We introduce the notion of mixnets. In a traditional mixnet, multiple mix-servers jointly permute and decrypt a list of ciphertexts to produce a list of plaintexts, along with a proof of correctness, such that the association between individual ciphertexts and plaintexts remains completely hidden. However, in many applications, the privacy-utility tradeoff requires answering some specific queries about this association, without revealing any information beyond the query result. We consider queries of the following types: a) given a ciphertext in the mixnet input list, whether it encrypts one of a given subset of plaintexts in the output list, and b) given a plaintext in the mixnet output list, whether it is a decryption of one of a given subset of ciphertexts in the input list. Traceable mixnets allow the mix-servers to jointly prove answers to the above queries to a querier such that neither the querier nor a threshold number of mix-servers learn any information beyond the query result. Further, if the querier is not corrupted, the corrupted mix-servers do not even learn the query result. We first comprehensively formalise these security properties of traceable mixnets and then propose a construction of traceable mixnets using novel distributed zero-knowledge proofs (ZKPs) of set membership and of a statement we call reverse set membership. Although set membership has been studied in the single-prover setting, the main challenge in our distributed setting lies in making sure that none of the mix-servers learn the association between ciphertexts and plaintexts during the proof. We implement our distributed ZKPs and show that they are faster than state-of-the-art by at least one order of magnitude.

**ABSTRACT**

We introduce the notion of traceable mixnets. In a traditional mixnet, multiple mix-servers jointly permute and decrypt a list of ciphertexts to produce a list of plaintexts, along with a proof of correctness, such that the association between individual ciphertexts and plaintexts remains completely hidden. However, in many applications, the privacy-utility tradeoff requires answering some specific queries about this association, without revealing any information beyond the query result. We consider queries of the following types: a) given a ciphertext in the mixnet input list, whether it encrypts one of a given subset of plaintexts in the output list, and b) given a plaintext in the mixnet output list, whether it is a decryption of one of a given subset of ciphertexts in the input list. Traceable mixnets allow the mix-servers to jointly prove answers to the above queries to a querier such that neither the querier nor a threshold number of mix-servers learn any information beyond the query result. Further, if the querier is not corrupted, the corrupted mix-servers do not even learn the query result. We first comprehensively formalise these security properties of traceable mixnets and then propose a construction of traceable mixnets using novel distributed zero-knowledge proofs (ZKPs) of set membership and of a statement we call reverse set membership. Although set membership has been studied in the single-prover setting, the main challenge in our distributed setting lies in making sure that none of the mix-servers learn the association between ciphertexts and plaintexts during the proof. We implement our distributed ZKPs and show that they are faster than state-of-the-art by at least one order of magnitude.

**KEYWORDS**

verifiable mixnets; traceability; distributed zero-knowledge proofs; set membership; reverse set membership

**1 INTRODUCTION**

A mixnet is a cryptographic primitive used for anonymous messaging, specifically for unlinking the identity information of data providers with the sensitive data they provide. Let $u_i$ denote the identity information of the $i^{th}$ individual and $v_i$ denote the sensitive data they contribute. For example, in a secure electronic voting context where mixnets are commonly used, $u_i$ may contain a voter id and $v_i$ may contain the vote. The link between $u_i$ and $v_i$ is hidden by encrypting $v_i$ to obtain ciphertexts (encrypted votes) $c_i$, uploading $u_i$ along with $c_i$ to an input list, and feeding the list of ciphertexts $c$ as the mixnet input. The mixnet, which consists of a series of mix-servers, processes these ciphertexts and outputs a list $\sigma'$ of decryptions of the ciphertexts in a randomly permuted order [23]. The secret permutation that links the ciphertexts with their corresponding plaintexts is shared among the mix-servers and remains hidden unless a threshold number of mix-servers are corrupted, thus completely hiding which ciphertext or voter id corresponds to which vote (see Figure 1a). Using verifiable mixnets [53], the mix-servers can also prove to a verifier that the output list is obtained correctly by permuting and decrypting each element of the input list, while keeping the linkages between the ciphertexts and plaintexts completely hidden.

However, sometimes, it is necessary to reveal specific partial information about the association between $c_i$ and $v_i$, while still preventing any additional information leakage. For example, consider a large-scale public election running across multiple polling booths. Further, assume a dual voting setup [8, 65] where voters cast their vote both electronically and on paper: the electronic system produces an encrypted vote as the voter receipt and processes these encrypted votes via a mixnet, whereas the paper votes are collected to form a physical audit trail. Such dual voting systems aim to improve the overall robustness and transparency of electoral processes. In such systems, revealing partial information about the encrypted and decrypted votes allows graceful recovery from disputes without re-running the entire election [2]. For example, if there is a mismatch between a plaintext vote in the mixnet output list and its corresponding paper record, the ability to pinpoint which specific booth the disputed vote came from enables a localised recovery of the election by selectively rerunning the election only at the corrupted booths. Yet, it is important to not reveal additional information such as vote counts of all the polling booths or, worse, votes of individual voters. Further, any such partial information revealed should be provable, to ensure that recovery steps lead to...
the correct election outcome. A traditional mixnet does not allow for the release of such controlled partial information because votes cast at all the booths are anonymised together before decryption.

Conversely, consider a voter claiming that their published encrypted vote does not match the one in their (possibly fake) receipt. If it can be shown that their published encrypted vote decrypted to a plaintext vote that mismatches with its corresponding paper record, it supports the voter’s claim because an incorrectly uploaded encrypted vote must also mismatch with the paper vote on decryption. As above, this revealed information should be provable and should not leak any additional information, e.g., the voter’s specific vote. Traditional mixnets do not allow this either.

To address this gap, we introduce the notion of traceable mixnets. Let $c_l := \{c_{i} \mid i \in I\}$ for some index set $I$ denote a subset of ciphertexts in the input list and $\sigma'_{j} := \{\sigma'_{j} \mid j \in J\}$ for some index set $J$ denote a subset of plaintexts in the output list. A traceable mixnet allows its mix-servers to jointly and provably answer the following queries to an interested and authorised querier:

- **TraceIn**($c_i, \sigma'_{j}$): Does the ciphertext $c_i$ in the input list encrypt a plaintext in set $\sigma'_{j}$ (Figure 1b)? Given an encrypted vote $c_i$ mismatching a voter receipt and a set $\sigma'_{j}$ of plaintext votes mismatching with their paper records, this query answers if $c_i$ decrypted to a matching plaintext vote and thus was incorrect.

- **TraceOut**($c_i, \sigma'_{j}$): Is the plaintext $\sigma'_{j}$ in the output list encrypted in a ciphertext in set $c_i$ (Figure 1c)? Given an output plaintext vote $\sigma'_{j}$ mismatching with its paper vote and a set $c_i$ of encrypted votes cast at a given polling booth $B$, this query answers if the disputed vote $\sigma'_{j}$ came from booth $B$.

The queries are answered such that no additional information beyond the query output is leaked to an adversary controlling the querier and less than a threshold number of mix-servers. Also, to prevent mix-servers from accumulating query responses issued to different queriers over time, the mix-servers are not allowed to even learn the output of a query if they do not control the querier. Such controlled querying mechanism is useful in voting as well as several other applications where privacy-preserving data sharing with guaranteed correctness is required.

Note, though, that the query outputs may themselves leak sensitive information, especially when multiple queries are combined. Thus, arbitrary queries cannot be allowed in any application. Deciding what queries to allow requires a privacy risk analysis of the overall information leaked by the queries, followed by an analysis of whether the leakage is acceptable for the application’s privacy requirements. For example, in the voting application, the allowed queries are decided so that the recovery process does not leak the vote counts of any booth except the corrupted booths [2]. Once a policy detailing the allowed queries is decided, possibly allowing different queries to different queriers, the application layer must also ensure that the mix-servers comply with the policy. The traceable mixnet guarantees that if an honest mix-server follows the policy, the adversary learns nothing about the output of the queries disallowed by the policy. Without such analysis and policy compliance though, a traceable mixnet solution should not be deployed.

We emphasise that a traceable mixnet’s secrecy requirements are more stringent than what existing proof-of-shuffle and verifiable decryption techniques in the area of verifiable mixnets [53] can provide. For example, given a mixnet input list $\{c_1, \ldots, c_5\}$ and output list $\{\sigma'_1, \ldots, \sigma'_6\}$, a \texttt{TraceIn}(c_4, {\sigma'_1, \sigma'_2, \sigma'_4}) query requires proving in ZK that $c_4$ decrypts to one of $\sigma'_1, \sigma'_2$, and $\sigma'_4$. Using these techniques, the mix-servers can prove this by, e.g., verifiably revealing the subset $C := \{c_1, c_2, c_4\}$ of the input ciphertexts that decrypt to the set of plaintexts $\{\sigma'_1, \sigma'_2, \sigma'_4\}$ and letting the querier verify that $c_4 \in C$ (see Figure 2b). However, this is not ZK: it reveals, e.g., that ciphertexts $c_1, c_2$ decrypt to one of $\sigma'_1$, $\sigma'_2$, and $\sigma'_4$ (and not $\sigma'_3$ or $c_5'$). Figure 2 shows other similar leakage scenarios. A traceable mixnet does not reveal any such intermediate information.

Finally, our focus is on offline batch processing, i.e., efficiently answering multiple \texttt{TraceIn}/\texttt{TraceOut} queries against the same set. Thus, we define the following batched queries: a) **BTraceIn**($c_i, \sigma'_{j}$): which ciphertexts in set $c_i$ encrypt a plaintext in set $\sigma'_j$; and b) **BTraceOut**($c_i, \sigma'_{j}$): which plaintexts in set $\sigma'_j$ are encrypted in a ciphertext in set $c_i$. We require the amortised time for batched queries to be linear in the input list size to enable practical applications like dispute resolution in elections with millions of votes. Note that offline batch processing in verifiable mixnets is distinct from mixnets in real-time anonymous communication networks [36]. Thus, we do not aim to answer the queries in real time.

### 1.1 Our contributions

Our main contributions are the following. First, we introduce the notion of **traceable mixnets** and formalise their completeness, soundness and secrecy requirements (Section 2).

(a) A traditional verifiable mixnet.

(b) A TraceIn($c_2, \{\sigma_3, \sigma_2, \sigma_1\}$) query in a traceable mixnet: whether $c_2$ encrypted one of $\{\sigma_3, \sigma_2, \sigma_1\}$.

(c) A TraceOut($c_1,\{c_2, c_3\}, \sigma_1$) query in a traceable mixnet: whether $\sigma_1$ was encrypted in one of $\{c_1, c_2, c_3\}$.

**Figure 1**: Traditional and traceable mixnets. $u_i$ denotes the $i$th individual’s identity information and $v_i$ denotes their sensitive data; $c_i$ encrypts $v_i$ and are passed as input to the mixnet; the mixnet consists of mix-servers $M_1$ and $M_2$ that jointly decrypt and permute input list $c$ to output a plaintext list $\sigma' := (\pi_{\sigma_{M_i}(i)})_{i=1}^{5}$, where $\pi$ is composed of secret permutations $\pi^{(1)}$ and $\pi^{(2)}$ of $M_1$ and $M_2$. 

