by $P_a + H(a)$, and $H(a)$ by its programmed value. We get:

$$\sum \gamma_a P_a = -\sum \gamma_a H(a) = -\left(\sum \gamma_a u_a\right) \cdot G - \left(\sum \gamma_a t_a\right) \cdot Y$$

If $\mathcal{A}$ wins, then $K_a = r_a \cdot P_a = \alpha \cdot P_a + \sigma_a \cdot P_a$ for every $a \in S^*$. Thus:

$$\begin{aligned}\sum \gamma_a (K_a - \sigma_a \cdot P_a) &= \alpha \cdot \sum \gamma_a P_a \\ &= -\left(\sum \gamma_a u_a\right) \cdot X - \left(\sum \gamma_a t_a\right) \cdot \alpha\beta \cdot G\end{aligned}$$

$\mathcal{B}$ knows all terms in this equation other than $\alpha\beta \cdot G$, and can solve for it whenever $\sum \gamma_a t_a \neq 0$. Since H reveals nothing about the t_a, and $\vec{\gamma}$ is non-zero, this sum is zero with probability $1/q$. Therefore, $\mathcal{B}$ solves the CDH problem with probability at least $\epsilon - 1/q$. □

A.4 Proof of Theorem 4

Proof. We prove both parts by analyzing the linear algebraic structure of the auditkey vector V. Let S be a list of addresses, and let S' be the addresses that `AuditKeyVecGen` keeps when it runs on S (after padding). The encoding requirement defines a linear system over $\mathbb{G}$:

$$\mathbf{M} \times \vec{V} = \vec{AK}$$

Where, $V \in \mathbb{G}^{|V|}$ is the variable column vector, $\vec{AK} \in \mathbb{G}^{|S'|}$ is the column vector of target audit keys, and $\mathbf{M} \in \mathbb{Z}_q^{|S'| \times |V|}$ is the coefficient matrix induced from applying the hash functions (modeled as random oracles) on the addresses in S'. Specifically, we initialize M to a matrix of zeroes and then update each matrix cell as follows, using the hash functions h_ℓ (position) and c_ℓ (coefficient):

$$\forall 1 \le \ell \le k, 1 \le i \le |S'| : M_{i,h_\ell(a_i)} \leftarrow M_{i,h_\ell(a_i)} + c_\ell(a_i)$$

Since `AuditKeyVecGen` drops an address whose equation is linearly dependent on the equations before it, the rows of $\mathbf{M}$ are linearly independent.

Membership Hiding. We show that for every list S, the vector V is uniformly distributed over $\mathbb{G}^{|V|}$.

1. **The target vector is uniform:** Each audit key is $AK_a = sk_a \cdot G + H(a)$ for a secret key sk_a that `AuditKeyVecGen` samples at random. Thus the target vector $\vec{AK}$ of audit keys is uniformly distributed over $\mathbb{G}^{|S'|}$, whatever the addresses in S' are.
2. **The solution vector is uniform:** Since the rows of $\mathbf{M}$ are linearly independent, every target vector $\vec{AK}$ has the same number of solution vectors, namely $q^{|V|-|S'|}$, and every vector in $\mathbb{G}^{|V|}$ is a solution for exactly one target vector. `AuditKeyVecGen` outputs a uniformly random solution for a uniformly random target vector, so its output is uniformly distributed over $\mathbb{G}^{|V|}$.

Therefore, both options for the resulting vector V, for S_0 and for S_1, are identically distributed, and the adversary cannot distinguish between them:

$$\mathbf{Adv}_{\mathcal{A}}^{\text{Membership Hiding}} = \left|\Pr[b' = b] - \frac{1}{2}\right| = 0$$

Monitoring Coverage. Let Q be the number of addresses on which $\mathcal{A}$ queries the hash functions (we assume, without loss of generality, that these include the addresses in S), and let $s = \lfloor |V|/2 \rfloor$. If `AuditKeyVecGen` drops one of the first s addresses of S, then the equations of some $t \le s$ of the queried addresses are linearly dependent. Namely, there exists a set T of t queried addresses and non-zero scalars $\{\gamma_a\}_{a \in T}$ such that $\sum_{a \in T} \gamma_a \mathbf{M}_a = 0$, where $\mathbf{M}_a$ is the row of coefficients of address a. We bound the probability that such a set exists. Since the hash functions are random oracles, the answer to each new query is uniformly random and independent of the previous answers, even though $\mathcal{A}$ chooses its queries adaptively. Thus the rows of the queried addresses are independent, and it suffices to bound the probability for each of the $\binom{Q}{t}$ sets T of each size $t \le s$ and sum. Let U be the set of slots in V that the addresses in T are mapped to. We analyze two events:

1. **Structural dependency, $|U| < t$:** All $k \cdot t$ positions of the addresses in T fall in some set of $t - 1$ slots. This happens with probability at most $\binom{|V|}{t-1}\left(\frac{t-1}{|V|}\right)^{kt}$.
2. **Algebraic dependency, $|U| = u \ge t$:** Fix the positions and the scalars γ_a. For each slot in U, the corresponding coordinate of $\sum_{a \in T} \gamma_a \mathbf{M}_a$ is a non-trivial linear combination of the random coefficients mapped to that slot, which is zero with probability $1/q$, independently of the other slots. So all u coordinates are zero with probability q^{-u}. There are at most q^{t-1} choices for the scalars γ_a (up to scaling), and the positions fall in a set of u slots with probability at most $\binom{|V|}{u}\left(\frac{u}{|V|}\right)^{kt}$.

Summing over the sets T gives the bound from §7.4:

$$\Pr[\mathcal{A} \text{ wins}] \le \sum_{t=1}^{s} \binom{Q}{t} \Big[\binom{|V|}{t-1} \left(\tfrac{t-1}{|V|}\right)^{kt} + \sum_{u=t}^{|V|} \binom{|V|}{u} \left(\tfrac{u}{|V|}\right)^{kt} q^{\,t-1-u} \Big]$$

It remains to show that this bound is negligible in the security parameter λ when $k = \omega(\log \lambda)$ (in AuditPay, k is linear in λ). For the structural term, since $t - 1 < |V|/2$, we have

$$\binom{Q}{t}\binom{|V|}{t-1}\left(\tfrac{t-1}{|V|}\right)^{kt} \le Q^t \cdot e^{t-1} \cdot \left(\tfrac{t-1}{|V|}\right)^{(k-1)t+1} \le \left(e \cdot Q \cdot 2^{-(k-1)}\right)^t,$$

and the term for $t = 1$ is zero. Since a PPT adversary makes a polynomial number of queries Q, the sum of these terms over $t \ge 2$ is negligible. A similar calculation, which also uses that q is exponential in λ, shows that the sum of the algebraic terms is negligible. We evaluate the bound for AuditPay's parameters in §7.4. □