...
Traceable mixnets

![Diagram of Traceable Mixnets]

**Figure 2:** Subfigures (a) and (b) are for a TraceIn($e_i, V$) query, where $V = \{\nu'_1, \nu'_2, \nu'_3\}$. With proofs-of-shuffle and verifiable decryption, the mix-servers can either a) verifiably reveal the plaintext encrypted by $e_i$, say $\nu'_1$, and let the querier check if $\nu'_1 \in V$, or b) verifiably reveal the set of ciphertexts encrypting set $V$, say $C := \{e_1, e_2, e_3\}$, and let the querier check if $e_i \in C$. Subfigures (c) and (d) are for a TraceOut($C, \nu'_2$) query, where $C = \{e_1, e_4, e_5\}$. The mix-servers can either c) verifiably reveal the ciphertext encrypting $\nu'_2$, say $e_5$, and let the querier check if $e_5 \in C$, or d) verifiably reveal the set of plaintexts that set $C$ decrypts to, say $V := \{\nu'_1, \nu'_2, \nu'_3\}$, and let the querier check that $\nu'_2 \in V$. With traceable mixnets, they do not need to reveal any such intermediate information.

Second, we propose a construction of traceable mixnets (Section 4) using novel distributed ZKPs of set membership [19] and a novel primitive called reverse set membership. Given a commitment scheme [74] comm with commitment space $\Gamma$, message space $V$ and randomness space $R$, a ZKP of set membership for a commitment $\gamma \in \Gamma$ and a set of values from $V$ proves that $\gamma$ commits a member of $V$. Formally, we denote this as $\rho_{SM}(\gamma, \phi) := \text{PK}((\nu, r) : \gamma = \text{comm}(\nu; r) \land \nu \in \phi)$, a proof of knowledge of a member $\nu$ of set $\phi$ and a randomness $r$ such that $\gamma$ commits $\nu$ with randomness $r$. A ZKP of reverse set membership for a value $\nu \in V$ and a set $\Phi$ of commitments from $\Gamma$ proves that $\nu$ is committed by a member of $\Phi$. Formally, we denote this as $\rho_{RSM}(\Phi, \nu) := \text{PK}((\nu, r) : \nu = \text{comm}(\nu; r) \land \nu \in \Phi)$, a proof of knowledge of a randomization $r$ such that some member $\gamma$ of $\Phi$ commits $\nu$ with randomness $r$. Prior work has mainly focused on the ZKP of set membership and the single-prover case, whereas our ZKPs (both for set membership and the novel reverse set membership) work in a distributed setting where the mixnet’s mix-servers jointly act as the provers. The mix-servers need to carry out the proof without themselves learning any information about either the commitment openings or the association between commitments and plaintext values. Our ZKPs are interactive, which is acceptable for our inherently interactive use-case.

Third, we provide detailed security analysis of our construction and formal rules for privacy risk analysis of a given set of allowed TraceIn/TraceOut queries (Section 5 and Appendices C-D). Fourth, we provide a comprehensive implementation of our proposal (Section 6). Our construction has linear time complexity in the size of the mixnet input list for batched queries and greatly outperforms even single-prover existing techniques. Specifically, our distributed ZKPs of set membership and reverse set membership (which enable BTraceIn and BTraceOut respectively) have per-prover proving times respectively 43x and 9x faster than single-prover zkSNARK-plus-Merkle tree based proofs. By conservative estimates, this makes them at least 86x and 18x faster than the state-of-the-art collaborative zkSNARKs [72] in the distributed setting. Our implementation is open source and available at [1].

1.2 Related work

1.2.1 Controlled information release in statistical databases. A common approach for controlled information release for statistical analysis is by anonymising (releasing a noisy version of) the dataset. Another approach is differential privacy (DP) [37]; interactively providing noisy answers to analytics queries such that the answer distribution is insensitive to any given individual’s data. However, anonymisation is a known poor safeguard against re-identification attacks [31, 69]. Further, because of correlations in different individuals’ data items, differentially private mechanisms can still reveal arbitrary partial information about the link between users’ identity information and sensitive data to an adversary running arbitrary queries [84]. In comparison, traceable mixnets reveal answers to only pre-approved queries and otherwise keep users’ identity information and sensitive data unlinkable. Also, their distributed setting naturally fits into privacy-sensitive applications, whereas most anonymisation/DP solutions assume a trusted data curator.

1.2.2 Existing traceability notions. Many works [5, 25, 54, 78–80] aim to balance anonymity with accountability by letting users communicate anonymously by default but allowing a trusted third party to revoke the anonymity of users that misbehave by sending illegal, misinformative or offensive messages. This is done by tracing the exact senders of the offending messages, in contrast to our generalised set-based and bidirectional notion of traceability.

Group/ring signatures [22, 75] let a verifier verify that a message was sent by a member of a group, without learning which member. This resembles our TraceOut query if we map the group of senders with the subset of input ciphertexts. However, these signatures require active involvement of the senders and are not logistically suitable for backend analytics applications. Likewise, anonymous credentials [20] let individuals prove in ZK that they satisfy some eligibility criteria, resembling our TraceIn queries, but they also require individuals’ active involvement in securing their anonymity, whereas this is done by distributed mix-servers in traceable mixnets.

1.2.3 Existing verifiable mixnets. Haines and Müller [53] review and identify the following existing techniques for building verifiable mixnets: message tracing [61, 82], verification codes [61, 76], trip wires [17, 58], message replication [58], randomised partial checking (RPC) [55, 59, 62, 63] and proofs of shuffle [70, 77, 83]. These techniques only verify that the mixnet output list was a decryption and permutation of its input ciphertext list and do not support the fine-grained TraceIn/TraceOut queries. Message tracing and verification codes provide limited traceability by letting senders verify the processing of their own ciphertexts, but one who does not hold the ciphertext secrets cannot perform this verification.

Proofs of shuffle are the state-of-the-art verifiable mixnet techniques. A proof-of-shuffle proves in ZK that two ciphertext lists are permutations and re-encryptions of each other. This, combined with verifiable decryption techniques [44], provides a ZKP that a list of plaintexts is a decryption and permutation of a list of ciphertexts. These approaches can also prove that a sublist of the plaintext list is a decryption and permutation of its corresponding sublist in the input ciphertext list. However, as shown in Figure 2, they leak extra information beyond TraceIn/TraceOut query outputs.
Note that unlike non-interactive verifiable mixnets, an interactive verifiable/traceable mixnet necessarily requires the mix-servers to store their permutations. Thus, forward secrecy of query results is not maintained under future compromise of stored permutations.

### 1.2.4 Set membership proofs

Below we review set membership and reverse set membership ZKP techniques, first for a single prover.

**Techniques with quadratic complexity.** Cramer et al. [28] propose a generic technique to create a zero-knowledge $\Sigma$-protocol for the OR composition of two statements, given $\Sigma$-protocols for each individual statement. Both ZKPs of set membership and reverse set membership can be constructed using this technique: $\rho_{\text{SM}}(y, \phi) \equiv \text{PK}(\langle r \rangle : \sqrt{y} = \text{comm}(y; r))$ and $\rho_{\text{SM}}(\Phi, v) \equiv \text{PK}(\langle r \rangle : \sqrt{y} = \text{comm}(y; r))$. However, for proving $\rho_{\text{SM}}(y, \phi)$ for multiple $y$s against the same set $\phi$ or $\rho_{\text{SM}}(\Phi, v)$ for multiple $v$s against the same set $\Phi$ (the “batched” queries), it results in an overall $O(n^2)$ complexity, where $n = |\Phi|, |\phi|$. Groth and Kohlweiss [51] propose a ZKP of knowledge of the form $\rho_{\text{SM}}(\phi) \equiv \text{PK}(\langle r \rangle : \sqrt{y} = \text{comm}(y; r))$, i.e., a proof that one of the commitments in a set commits to 0, with $O(\log(n))$ communication complexity. Interestingly, $\rho_{\text{SM}}(\Phi, v)$ can be used to prove both $\rho_{\text{SM}}$ and $\rho_{\text{RSM}}$ [51], resulting in an $O(n \log(n))$ communication complexity for the batched queries. However, the computational complexity (for both the prover and the verifier) remains $O(n^2)$.

**Accumulator-based techniques.** Cryptographic accumulators [9, 68, 71] enable efficient ZKPs of set membership. An accumulator scheme ($\text{Acc, GenWitness, AccVer}$) allows computing a short digest $A_y \leftarrow \text{Acc}(\phi)$ and a short membership witness $w_v$ for a member $v \in \phi$ as $w_v \leftarrow \text{GenWitness}(A_y, \phi)$ such that $\text{AccVer}(A_y, v, w_v) = 1$ is a proof that $v \in \phi$. A ZKP of set membership can thus be constructed by requiring both prover and verifier to compute $A_y$, the prover to compute $w_v$ and both to then engage in $\rho_{\text{SM}}(\Phi, v) \equiv \text{PK}(\langle r, v, w_v \rangle : y = \text{comm}(y; r) \land \text{AccVer}(A_y, v, w_v) = 1)$. If the accumulator scheme allows commitments to be set members, a ZKP of reverse set membership can be similarly constructed using $\rho_{\text{SM}}(\text{Acc}, A_y) \equiv \text{PK}(\langle r, w \rangle : \text{AccVer}(A_y, \text{comm}(v; r), w) = 1)$, where $w$ denotes the membership witness of the commitment $\text{comm}(v; r) \in \Phi$.

A popular approach for accumulator-based ZKPs of set membership involves Merkle accumulators [68] as the accumulator scheme and zkSNARKs [24, 45, 50] as the ZK proof system. With Merkle accumulators, both $\text{GenWitness}$ and $\text{AccVer}$ take $O(\log(n))$ time, where $n = |\phi|$, which allows $n$ set membership and reverse set membership proofs in $O(n \log(n))$ time. This approach can also generically support $\rho_{\text{SM}}(\text{Acc})$ by computing Merkle accumulators for the set of commitments. However, it involves expensive hash computations inside the zkSNARK circuit. Benarroch et al. [12] present a non-generic ZKP of set membership using RSA accumulators, avoiding these expensive hash computations. However, the technique does not support reverse set membership. Also, computing witness $w_v$ for RSA accumulators takes $O(n)$ time, which makes it $O(n^2)$ for batched queries. Techniques to efficiently batch multiple membership proofs together also exist [15, 21].

Extending to the distributed setting. All the above techniques work on the single-prover case. Extending them to our distributed setting where none of the provers (the mix-servers) know either the commitment openings or the permutation between the commitments and the plaintexts is non-trivial. The batching techniques [15, 21] also fail in the distributed setting since they require the prover to know upfront which entries pass the membership proof.

Collaborative zkSNARKs [72] allow a distributed set of provers holding secret shares of a SNARK witness to prove joint knowledge of the same. They add roughly 2x overhead in per-prover proving time over the standard zkSNARKs [72]. DPZKs [32] provide similar guarantees. However, even to securely obtain shares of the SNARK witness, an extra MPC protocol is likely required.

**Signature-based set membership.** Camenisch et al. [19] initiated a different approach to ZKPs of set membership: the verifier provides to the prover signatures on all members of the set under a fresh signing key generated by the verifier, and the prover proves knowledge of a signature on the committed value in ZK. This proves set membership because if the commitment commits a value outside the set, the prover does not obtain a signature on it and must forge it. This approach gives $O(n)$ batched query complexity because verifier signatures can be reused. We extend these ZKPs by a signature-based ZKP of reverse set membership and by distributed signature-based ZKPs for both set membership and reverse set membership.

## 2. Formal Definitions

We now formalise traceable mixnets. We directly present the batched BTraceIn/BTraceOut protocols as that is our main focus (the TraceIn/TraceOut protocols are trivial special cases). Also, for simplicity, we present the special case when all $t = m$ mix-servers are required to decrypt the ciphertexts but any set of less than $t$ mix-servers cannot break secrecy. Extension to a general $t$ is possible with standard threshold cryptography techniques [34]. Finally, we assume an authenticated broadcast channel that ensures authenticity, availability and non-repudiability of all published messages.

**Notation.** Given a positive integer $n$, we denote the set $\{1, \ldots, n\}$ by $[n]$. We let $(\text{boldface}) x \in \mathbb{X}^n$ denote an $n$-length vector of values drawn from a set $X$, $x_k$ denote the $k$th component of $x$, and, given an index set $I \subseteq [n]$, $x_I$ denote the set $\{x_i : i \in I\}$. For any function $f : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{X}$ and vectors $x, y \in \mathbb{X}^n$, we let $x \circ y$ denote the vector $(x_i \circ y_j)_{i,j \in [n]}$, i.e., the vector obtained by component-wise application of $\circ$. We let $u^\circ$ denote $(u_i^\circ)_{i \in [n]}$, $f(u)$ denote $(f(u_i))_{i \in [n]}$, $f(x, y)$ denote $(f(x, u_i), y_i)_{i \in [n]}$.

We denote a multiparty computation protocol $P$ between parties $P_1, \ldots, P_m$ where the common input of each party is $c_i$, $P_k$’s secret input is $s_{ik}$, the common output is $o_{ik}$ and $P_k$’s secret output is $s_{ok}$ as follows: $c_i, (P_k[\langle s_{ok} \rangle])_{k \in [m]} \leftarrow P(c_i, (P_k[\langle s_{ik} \rangle])_{k \in [m]}$. When a party does not have a secret output, we drop it from the left-hand side. In security experiments where the experimenter plays the role of honest parties $(P_k)_k \in \mathcal{H}$ for some $H \subset [m]$ and an adversary $A$ plays the role of $(P_k)_k \notin \mathcal{H}$, we indicate it as $(P_k[\langle s_{ok} \rangle])_{k \notin \mathcal{H}} \leftarrow P(c_i, (P_k[\langle s_{ik} \rangle])_k \in \mathcal{H})$. We let $x^{(k)}$ denote a component (typically, secret-share) of $x$ designated for $P_k$. We call a function $f$ negligible if for any polynomial $p$, there exists an $N \in \mathbb{N}$ such that $f(x) < 1/p(x)$ for all $x > N$.

**Definition 1 (Traceable mixnets).** A traceable mixnet is a tuple of protocols/algorithms (Keygen, Enc, Mix, BTraceIn, BTraceOut)
between $n$ senders ($S_i$)$_{i \in [n]}$, $m$ mix-servers ($M_k$)$_{k \in [m]}$ and a querier or verifier $Q$ such that:
- $\text{mpk} = (M_k[[\text{msk}(k)]]_{k \in [m]} \leftarrow \text{Keygen}(1^\lambda, (M_k[[\text{msk}(k)]]_{k \in [m]}))$ is a key generation protocol between ($M_k$)$_{k \in [m]}$, where $\lambda$ is a security parameter (given in unary). Individual mix-servers do not have any secret input, the common output is a mixnet public key $\text{mpk}$ and each $M_k$'s secret output is a secret key $\text{msk}(k)$.
- $c_i \leftarrow \text{Enc}(\text{mpk}, a_i)$ is an algorithm run by sender $S_i$, where $\text{mpk}$ is the mixnet public key and $a_i$ is $S_i$'s sensitive input drawn from some plaintext space $\mathbb{V}$. Output $c_i$ is a ciphertext "encrypting" $a_i$.
- $\psi' = (M_k[[\omega(k)]]_{k \in [m]} \leftarrow \text{Mix}(\text{mpk}, c, (M_k[[\text{msk}(k)]]_{k \in [m]}))$ is a mixing protocol between ($M_k$)$_{k \in [m]}$, where $\text{mpk}$ is the mixnet public key and $c \leftarrow (\text{Enc}(\text{mpk}, a_i))_{i \in [n]}$ is a vector of ciphertexts encrypting the senders' plaintexts ($a_i$)$_{i \in [n]}$. Each $M_k$'s secret input is its secret key $\text{msk}(k)$. The common output is a vector $\psi'$ of plaintext values obtained after permuting and decrypting $c$ (thus $\psi' = (a_{\pi(i)})_{i \in [n]}$ for some permutation $\pi$). Each $M_k$'s secret output is a witness $\omega(k)$ to be used in proving correctness of the $\text{BTraceIn}/\text{BTraceOut}$ outputs (see below).
- $\text{BTraceOut}(\text{mpk}, c, a', I, J, (M_k[[\text{msk}(k), \omega(k)]]_{k \in [m]}), Q([\bullet]))$ is a protocol between ($M_k$)$_{k \in [m]}$ and querier $Q$, where $a', M_k[[\omega(k)]]_{k \in [m]} \leftarrow \text{Mix}(\text{mpk}, c, (M_k[[\text{msk}(k)]]_{k \in [m]}))$, $c \leftarrow (\text{Enc}(\text{mpk}, a_i))_{i \in [n]}$ and $I, J \subseteq [n]$. $Q$'s secret output is a set of ciphertexts $c_I = \{c_j \mid j \in I\} \cup \{c_j \mid j \in J\}$; $Q$ may abort if it is not convinced about the correctness of $c_I$.
- $\text{BTraceOut}(\text{mpk}, c, a', I, J, (M_k[[\text{msk}(k), \omega(k)]]_{k \in [m]}), Q([\bullet]))$ is a protocol between ($M_k$)$_{k \in [m]}$ and $Q$, where all inputs are exactly the same as $\text{BTraceIn}$ and $Q$'s secret output is a set of plaintexts $a'_I = \{a'_j \mid j \in I\} \cup \{a'_j \mid j \in J\}$; $Q$ may abort if it is not convinced about the correctness of $a'_I$.

2.1 Completeness

The completeness definition for traceable mixnets (Definition 2 and Figure 3) models that when all the parties are honest then a) $Q$'s output $c_I$ on a $\text{BTraceIn}$ query on $(c, a', I, J)$ is exactly the set of ciphertexts in $c_I$ that encrypted some plaintext in $a'_I$ (line 7), and b) its output $a'_I$ on a $\text{BTraceOut}$ query on $(c, a', I, J)$ is exactly the set of plaintexts in $a'_I$ that were encrypted by some ciphertext in $c_I$ (line 8). Note that we only consider the case of distinct values. The case of repeated values is trivially reducible to this case if $S_i$ prefixes $a_i$ with a nonce drawn uniformly from a large set.

**Definition 2 (Completeness).** A traceable mixnet is complete if for each security parameter $\lambda \in \mathbb{N}$, number of ciphertexts $n \in \mathbb{N}$, vector $c \in \mathbb{V}^n$ of distinct plaintext values and index sets $I, J \subseteq [n]$, there exists a negligible function $\negl$ such that

$$\Pr[\text{Exp}^{\text{completeness}}(1^\lambda, n, a, I, J) = 1] \geq 1 - \negl(\lambda),$$

where $\text{Exp}^{\text{completeness}}$ is as defined in Figure 3.

2.2 Soundness

The soundness definition (Definition 3) models that as long as input ciphertexts are well-formed, even if all the mix-servers are dishonest, sets $c_I$ and $a'_I$ output by $Q$ in $\text{BTraceIn}$ and $\text{BTraceOut}$ respectively must be "correct," where the correctness of $c_I$ and $a'_I$, is exactly as defined in $\text{Exp}^{\text{completeness}}$. Thus, we do not allow the cheating mix-servers to force $Q$ to produce incorrect output ($Q$ may abort, though). We only consider well-formed input ciphertexts because "correct" processing of ill-formed inputs is undefined. Nevertheless, proofs of well-formedness of inputs are generally required for application-level correctness and these can be constructed by the senders at the time of uploading their ciphertexts.

The experiment (Figure 4) begins with the key generation protocol where the adversary $\mathcal{A}$ controlling all the mix-servers ($M_k$)$_{k \in [m]}$ provides the mixnet public key $\text{mpk}$ (line 1). We let $\mathcal{A}$ supply the plaintexts but create ciphertexts from them honestly (lines 2-3), to model that $\mathcal{A}$ can supply plaintexts of its choice to cheat but the ciphertexts must be well-formed. We then allow $\mathcal{A}$ to run the Mix protocol and produce the output list $\psi'$ (line 4). $\mathcal{A}$ then outputs index sets $I$ and $J$ on which it wants to break the subsequent $\text{BTraceIn}/\text{BTraceOut}$ queries (line 3). During these queries, $Q$ outputs $c_I$ and $a'_I$, respectively (lines 6-7). $\mathcal{A}$ wins if it supplies distinct entries and $Q$ produces valid outputs $c_I$ and $a'_I$ (does not abort) but at least one of them is incorrect (lines 8-10).

**Definition 3 (Soundness).** A traceable mixnet is sound if for each PPT adversary $\mathcal{A}$ and security parameter $\lambda \in \mathbb{N}$, there exists a negligible function $\negl$ such that

$$\Pr[\text{Exp}^{\mathcal{A}}_{\text{soundness}}(1^\lambda) = 1] \leq \negl(\lambda),$$

where $\text{Exp}^{\mathcal{A}}_{\text{soundness}}$ is as defined in Figure 4.

2.3 Secrecy

The secrecy definition (Definition 4) extends a standard anonymity property [10, 11, 13, 33] to the case when the $\text{BTraceIn}/\text{BTraceOut}$ queries are also allowed. The standard anonymity property can be
stated for our setting as follows: an adversary controlling all-but-two senders, the querier and any set of less than \( m \) mix-servers should not be able to distinguish between a world where ciphertexts \((c_0, c_1)\) sent by the two honest senders encrypt values \((v_0, v_1)\) (world 0) and the world where they encrypt \((v_1, v_0)\) (world 1). When BTraceIn/BTraceOut queries are allowed, distinguishing between the two worlds is trivial because of the query outputs (e.g., if \( l, j \) given to a BTraceIn query are such that \( c_0, c_1 \in c_1 \) and \( v_0, v_1 \in v' \) but \( v_1 \not\in v' \), then \( \mathcal{A} \) immediately knows it is in world 0 if output \( v' \) includes \( c_0 \)). Thus, we require that \( a \) in all the BTraceIn queries either both \( v_0, v_1 \in v' \) or both \( v_0, v_1 \not\in v' \) and \( b \) in all the BTraceOut queries either both \( c_0, c_1 \in c_1 \) or both \( c_0, c_1 \not\in c_1 \).

In more detail, in this experiment (Figure 5), adversary \( \mathcal{A} \) engages in the key generation protocol where it controls all the mix-servers except one, i.e., \( M_{\mathcal{L}} \) (line 1). It then supplies input ciphertexts for all the senders except the two that it does not control, say \( S_i \) and \( S_j \). For these senders, it supplies the values \( v_0, v_1 \) (line 2). In world 0 \((b = 0)\), \( S_i \)'s ciphertext \( c_0 \) encrypts \( v_0 \) and \( S_j \)'s ciphertext \( c_1 \) encrypts \( v_1 \); in world \( 1 \) \((b = 1)\), this order is reversed (lines 3-4). The ciphertext list thus formed is processed through the Mix protocol, where \( \mathcal{A} \) controls all mix-servers except \( M_{\mathcal{L}} \) and produces an output plaintext list \( v' \) (line 5). Then, \( \mathcal{A} \) obtains access to oracles OTraceIn, OTraceOut, and OTraceOut which let it choose \( l, f \), control \( Q \) and all mix-servers except \( M_{\mathcal{L}} \), and interact with \( M_{\mathcal{L}} \) in the BTraceIn, BTraceOut protocols (lines 6-14). \( \mathcal{A} \) is required to respect the condition of including either both or none of the honest senders’ ciphertexts/plaintexts in its oracle calls (lines 12 and 18). Finally, \( \mathcal{A} \) outputs a bit \( b' \) as its guess of the bit \( b \) (line 5) and wins if its advantage in making the correct guess is non-negligible.

**Definition 4 (Secrecy).** A traceable mixnet protects secrecy if for each PPT adversary \( \mathcal{A} \), security parameter \( \lambda \in \mathbb{N}, k^* \in [m], \) and \( i_0, i_1 \in [n], \) there exists a negligible function \( \text{negl} \) such that

\[
\Pr[\text{Exp}_{\text{secrecy}}(1^k, k^*, i_0, i_1, 0) = 1] - \\
\Pr[\text{Exp}_{\mathcal{A}}(1^k, k^*, i_0, i_1, 1) = 1] \leq \text{negl}(\lambda),
\]

where \( \text{Exp}_{\text{secrecy}} \) is as defined in Figure 5.

Definition 5 below models that when \( \mathcal{A} \) does not control \( Q \), it should not even learn the query outputs.

**Definition 5 (Output secrecy).** A traceable mixnet protects output secrecy if it protects secrecy as per Definition 4 except that in experiment \( \text{Exp}_{\mathcal{A}} \) (Figure 5), \( \mathcal{A} \) does not control \( Q \) during the BTraceIn/BTraceOut calls (lines 10 and 14) and the constraints on lines 9 and 13 are removed.

Note that Definition 4 also models that if the honest mix-server follows a query policy then the adversary gains no information about the output of queries disallowed by the policy, since this case corresponds to an adversary that simply does not call the OTraceIn or OTraceOut oracles for the disallowed queries. The privacy risk associated with the outputs of allowed queries is outside the scope of formal security requirements of traceable mixnets, but we provide a mechanism to analyse this risk in Appendix D.

3 PRELIMINARIES

**Setup.** We assume that the output of the following Setup algorithm is implicitly available to all the parties: \((q, G_1, G_2, G_T, e, f_1, g_1, h_1, f_2, g_2, f_T) \leftarrow \text{Setup}(1^m, m, n)\). Setup takes as input a security parameter \( \lambda \in \mathbb{N} \), integers \( m \) and \( n \) (\( m \) represents the number of mix-servers and \( n \) represents the number of input ciphertexts) and outputs the following setup parameters: a large prime number \( q (q \gg m, n) \), cyclic groups \( G_1, G_2, G_T \) of order \( q \), generators \( (f_1, g_1, h_1), (f_2, g_2) \) and \( f_T \) of groups \( G_1, G_2 \) and \( G_T \) respectively, and an efficiently computable bilinear map \( e : G_1 \times G_2 \rightarrow G_T \). We assume that the \( \lambda \)-Strong Diffie-Hellman (SDH) assumption \([14]\) holds in groups \((G_1, G_2)\) and that the decisional Diffie-Hellman (DDH) and discrete logarithm (DL) problems are hard in \( G_1 \). We assume that all generators are randomly generated, e.g., as the output of a hash function modelled as a random oracle.

### 3.1 Key cryptographic primitives

#### 3.1.1 (Basic) Boneh-Boyen (BB) signatures (Section 3.1, [14]).

In this signature scheme, the signer chooses its secret key (SK) as \( x \in \mathbb{Z}_q \) and verification key (VK) as \( y \leftarrow g_2^x \). To sign a message \( m \in \mathbb{Z}_q \), it computes \( \sigma = g_1^m \). The signature is verified if \( e(\sigma, yf_T^m) \equiv e(g_1, g_2) \). This scheme is unforgeable against weak chosen message attacks under the n-SDH assumption \([14]\).

#### 3.1.2 BBS+ signatures [4].

In this signature scheme, the signer chooses its SK as \( x \in \mathbb{Z}_q \) and VK as \( y \leftarrow f_2^x \). To sign a message \( m \in \mathbb{Z}_q \), it computes \( c, r \in \mathbb{Z}_q \) and \( S \leftarrow (f_1g_T^m h_1^c)^{1/\sqrt{\lambda}} \) and outputs \( \sigma := (S, c, r) \). The signature is verified if \( e(S, yf_T^m) \equiv e(f_1g_T^m h_1^c, f_2) \). The scheme is unforgeable against adaptively chosen message attacks under the n-SDH assumption \([4]\).

#### 3.1.3 Signatures on committed values.

BBS+ signatures also let one reveal only a Pedersen commitment \([74]\) \( y = g_1^m h_1^c \) to a signer (and a Fok of \( v, r \)) and obtain a signature on the committed value \( v \):

- The requester sends \( y \) and \( \rho_T \leftarrow PK\{v, r\} : y = g_1^m h_1^c \) to the signer.
- The signer verifies \( \rho_T \).

\footnote{For all \( a, b \in \mathbb{Z}_q \) and generators \( g_1, g_2 \) of \( G_1 \) and \( G_2 \) respectively, \( e(g_1^a, g_2^b) = e(g_1, g_2)^{ab} \) and \( e(g_1, g_2) \neq 1_{G_T} \), where \( 1_{G_T} \) denotes the identity element of \( G_T \).}
We denote these DPKs as Traceable mixnets.

3.1.6 In this ZKP, given a Pedersen commitment $\pi(x, y)$ and a set of values $\phi$, a single prover proves that

$$\sigma \equiv (S, c, \hat{r})$$

can be verified by checking if $e(S, yf_2^7) \equiv e(f_4h, \gamma, f_2)$. 

3.1.4 $(m, m)$-threshold secret sharing. We consider the following standard $(m, m)$-threshold secret sharing scheme where Share allows sharing a secret $x$ among $m$ parties and Recons allows its reconstruction by all of them (fewer parties do not learn $x$):

- $(x(k))_{k \in [m]} \leftarrow \text{Share}_m(x \in \mathbb{Z}_q)$ : $(x(k))_{k \in [m-1]} \leftarrow \mathbb{Z}_q$;
- $x \leftarrow \text{Recons}(x(k))_{k \in [m]}$ : $x \leftarrow \sum_{k \in [m]} x(k) \pmod{q}$

A secret sharing scheme is called additive (resp. multiplicative) if $\mathcal{P}_K$ on input its shares $x(k), y(k)$ of secrets $x$ and $y$ respectively can obtain its share of $x+y$ (resp. $xy$) without any additional interaction with other parties. The above scheme is clearly additive. It can also be made multiplicative using Beaver’s trick [27], which employs an input-independent precomputation step and an algorithm Multi such that $\mathcal{P}_K$ can obtain its share of $xy$ as Multi$(x(k), y(k))$.

3.1.5 $(m, m)$-threshold proofs of knowledge. An $(m, m)$-threshold proof of knowledge (also called a distributed proof of knowledge or a DPK) is a protocol between provers $(\mathcal{P}_K)_{k \in [m]}$ and a verifier $\mathcal{V}$ that convinces $\mathcal{V}$ that for a given common input $x$, the provers know secret shares $\omega(k)$ of a secret $\omega$ such that a predicate $p(x, \omega)$ is true. We denote these DPKs as $\mathcal{V} \leftarrow \text{Prove}_m(x \in \mathbb{Z}_q) \leftrightarrow (\mathcal{P}_K \leftarrow \omega(k)_{k \in [m]}) \mathcal{V} \leftarrow \text{Res}_{m-1} \omega(k) \pmod{q}$, where $\mathcal{V}$ accepted the proof. The secrecy guarantee is that an adversary $\mathcal{A}$ controlling $\mathcal{V}$ and all $(\mathcal{P}_K \leftarrow \omega(k))$ for some $k$ cannot learn anything about $\omega(k)$ (and thus $\omega$) [57].

We use DPKs where the predicate $p$ is of the form $\land_{i \in [t]} y_i = \Pi_{i \in [t]} y_i^{(t)}$ for $t, t' \in \mathbb{N}$, public values $y_i, y_j \in G_1, G_2$ or $G_T$ and $\omega \in \mathbb{Z}_q$. These DPKs can be constructed using standard $\Sigma$-protocol techniques [18, 35, 37] if each prover $\mathcal{P}_k$ knows share $\omega(k)$ of each $\omega_j$. We use the following NIZK variant obtained using the Fiat-Shamir heuristic [42]:

- $(\mathcal{P}_K)_{k \in [m]}$: Publish $a_i^k \leftarrow \Pi_{j \in [t')} y_{i,j}^{(t')}$, where $\gamma^{(t')}$ $\mathbb{Z}_q$.
- $(\mathcal{P}_K)_{k \in [m]}$: Compute $a_i \leftarrow \Pi_{k \in [m]} a_i^k$; $c \leftarrow H(p(||(a_i)||_e))$, where $H$ is a cryptographic hash function modelled as a random oracle; $z_i^k \equiv i(k) - c \omega_j (\pmod{q})$. Send $z_i^k$ to $\mathcal{V}$.
- $\mathcal{V}$: Obtain $(z_i^k)_{i \in [t], k \in [m]}, c$, $(z_i^k)_{i \in [t], k \in [m]}$ from $(\mathcal{P}_K)_{k \in [m]}$. Compute $a_i \leftarrow \Pi_{k \in [m]} a_i^k$; $z_i \equiv \sum_{k \in [m]} z_i^k (\pmod{q})$. Check $c \leftarrow H(p(||(a_i)||_e))$ and $\land_{i \in [t]} a_i \equiv y_i^{(t')} \Pi_{i \in [t']} y_{i,j}^{(t')}$.

In this variant, if $\mathcal{A}$ controls $(\mathcal{P}_K)_{k \in [m]}$ but not $\mathcal{V}$, it does not even learn whether the statement was proved successfully or not, because it only sees $a_i^k \equiv \Pi_{j \in [t]} w^{(k)}$ and not $z_i^k$.

3.1.6 $(m, m)$-threshold homomorphic encryption. An $(m, m)$-threshold encryption scheme $\mathbb{E}$ between parties $(\mathcal{P}_K)_{k \in [m]}$ with plaintext space $\mathbb{M}(\mathbb{E})$ and ciphertext space $\mathbb{C}(\mathbb{E})$ is a tuple (Keygen, Enc, TDec), where Keygen is a key generation protocol, Enc is an encryption algorithm and TDec is a threshold decryption protocol, such that for all $x \in \mathbb{M}(\mathbb{E})$, security parameters $\lambda \in \mathbb{N}$, (pk, $(\mathcal{P}_K \leftarrow \omega(k))_{k \in [m]}$) = Keygen($\lambda$), TDec(Enc(pk, x), $(\mathcal{P}_K \leftarrow \omega(k))_{k \in [m]}$) = $\mathbb{I}$. IND-CPA security of these schemes is analogous to the IND-CPA security of vanilla public-key encryption schemes [48], where the adversary controls less than $m$ parties.

We use the following threshold encryption schemes: a) $\mathbb{E}^{\text{EG}}_\lambda$: the threshold El Gamal encryption scheme [34] where $\mathbb{M}(\mathbb{E}^{\text{EG}}_\lambda) = G_1$ and $\mathbb{C}(\mathbb{E}^{\text{EG}}_\lambda) = G_1 \times G_1$, and b) $\mathbb{E}^{\text{EG}}_\lambda$; an optimised threshold Paillier encryption scheme from Damgård et al. [29] where $\mathbb{M}(\mathbb{E}^{\text{EG}}_\lambda) = Z_N^*$ for an RSA modulus $N$ and $\mathbb{C}(\mathbb{E}^{\text{EG}}_\lambda) = Z_N^*$, $\mathbb{E}^{\text{EG}}_\lambda$ is multiplicatively homomorphic in $G_1$: for any two ciphertexts $c_1, c_2 \in G_1$, $\mathbb{E}^{\text{EG}}_\lambda$ is additively homomorphic in $Z_N^*$: for any two ciphertexts $c_1, c_2 \in Z_N^*$, encrypting messages $m_1, m_2 \in G_1$, $c_1c_2$ (their component-wise multiplication in $G_1$) decrypts to the message $m_1 + m_2$ (multiplication in $G_1$). $\mathbb{E}^{\text{EG}}_\lambda$ is additively homomorphic in $Z_N$: for any two ciphertexts $c_1, c_2 \in Z_N^*$, encrypting messages $m_1, m_2 \in Z_N$, $c_1 + c_2$ mod $N^2$ decrypts to the message $m_1 + m_2$ mod $N$. However, we require additive homomorphism in $Z_q$ (a prime order group). Thus, we let $N > q$, interpret messages in $Z_q$ as messages in $Z_N$ and carefully use $\mathbb{E}^{\text{EG}}_\lambda$'s homomorphic addition modulo $q$ to obtain homomorphic addition modulo $q$ (see Section 4.3).

Both $\mathbb{E}^{\text{EG}}_\lambda$ and $\mathbb{E}^{\text{EG}}_\lambda$ are IND-CPA secure, respectively under the DDH assumption in $G_1$ [40] and the decisional composite residuosity (DCR) assumption [73]. Also, both support distributed key generation protocols. For $\mathbb{E}^{\text{EG}}_\lambda$, each party already generates its key shares independently; for $\mathbb{E}^{\text{EG}}_\lambda$, a secure key generation protocol can be designed [30]. Further, the TDec protocols of both schemes provide simulation security, i.e., the adversary’s view in the TDec protocol can be simulated given access to a decryption oracle.

We also use a standard (non-threshold) IND-CPA secure public-key encryption scheme $\mathbb{E}$ on message space $\mathbb{Z}_q$.

3.1.7 Shuffles. Let $\mathbb{E}$ be an $(m, m)$ threshold homomorphic encryption scheme, pk be a public key under $\mathbb{E}$, $e$ be an n-length vector of ciphertexts against pk, $\pi^{(1)}, \ldots, \pi^{(m)} \in \text{Perm}(n)$ be secret permutations of parties $P_1, \ldots, P_m$, where $\text{Perm}(n)$ denotes the space of permutation functions, and $\mathbb{E}^\text{Rand}(pk, c)$ be encryption under $\mathbb{E}$ of ciphertext $c$ with fresh randomness. We let $e' \equiv \mathbb{E}^\text{Shuffle}(\mathbb{E}, pk, e, P_1[\pi^{(1)}], \ldots, P_m[\pi^{(m)}])$ be a shorthand for repeated re-encryption and permutation of $e$ by each of $(\mathcal{P}_K)_{k \in [m]}$ in sequence, such that for all $j \in [n], e'_j = \mathbb{E}((\mathcal{P}_k \leftarrow \omega(k))_{k \in [m]}$, $e_{\pi(j)}$, etc.). The order of parties in the Shuffle protocol denotes that first $P_1$ runs, then $P_2$, etc.

4 OUR CONSTRUCTION

Our traceable mixnet construction extends Camenisch et al.’s single-prover ZKPs of set membership [19] and our novel ZKP of reverse set membership (Section 4.1.2). We explain these protocols first.

4.1 Single prover case

4.1.1 ZKP of set membership [19]. In this ZKP, given a Pedersen commitment $\gamma$ and a set of values $\phi$, a single prover proves
knowledge of \(v, r\) such that \(y = g_1^{h^r_1^v} \) and \(v \in \phi\). The main idea is that the verifier generates a fresh BB signature key pair \(x \overset{\$}{\leftarrow} \mathbb{Z}_q, y \overset{\$}{\leftarrow} g_2^x\) and sends to the prover the verification key \(y\) and signatures \(\sigma_\nu \overset{\$}{\leftarrow} g_1^{\nu \hat{\gamma}}\) on each \(\nu\) \(\in \phi\). The prover chooses a blinding factor \(b \overset{\$}{\leftarrow} \mathbb{Z}_q\) and sends to the verifier a blinded version \(\hat{\sigma}_\nu\) of the signature on the value \(v\) committed by \(y\), as \(\hat{\sigma}_\nu = \sigma_\nu^b = g_1^{\nu \hat{\gamma} b}\). Both then engage in a ZKP of knowledge \(\text{PK}((v, r, b)) : y = g_1^{h^r_1^v} \land e(\hat{\sigma}_\nu, y) = e(g_1, g_2)^v e(\hat{\sigma}_\nu, g_2)^{-b}\), which proves knowledge of a valid signature \((\hat{\sigma}_\nu)^{1/b}\) on the value committed by \(y\) (see Section 3.1.1). This is a property of set membership because if \(y\) does not commit a member of \(\phi\) then the proof fails since the prover does not obtain signatures on non-members of the set and cannot forge them. The scheme is an honest-verifier ZKP of set membership if \(|\phi|\)-Strong Diffie Hellman assumption holds in \((\mathcal{G}_1, \mathcal{G}_2)\) [19].

A nice property of the scheme is that multiple proofs for a set \(\Phi\) of commitments against each \(\phi \in \Phi\) can be given efficiently by reusing verifier signatures. After obtaining signatures \((\sigma_\nu)_{\nu \in \phi}\), the prover can precompute \((\hat{\sigma}_\nu)_{\nu \in \phi}\) in a stage 1. In stage 2, for each commitment \(y \in \Phi\) committing a value \(v \in \phi\), the corresponding \(\hat{\sigma}_\nu\) can be looked up and the ZKP of knowledge can be constructed in \(O(1)\) time. This results in an \(O(|\phi| + |\Phi|)\) amortised complexity for proving set membership for \(\Phi\) commitments.

4.1.2 ZKP of reverse set membership. Now we show how to extend this idea to prove reverse set membership. Note that the BB signatures used above require messages to be in group \(\mathbb{G}_q\). Since commitments are members of \(\mathcal{G}_1\), one cannot use BB signatures to sign members of the set \(\Phi\) of commitments for the reverse set membership proof. Recall, however, that the BBS+ signature scheme [4] lets one present a commitment \(y = g_1^{h^r_1^v}\) along with \(\rho_y := \text{NIZKP}((v, r)) : y = g_1^{h^r_1^v}\) to the signer and obtain a BBS+ signature on the value \(v\), without leaking \(v\) to the signer. We exploit this property for the reverse set membership proof.

Our reverse set membership verifier generates fresh BBS+ signature key pairs \(x \overset{\$}{\leftarrow} \mathbb{Z}_q, y \overset{\$}{\leftarrow} f^x\) and sends quasi-signatures \(\hat{\sigma}_\nu := (S, c, r) \leftarrow (f g_1^{r \hat{\gamma}} y^{\frac{1}{b}}, c, r)\) for each \(y = g_1^{h^r_1^v} \in \Phi\), after verifying \(\rho_y\). If the prover knows commitment randomness \(r\) for each \(y \in \Phi\), it can use \(\hat{\sigma}_\nu\) to derive a valid BBS+ signature \(\sigma_\nu \leftarrow (S, c, r := r + r) \leftarrow (f g_1^{r \hat{\gamma}} y^{\frac{1}{b}}, c, r + r)\) on the committed value \(v\) and store \(\sigma_\nu\) indexed by \(v\). To prove that a given \(v\) is committed by some commitment in \(\Phi\), the prover looks up \(\sigma_\nu\) in \(O(1)\) time, blinds each component of \(\sigma_\nu\) to obtain a blinded signature \(\tilde{\sigma}_\nu\), and proves knowledge of a BBS+ signature on \(v\) by revealing only \(\hat{\sigma}_\nu\) to the verifier. This is a proof of reverse set membership because the prover can obtain valid BBS+ signatures only on commitments by \(y \in \Phi\) and cannot forge it for \(v\) if no \(y \in \Phi\) committed \(v\). This protocol also enjoys \(O(|\phi| + |\Phi|)\) amortised complexity for multiple proofs for each \(v \in \phi\) against the same set of commitments \(\Phi\). See Appendix A for the detailed protocol.

4.2 Overview of our construction

We now give an overview of our traceable mixnet construction (see Section 4.3 for detailed protocol steps). In our construction, senders send threshold encryptions of their sensitive values as input ciphertexts which get shuffled by a series of mix-servers and eventually decrypted just like a standard re-encryption mixnet [53]. However, in addition, the senders also upload Pedersen commitments to the encrypted values along with the input ciphertexts and secret-share the commitment openings among the mix-servers (see Figure 6a). The mix-servers use these shares to distributedly answer BTraceIn and BTraceOut queries via our distributed and batched ZKPs of set membership and reverse set membership (DB-SM and DB-RSM).

A DB-SM protocol for index sets \(I, J\) allows the mix-servers to prove for each uploaded commitment at an index \(i \in I\) that it commits a value in the set of output plaintexts at indices \(J\); the querier’s output is the indices \(I^* \subseteq I\) where the statement holds. As in the single-prover ZKP of set membership, the querier is asked to provide BB signatures on the set of plaintexts at indices \(J\). Note, however, that in the single prover case, the prover knows the commitment’s committed value \(v_i\), which allows it to look-up its blinded signature \(\hat{\sigma}_i\) in \(O(1)\) time. In the distributed mixnet setting, no prover (mix-server) knows the committed value, randomness or the permutation between the list of commitments and plaintexts. The challenge is to efficiently identify the blinded signature on a given commitment’s committed value without letting any set of less than \(m\) mix-servers or the querier learn these secrets.

To solve this challenge, the querier signatures are encrypted and shuffled by the mix-servers in the reverse direction as the forward mixnet shuffle — following the inverse of the mixnet permutation — and are homomorphically blinded by random blinding factors before decryption (see Figure 6b). This process produces the blinded signatures next to the corresponding input commitments. To prove set membership, the mix-servers use the blinding factors and the commitment opening shares sent by the senders to jointly prove knowledge of a BB signature on the committed value via a DPK.

A technical complication in the above outline is that in addition to supplying BB signatures for each plaintext at an index \(j \in J\), the querier must also provide invalid signatures for plaintexts at indices \(j \notin J\). These invalid (“fake”) signatures should also be encrypted, reverse-shuffled and blinded in the same way as the valid signatures. Without these invalid signatures, the blinded signatures would appear against exactly the input list commitments that committed a plaintext in set \(J^*\). This would reveal even for commitments outside set \(J^*\) whether they committed a plaintext in set \(J^*\) or not, violating our secrecy definition. With these invalid signatures, the DPKs pass only for commitments in \(J^*\) that commit a value in set \(J^*\) but no information is revealed for commitments outside \(J^*\).

A DB-RSM protocol for index sets \(I, J\) allows the mix-servers to prove for each output plaintext at an index \(j \in J\) that it is committed by a member of the set of commitments at indices \(I\); the querier’s output is the indices \(J^* \subseteq J\) where the statement holds. As in the single-prover ZKP of reverse set membership, the querier is asked to provide BBS+ quasi-signatures on the set of commitments at indices \(I\) (and invalid quasi-signatures for indices \(\{n\} \setminus I\)). These quasi-signatures are encrypted, homomorphically converted to encrypted BBS+ signatures and then shuffled by the mix-servers in the forward direction — following the mixnet permutation — to obtain encrypted BBS+ signatures next to the corresponding plaintexts (see Figure 6c). These encrypted signatures are then homomorphically blinded...
before decryption and the mix-servers use the blinding factors to provide a DPK of a BBS+ signature on the corresponding plaintext.

Even when all the mix-servers are cheating, they cannot make the proofs pass for an incorrect entry and include, e.g., a commitment that did not commit a value in set \( \sigma'_i \) in DB-SM output. However, this does not prevent them from deliberately failing proofs for commitments that actually committed a value in \( \sigma'_i \), producing a smaller-than-correct output set and violating Definition 3. Thus, we run DB-SM against both \( J \) and \( [n] \setminus J \) in a BTraceIn call and make the querier abort if proofs against both the runs failed for some commitment (similarly for BTraceOut).

4.3 Technical details

Figure 6 shows our traceable mixnet construction in detail. This construction preserves secrecy even in the honest-but-curious (HBC) setting; see Section 4.3.1 for malicious security. We use the threshold \( E_{Pa}^h \) scheme to create ciphertexts \( e_i \), the \( E_{EG} \) scheme to reverse-shuffle BB signatures in DB-SM and both \( E_{EG} \) and \( E_{Pa}^h \) to forward-shuffle BBS+ signatures in DB-RSM. Further, we use a standard public-key encryption scheme \( E \) for securely sending shares of commitment openings to the individual mix-servers. The Keygen step creates public/private keys for all these schemes. Secret keys for \( E_{EG} \) and \( E_{Pa}^h \) are shared among the mix-servers and secret keys for \( E \) for each public key is held by individual mix-servers. Via the Enc algorithm, senders upload ciphertexts \( e_i \) encrypting their secret values, along with commitments \( y_j \) and encryptions of secret shares of the commitment openings for each mix-server. They also upload proofs of knowledge \( \rho_f \) of the commitment openings and encryptions \( e_r \) of commitment randomnesses to enable the DB-RSM proofs. During the Mix protocol, the mix-servers shuffle and threshold-decrypt \( e_i \) to produce permutated plaintexts \( \sigma'_j = (\sigma_i(j))_{j \in [n]} \), where \( \pi \) is composed of secret permutations \( \pi(k) \) of each mix-server \( M_k \). Each \( M_k \) stores \( \pi(k) \) and decryptions of commitment opening shares to jointly answer BTraceIn/BTraceOut queries via DB-SM/DB-RSM.

Figure 8 shows our DB-SM protocol. In stage 1, the querier \( Q \) publishes valid BB signatures \( \sigma'_j \) on \( \sigma' \) for each \( j \in J \) and invalid signatures (a fixed group element) for \( j \notin J \). These “signatures” are encrypted under \( E_{Pa}^h \) and shuffled by the mix-servers in the reverse direction from \( M_9 \) to \( M_1 \), with each \( M_k \) using permutation \( (\pi(k))^{-1} \) to produce encrypted signatures \( e_{\sigma'_j} \) on the value \( \alpha_i \) committed by \( y_j \) (the encrypted signature being valid only if \( \alpha_i \in \sigma'_j \)). The mix-servers then use the multiplicative homomorphism of \( E_{EG} \) to jointly obtain encryptions \( e_{\sigma_i} = e_{\Sigma_{k \in [m]} \hat{b}_k(i)} \), where each \( M_k \) contributes binding factors \( \hat{b}_k(i) \mapsto Z_q^{*} \). The plaintext binding signatures \( \sigma_i \) are finally obtained by threshold decryption of \( e_{\sigma_i} \) and published alongside \( y_j \). In stage 2, \( \sigma_i \) for each \( y_i \) is looked up in \( O(1) \) time. A DPK of a BB signature on the value committed by \( y_i \) is given by proving joint knowledge of the commitment openings and bindings to a valid signature. This DPK has the format of Section 3.1.5 and can be given efficiently since each \( M_k \) knows additive shares \( v^{(k)}(i) \) for the commitment openings and \( d^{(k)}(i) \) for the binding factors. All indices for which the DPK passed are included in \( Q \)'s output \( I^* \). If \( Q \) is not corrupted, \((M_{\hat{k}})_{k \in [m]} \) do not learn \( I^* \) since they do not learn which DPK passed. The amortised complexity of the entire protocol is \( O(n) \).

Figure 9 shows our DB-RSM protocol. Note that to obtain BBS+ signatures on committed values, knowledge of commitment openings must be first shown. For this, \( Q \) checks the NIZKS \( \rho_f \) uploaded by the senders for each \( i \in I \). Then \( Q \) sends valid BBS+ quasi-signatures \( \sigma_i \) on \( y_i \) for \( y_i \in I \) and invalid ones for \( y_i \notin I \). \( Q \) encrypts each component \( (S_i, e_{\sigma_i}, e_{\sigma_i}) \) independently, using \( E_{EG} \) for \( S_i \) and \( E_{Pa}^h \) for \( e_{\sigma_i} \) and \( e_{\sigma_i} \). Then \( Q \) sends the sender-uploaded encryptions \( e_{\sigma_i} \), encrypted BBS+ signatures on the committed values are derived by homomorphically adding the commitment randomness \( r_i \) to the signature's \( r_i \) component. Thus, encrypted (valid and invalid) BBS+ signatures \( e_{\sigma_i} \) for \( i \in I \) are obtained next to each commitment \( y_i \) in the input list.

To obtain blinded BBS+ signatures on plaintext values in the permuted list \( \sigma' \), each component \( e_{\sigma_i}(e_{\sigma_i}, e_{\sigma_i}) \) of the BBS+ signature is re-encrypted individually using encryption schemes \( (E_{EG}^h, E_{Pa}^h) \) respectively and shuffled in the forward direction. The permuted and re-encrypted signatures \( (e_{\sigma_i}, e_{\sigma_i}, e_{\sigma_i}) \) are individually blinded to obtain \( (e_{\sigma_i}, e_{\sigma_i}, e_{\sigma_i}) \), which are individually threshold-decrypted to obtain blinded BBS+ signatures \( \sigma'_j = (\hat{S}_j, \hat{e}_j, \hat{r}_j) \) alongside \( \sigma'_j \).

Note that \( E_{Pa}^h \) used for encrypting \( e_{\sigma_i} \) is homomorphic in group \( Z_q \), which induces addition modulo \( N \) in the plaintext space, not modulo \( q \). Thus, blinding by binding factors drawn from \( Z_q \)
would not be perfectly hiding. To circumvent this issue, we follow 

\[ r \in \mathbb{Z}_q \rightarrow g^{r_k} \rightarrow \text{NIZKP}(r, y = g^{r_k}) \]

\[ c \rightarrow \text{Enc}(\text{mpk}, r) \rightarrow \text{interpret } r \text{ as an element of } \mathbb{Z}_N \]

\[ (s^{(k)})_{k \in \mathcal{M}} : \text{Share}(\text{mpk}(r), (r_k)_{k \in \mathcal{M}} : \text{Share}(\mathcal{M}, r)) \]

\[ (e^{(k)})_{k \in \mathcal{M}} : \text{Enc}(\text{mpk}, r_k)_{k \in \mathcal{M}} \]

\[ \text{output } e = (e_r, (e^{(k)}), (k)_{k \in \mathcal{M}}) \]

\[ \text{Mix}(\text{mpk} = (., \text{mpk}), e = (e_r, (\text{ev}^{(k)}), (e^{(k)}), (k)_{k \in \mathcal{M}})) \]

\[ (M_k[[\text{mk}^{(k)}]] = (s^{(k)}, (\text{sk}^{(k)}), (k)_{k \in \mathcal{M}})_{k \in \mathcal{M}}) \]

\[ \text{BTraceIn}(\text{mpk} = (., \text{mpk}), e = (e_r, (r', i)), i \in \mathcal{M}) \]

\[ (M_k[[\text{mk}^{(k)}]] = (s^{(k)}, (\text{sk}^{(k)}), (k)_{k \in \mathcal{M}})_{k \in \mathcal{M}}) \]

\[ \text{BTraceOut}(\text{mpk} = (., \text{mpk}), e = (e_r, (r', i)), i \in \mathcal{M}) \]

\[ (M_k[[\text{mk}^{(k)}]] = (s^{(k)}, (\text{sk}^{(k)}), (k)_{k \in \mathcal{M}})_{k \in \mathcal{M}}) \]

\[ \text{Participants: Mix-servers } (M_k[[\text{mk}^{(k)}]]), \text{Querier } Q \]

\[ \text{Common input: } (pk_{EG}, y = (y_j)_{j \in \mathcal{M}}, \sigma = (\sigma_j)_{j \in \mathcal{M}}) \]

\[ \text{Stage 1:} \]

\[ \text{Signature generation} \]

\[ Q : \rightarrow \mathbb{Z}_q \rightarrow y \rightarrow \sigma_j \]

\[ \sigma' \rightarrow (\sigma_j') \rightarrow (y_j') \rightarrow \text{other if } j \rightarrow \text{other otherwise} \]

\[ \text{Publish } y, \sigma', \sigma'' \]

\[ \text{Shuffling} \]

\[ \epsilon = \text{Shuffle}(pk_{EG}, pk_{GC}, \sigma) \rightarrow \mathbb{Z}_q \rightarrow (y_j') \rightarrow \mathbb{Z}_q \rightarrow \text{other if } j \rightarrow \text{other otherwise} \]

\[ \text{Homomorphic binding} \]

\[ (M_k[[\text{mk}^{(k)}]] : \rightarrow \mathbb{Z}_q \rightarrow (\text{perm}^m) \rightarrow \mathbb{Z}_q \rightarrow \text{other if } j \rightarrow \text{other otherwise} \]

\[ \text{Threshold decryption} \]

\[ \sigma \rightarrow \mathbb{Z}_q \rightarrow \text{other if } j \rightarrow \text{other otherwise} \]

\[ \text{Publish } \hat{\sigma} \]

\[ \text{Stage 2:} \]

\[ Q' \rightarrow \mathbb{Z}_q \rightarrow \text{other if } j \rightarrow \text{other otherwise} \]

\[ \text{Endfor output } Q'' \]

Figure 8: Protocol DB-SM (see summary in Figure 6b).
5 SECURITY ANALYSIS

Theorem 1 (Completeness). Let $\Pi_{T\text{M}}$ be the protocol of Figure 7. $\Pi_{T\text{M}}$ is complete (Definition 2). (Proof in Appendix C.1).

Theorem 2 (Soundness). Under the DL assumption in $G_1$ and n-SDH assumption in $(G_1, G_2)$ [14], $\Pi_{T\text{M}}$ is sound (Definition 3).

Proof sketch (full proof in Appendix C.2): If an adversary won $\text{Exp}_{\text{soundness}}$ then $Q$ must have output either 1) a wrong $c_T$ in a $B\text{TraceOut}(I, J, Q)$ query, or 2) a wrong $\sigma'_J$, in a $B\text{TraceOut}(I, J, Q)$ query. Case 1 implies that either a) $c_T$ included a $c_i \in c_T$, not encrypting a plaintext in $\sigma'_J$, or b) it included a $c_i \in c_T$, encrypting a plaintext in $\sigma'_J$. Case 2a directly reduces to breaking the soundness of DB-SM for $y_J$, against set $\sigma'_J$. Further, since each $c_i \in c_T$, $c_T$ must be in $c_T$ for $Q$ to not abort, case 1b also reduces to breaking the soundness of DB-SM (because if $c_i \notin c_T$, encrypts a plaintext in $\sigma'_J$, it does not encrypt a plaintext in $\sigma'_J$ due to distinctness of $\sigma'_J$s, but $c_i \in c_T$).

Case 2 similarly reduces to breaking the soundness of DB-RSM. 

Soundness of DB-RSM for $I, J$: The DPK proves that $(M_k \in \{0, 1\})$ knowing binding factors to unblind $\sigma_I$ to a valid BBS signature on the value $v_I$ committed by $y_I$ under $Q$'s fresh public key. Since $Q$ issued valid signatures only for plaintexts in $\sigma'_J$, passing the DPK if $v_j \notin \sigma'_J$ requires $(M_k \in \{0, 1\})$ to forge a BBS signature under $Q$'s public key, which is hard under the stated assumptions.

Soundness of DB-RSM for $I, J$: The DPK proves that $(M_k \in \{0, 1\})$ knowing binding factors to unblind $\sigma'_J$ to a valid BBS signature on $\sigma'_J$ under $Q$'s fresh public key. Since $Q$ issued valid signatures only for commitments in $y_J$, deriving a signature on a plaintext not committed in any commitment in $y_J$ reduces to breaking the soundness of the BBS-1 scheme for obtaining signatures on committed values, which is hard under the stated assumptions.

Theorem 3 (Secrecy). Under the IND-CPA security of $E$, the DDH assumption in $G_1$ and the DCR assumption [73], $\Pi_{T\text{M}}$ protects secrecy (Definition 4) against HBC adversaries in the random oracle model.

Proof sketch (full proof in Appendix C.3): We need to show that for any pair of values $v_i, v_j$, no PPT adversary controlling $(M_k \in \{0, 1\})$ and $Q$ can distinguish between world 0 where $S_{ih}$ sends $v_i$ and $S_{hj}$ sends $v_j$, and world 1 where this order is reversed, if the assert conditions in all $O\text{TraceIn}$ and $O\text{TraceOut}$ calls of $\text{Exp}_{\text{sec}}$ are respected (and all adversarial parties are honest-but-curious).

We do this by simplifying world $b$ for each $b \in \{0, 1\}$ by the following sequence of indistinguishability arguments:

- For each $i \in \{i_0, i_1\}$, NIZK $\text{Pr}_{\text{r}_i}$ can be simulated, shares $p_{\text{Pr}_{\text{r}_i}}(k, r_i)$ given to $(M_k \in \{0, 1\})$ can be replaced by random elements drawn from $Z_q$, and encryptions $\text{ev}_{\text{Pr}_{\text{r}_i}}(k, r_i)$ can be replaced by encryptions of 0 by the IND-CPA security of $E$.

Threshold decryption protocols for $\text{Enc}_{\text{EG}}$ and $\text{Enc}_{\text{Pa}}$ do not reveal any information beyond the decryption output, which can be
obtained by permuting the list of input values, where \( S_k \) and \( S'{k} \)’s input values are \( o_{ik} \) and \( v_{ik} \), respectively. With the decryption oracles now eliminated, all \( E_{\text{EC}}^k \) and \( E_{\text{Pa}}^k \) encryptions/re-encryptions can be replaced by encryptions of dummy values, by the IND-CPA security of \( \text{E}_{\text{EC}}^k \) and \( \text{E}_{\text{Pa}}^k \) under the DDH and DCR assumptions respectively.

- The only information leaked during a DPK for \( S_k/S'{k} \)’s commitment/plaintext is whether the DPK passed or not, but the assert conditions ensure that this information is the same in both the worlds. Thus, these DPKs can be simulated by a ZK simulator that does not know which specific world it is in.

- For \( i \in \{i_0, i_1\} \), \( e_{\text{B}}^{(k)} \), \( e_{\text{r}}^{(k)} \) can be replaced by encryptions of 0 since the encrypted values \( o_{ij}^{(k)}, r_{ij}^{(k)} \) are not used anywhere anymore (they were used in the DPKs/encryptions earlier). Similarly, \( y_{i_0}, y_{i_1} \) can now be replaced with commitments of 0.

- The blinded signatures corresponding to \( S_k \) or \( S'k \)’s commitments or plaintexts during DB-SM and DB-RSM can now be replaced with random group elements because they are not used anywhere anymore and the binding factors chosen by \( M_k \) are random (note: for the \( c, r \) components of the BBS+ signature, this holds because of flooding with large binding factors).

The two worlds obtained now are indistinguishable because now only the mixnet output list \( o' \) depends on \( v_k \) and \( v_i \) and this list is identically distributed in the two worlds because of the uniformly chosen permutation \( \pi(i)^{(k)} \) by \( M_k \).

**Theorem 4 (Output secrecy).** Under the same assumptions as Theorem 3, \( \Pi_{\text{TM}} \) protects output secrecy (Definition 5). (Proof in Appendix C.4).

In Appendix C.5, we also sketch a proof that after applying the additional steps mentioned in Section 4.3.1 and Appendix B, our construction protects secrecy against general malicious adversaries.

### 5.1 Privacy risk analysis of query outputs

Theorem 3 guarantees that a traceable mixnet does not reveal any information beyond the output of the BTraceIn/BTraceOut queries. To evaluate the privacy risk impact of these outputs themselves, we provide a mechanism in Appendix D to statically analyse information leaked by a given set of queries. Specifically, given a set \( Q \) of proposed Traceln/TraceOut queries in an application, the mechanism outputs information potentially leaked by them: a) for each \( i \in [n] \), the smallest set \( J_{\min} \) representing potential plaintexts that \( c_i \) might encrypt and b) for each \( j \in [n] \), the smallest set \( I_{\min} \) representing potential ciphertexts that \( o'_{ij} \) might decrypt from \( I_{\min} = [n] \) denotes that queries in \( Q \) reveal no additional information about \( c_i \); likewise for \( I_{\min} \). The \( J_{\min} / I_{\min} \) information can then directly be used to analyse application-level security.

### 6 IMPLEMENTATION AND BENCHMARKS

We implemented a proof-of-concept for our traceable mixnet construction, with the primary goal of evaluating its runtime query performance. Given this focus, we mainly implemented the DB-SM and DB-RSM protocols. This allows us to directly estimate the total time for BTraceIn or BTraceOut queries as that of two DB-SM or DB-RSM invocations. However, an optimisation where the DB-SM calls for \( J \) and \( [n] \setminus J \) (similarly for DB-RSM) are combined to a single call as follows considerably improves this estimate: \( Q \) generates signature key pairs \( x, y \) for \( J \) and \( x_C, y_C \) for \( [n] \setminus J \) and sends signatures using key \( x_C \) for \( j \not\in J \) instead of invalid signatures, thus avoiding extraneous shuffling/decryption of invalid signatures.

We used standard threshold ElGamal encryption [34] for \( \text{E}_{\text{EC}}^k \) and Damgård et al.’s [29] optimised threshold Paillier encryption for \( \text{E}_{\text{Pa}}^k \). For key generation, we implemented a simplified protocol where a trusted dealer distributes key shares to the mix-servers. Secure distributed key generation is trivial for \( \text{E}_{\text{EC}}^k \) but requires a special protocol [30] for \( \text{E}_{\text{Pa}}^k \). However, our simplification is justified as key generation is a one-time setup step that can be executed ahead of time and does not affect runtime query performance.

We also implemented all the steps mentioned in Section 4.3.1 and Appendix B to evaluate performance in the realistic malicious setting. We used the following techniques to optimise these steps: 1) standard Σ-protocol techniques to let senders efficiently prove knowledge of their uploaded ciphertexts and mix-servers prove knowledge of blinding factors during homomorphic blinding; 2) batch-verification techniques [7, 41] to let each mix-server efficiently verify the querier’s signatures/quasi-signatures and the correctness of other mix-servers’ decryption shares during threshold decryption; and 3) permutation commitment-based techniques of [77, 83] to let each mix-server efficiently prove that they created their shuffles consistently during Mix, DB-SM and DB-RSM.

We implemented all sender proofs of knowledge required to achieve our soundness and secrecy requirements. However, we did not implement proofs of well-formedness of input ciphertexts (Section 2): proofs that \( y_i \) commit the same value as encrypted by \( e_i \) and that \( e_i, e_r \) encrypt values in the range \([0, q] \). This is primarily due to our focus on runtime query performance and because these costs are incurred by individual senders and mix-servers as and when senders send their data. Our proofs of knowledge incur \( \sim 0.15 \) seconds cost per-sender for both the sender and the mix-servers. With standard Σ-protocol techniques and FO commitments [43] for proving equality of committed and encrypted values and [16, 64] for range proofs, the total time is expected to remain \( <1 \) s per sender.

Finally, we did not implement the authenticated broadcast channel. Since our uploaded datasets are moderate in size and the round complexity is small, we do not expect this to be a bottleneck.

Our implementation [1] is based on the Charm library [3] with the PBC backend [66] for pairing operations. We chose the BN254 curve [6, 56] to instantiate pairing groups \( (G_1, G_2, G_T) \), which gives a group order \( q \) of 254 bits. We attempted to minimise the number of pairing operations wherever possible and used pre-computation of powers of fixed bases to speed-up exponentiation operations.

We ran all our benchmarks on an Intel(R) Xeon(R) W-1270 CPU @ 3.40GHz with 64 GB RAM on a single core. Figure 10 shows the performance of our DB-SM/DB-RSM implementation in the worst case. We slightly simplified [29] by skipping the factorial trick used for general \( t \)-out-of-\( m \) threshold cryptography in RSA groups, focusing on the \( m \)-out-of-\( m \) case.

Senders do not need to prove distinctness of their encrypted messages because duplicate values can be detected at the mixnet output and the offending senders could be identified (by, say, a BTraceIn query) on demand.

With NFS attacks [60], the security of BN254 curves has dropped to 100-110 bits [39, 67]. We are limited to BN254 because of our chosen Charm library, but we estimate that the overall performance hit in per-mix-server time on switching to a more secure curve BLS12-381 [39] is \(<1.3x \), given that BLS12-381 operations are \( \approx 2x \) slower than BN254 [26] and that curve operations predominantly only affect our stage 2 DPKs.
<table>
<thead>
<tr>
<th>$M_j$ and Q times (sec) in DB-SM and DB-RSM for different $n$ and $m$</th>
<th>$n = 10^2$</th>
<th>$n = 10^3$</th>
<th>$n = 10^4$</th>
<th>$n = 10^5$</th>
<th>$n = 10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_2$</td>
<td>35</td>
<td>349</td>
<td>3400</td>
<td>49</td>
<td>490</td>
</tr>
<tr>
<td>$M_4$</td>
<td>23</td>
<td>225</td>
<td>2268</td>
<td>23</td>
<td>227</td>
</tr>
<tr>
<td>$M_6$</td>
<td>20</td>
<td>190</td>
<td>2000</td>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>$M_8$</td>
<td>62</td>
<td>610</td>
<td>6200</td>
<td>62</td>
<td>620</td>
</tr>
<tr>
<td>$Q$</td>
<td>168</td>
<td>1600</td>
<td>20000</td>
<td>240</td>
<td>2400</td>
</tr>
<tr>
<td>$Q$</td>
<td>122</td>
<td>1188</td>
<td>1198</td>
<td>129</td>
<td>1291</td>
</tr>
<tr>
<td>$Q$</td>
<td>62</td>
<td>610</td>
<td>6200</td>
<td>62</td>
<td>620</td>
</tr>
<tr>
<td>$Q$</td>
<td>4160</td>
<td>43100</td>
<td>448920</td>
<td>4160</td>
<td>43100</td>
</tr>
</tbody>
</table>

Detailed breakdown for $n = 10^2$ and $m = 4$

- Size of $n$ input ciphertexts: 200 MB
- $(M_2)$ Mixing (Fig. 7): 343 s
- (Q) Generating $n$ BB signatures/encryptions: 8.3 s
- (M1) Verifying $n$ BB signatures/encryptions: 15 s
- (M1) Re-encryption of encrypted signatures: 7.3 s
- (M1) Proof-of-shuffle of encrypted signatures: 139 s
- (M1) Homomorphic binding of encrypted signatures: 130 s
- (M1) Threshold decryption of encrypted signatures: 22 s
- (M1) Generating $n$ DPK proofs for $p_B$: 170 s
- (Q) Verifying $n$ DPK proofs for $p_B$: 190 s
- Size of $n$ BB signatures: 0.3 MB
- Size of $n$ DPK proofs for $p_B$: 3.8 MB

Figure 10: Performance of DB-SM and DB-RSM ($n$: number of input ciphertexts, $m$: number of mix-servers). $M_j$ and $Q$ denote per-mix-server and querier times respectively; collab-zkSNARK denotes estimated per-prover times in collaborative zkSNARKs [72].

In the case when $I = J = |n|$, with the overall mix-server and querier times for different $n$ and $m$ at the top and the detailed breakdown for $n = 10000$ ciphertexts and $m = 4$ mix-servers at the bottom. All reported values are averages over 3 runs. The deviation from the average in any run is $<1.5\%$. We report per-mix-server times, which accurately capture real-world latencies as the heavy operations like proofs-of-shuffle, homomorphic binding, threshold decryption and stage 2 DPKs can be run in parallel by the mix-servers. The only sequential operation is re-encryption, but it is a negligible fraction of other steps. We also report timings for the HBC case to highlight the overhead introduced by the malicious security steps.

Our ZKPs are practical for offline batch processing tasks, finishing within an hour and requiring moderate amount of data to be published for 10000 ciphertexts. They scale linearly with $n$. The scale-up with $m$ is constant for the HBC mix-server time and for the querier, but in the malicious case, each mix-server needs to verify other mix-servers’ output, which leads to a $\sim 1.4x$ increase from $m = 2$ to $m = 4$. Our main bottlenecks are expensive pairing and exponentiation computations in our DPKs and expensive Paillier operations in DB-RSM. The additional proofs for malicious security add an overhead of roughly 1.5-2x over the HBC case where these steps are skipped.

There exists a high degree of task parallelism in our construction, since DPKs in stage 2 are independent of each other and stage 1 operations incur at most constant communication overhead. Thus, we expect significant speedups if each mix-server and the querier are given multiple cores. With a moderate parallel cluster of 100 nodes, recovery for an election with $10^6$ votes can thus be performed within a few hours. Further, if the signer in stage 1 of our ZKPs could be a separate trusted entity different than the querier and pre-sign all set entries, then our ZKPs become completely non-interactive and only incur stage 2 costs for verification.

**Comparison.** The only technique comparable to our distributed setting is collaborative zkSNARKs [72]. We indirectly estimate our performance against them by employing a thumbule given by [72] that the per-prover time in a collaborative zkSNARK, assuming each prover already has a share of the SNARK witness, is $\sim 2x$ the prover time in the corresponding single-prover zkSNARK. Thus, we implemented zkSNARKs for $p_{SM\text{-Acc}}$ and $p_{RSM\text{-Acc}}$ via Merkle accumulators (see Section 1.2.4). We used the ZoKrates toolchain [38] and the Groth16 proof system [50]. We used the Baby Jubjub curve [81], which has similar order as BN254 and allows efficient computation of Merkle hashes for commitments for $p_{RSM\text{-Acc}}$. For creating Merkle hashes, we used the Poseidon hash function [49].

Figure 10 (top) also shows the per-prover times in a collaborative zkSNARK approach estimated as above (averaged over 3 runs with deviation $<1\%$). We find the prover time for one $p_{SM\text{-Acc}}$ or $p_{RSM\text{-Acc}}$ proof against a set of size 10000 is $\sim 2.15$ s, which when scaled to 10000 commitments takes $\sim 21500$ s. From this, the per-prover time in collaborative zkSNARKs is estimated to be $\sim 43000$ s, as shown in the column for $n = 10^4$ (for any $m$). This estimate is conservative as it does not count the time taken to securely distribute shares of the SNARK witness among the collaborating provers. This makes our DB-SM and DB-RSM proofs $\sim 60x$ and $\sim 18x$ faster than collaborative zkSNARKs. We note that our verification times (200 s and 620 s for $n = 10000$) are slower than zkSNARKs’ $\sim 50$ s, but the dominant prover times in zkSNARKs imply that our techniques still bring drastic overall improvements.

We also ran Benarroch et al.’s [12] official implementation [52], which proves $p_{SM\text{-Acc}}$ for a single prover (but not $p_{RSM\text{-Acc}}$). This takes $\sim 2200$ s for 10000 $p_{SM\text{-Acc}}$ proofs, excluding the $O(n^2)$ time taken to generate the RSA accumulator witnesses (see Section 1.2.4).

**7 Conclusion**

We introduced and formalised the notion of traceable mixnets, extending traditional mixnets to provably answer useful subset queries in zero knowledge. We also proposed a traceable mixnet construction using novel distributed ZKPs of set membership and reverse set membership, which are useful in other settings too. We implemented these ZKPs and showed that they are significantly faster than the state-of-the-art techniques. Nevertheless, our current implementation is practical only for offline batch processing such as recovery in elections. Constructing traceable mixnets for real-time privacy applications is a challenging open problem.
ACKNOWLEDGMENTS
We wish to thank Rohit Vaish, Kabir Tomer and Mahesh Sreekumar Rajasree for their helpful discussions and comments, and Aarav Varshney for assistance in setting up the benchmarks. Our gratitude also extends to the anonymous reviewers whose suggestions significantly improved the manuscript’s presentation.

This research was partially supported by the Pankaj Gupta Chair in Privacy and Decentralisation. Prashant Agrawal received additional support from the Pankaj Jalote Doctoral Grant, and Abhinav Nakarmi was supported by a research grant from the MPhasis F1 Foundation.

REFERENCES
A SINGLE PROVER REVERSE SET MEMBERSHIP

Figure 11 shows the ZKP of reverse set membership in the single prover case. Note that in this protocol, the prover knows commitment openings for each \( y \in \Phi \) (not only for the commitment \( \psi \)).

\[
\text{Participants: Prover } \mathcal{P}, \text{ Verifier } \mathcal{V}
\]

**Common input:** \( \Phi \in \mathbb{G}^n (\text{set of commitments}), \psi \in \mathbb{Z}_q \)

**\( \mathcal{P}' \)'s input:** For each \( x, \psi \in \Phi \) indexed by \( \psi \), the verifier \( \mathcal{V} \) sends \( y \in \mathbb{G}_1^* \).

**\( \mathcal{V}' \)'s output:** if \( \mathcal{V} \) accepts else 0

**Stage 1:**

1. For each \( y \in \Psi \) do:
   - \( \text{abort if } \text{NIZKVer}(\psi, y) \notin \mathcal{V} \)
2. For each \( y \in \Phi \) do:
   - \( s_y \leftarrow \mathcal{V} \left( f_1^h y \right)^{1/\tau} \)
3. Each \( s_y \) gets a \( NIZKPK \) verifiable by \( \mathcal{V} \) indexed by \( \psi \).

**Stage 2:**

1. For each \( y \in \Phi \):
   - \( \text{res } \leftarrow \mathcal{P}_K((b_2, b_2, b_2, \ldots, b_2) ; (y_2 = y_b^1 b_2^0)) \)

- **output res**

**Figure 11:** ZKP of reverse set membership \( \text{PRSM}(\Phi, \psi) \) := \( \text{PK}(\{ \psi \} ; y = y_2^1 b_2^0 \land y \in \Phi) \), if the prover knows openings for each \( y \in \Phi \).

**B FROM HONEST-BUT-CURIOUS TO MALICIOUS MODEL**

In this section, we mention steps required to derive secrecy (Definition 4) in the general case when the adversary allows the corrupted parties (all-but-two senders, all-but-one mix-servers and the querier) to deviate from the protocol:

- Each sender \( S_i \) must attach NIZK proofs of knowledge \( (p_{c_i}, p_{c_{1i}}, (p_{c_{1k}}, p_{c_{1k}}, k \in [m]), (p_{c_{1k}}, p_{c_{1k}}, k \in [m]), k \in [m]) \) for plaintexts encrypted by commitments \( (\psi, \psi) \in \mathcal{P}_K \). Each \( (M_1, k) \in \mathcal{M}_1 \) must verify these proofs before processing anything.

- For encryptions \( c_{i, \psi} \) in DB-SM and \( c_{i, \psi}, c_{i, \psi} \) in DB-RSM, the querier must publish their randomnesses and each \( (M_1, k) \in \mathcal{M}_1 \) must verify that they were created correctly. Each \( (M_1, k) \in \mathcal{M}_1 \) must also verify that \( Q \) gave valid signatures/quotient-signatures for each element in the requested set and invalid ones for its complement. Note that this also involves verifying that \( c, P \) in DB-RSM contain only elements in the range \( [0, q] \).
Each \((M_k)_{k \in [m]}\) must provide proofs of correct shuffle in all the shuffle protocols. In a proof of shuffle, \(M_k\) for a given input ciphertext list \(e\) and output ciphertext list \(e'\) proves that \(e'\) is a permutation and re-encryption of \(e\) under a permutation that is consistent across Mix, DB-SM and DB-RSM protocols (i.e., the permutation used during DB-RSM is the same as that used during Mix and the permutation used during DB-SM is the inverse of it). Such proofs can be given efficiently using the permutation-commitment based techniques of [77, 83]. Each \((M_k)_{k \in [m]}\) must verify proofs given by other mix-servers before participating in the corresponding threshold decryption protocols.

Each \((M_k)_{k \in [m]}\) must provide proofs of knowledge of the binding factors of homomorphically blinded signatures. During DB-SM, this involves giving proofs \(\rho_{\tilde{e}_i}^{(k)}\) for each \(\tilde{e}_i\) published by \(M_k\) as a blinding of \(e_i\) with blinding factor \(b_i^{(k)}\). Note that \(\rho_{\tilde{e}_i}^{(k)} := \text{NIZKP}\{(r_i, b_i^{(k)}) : \tilde{c}_i = \frac{g_f^{y_i} b_i^{(k)}}{\tilde{c}_i} \land \tilde{c}_i = \text{pk}_{\text{EG}}^{r_i} b_i^{(k)}\}\), where \(e_i\) and \(\tilde{e}_i\) are parsed as ElGamal ciphertexts \((\alpha_i, \gamma_i)\) and \((\alpha_i, \tilde{\gamma}_i)\) respectively. During DB-RSM, this involves giving NIZK proofs \(\rho_{\tilde{b}_j r_j}^{(k)}\) for each \((\tilde{b}_j, r_j, b_j^{(k)})\) which are encryptions of \(g_j^{b_j^{(k)}}\) under \(E_G\). Each \((\tilde{b}_j, b_j^{(k)}, r_j)\) are proofs of knowledge of the plaintexts encrypted by \(b_j^{(k)}\). During the \(E_G\) TDec and \(E_{Pa}\) TDec protocols, each \((M_k)_{k \in [m]}\) must provide proofs that they produced correct decryption shares. Each \((M_k)_{k \in [m]}\) should proceed with stage 2 only if these proofs pass.

During the DB-RSM protocol, each \((M_k)_{k \in [m]}\) must provide a proof of knowledge of the opening of \(g_j^{b_j^{(k)}}\). Each \((M_k)_{k \in [m]}\) should proceed only if the proof passes.

Note that the DPKs in DB-SM and DB-RSM already employ the Fiat-Shamir heuristic [42] which makes them general ZKPs in the random oracle model (see Section 3.1.5).

In Section C.5, we sketch a proof that our construction with the above steps protects secrecy (Definition 4) against general malicious adversaries.

**C PROOFS**

**C.1 Proof for Theorem 1**

It can be inspected that when all the parties are honest, inputs to protocols DB-SM and DB-RSM satisfy the preconditions mentioned in Figures 8 and 9, respectively. This is also true for the reruns of DB-SM and DB-RSM in the BTraceIn/BTraceOut calls against the complement sets. Thus, by Lemma 1, \(I^*\) and \(I^*_c\) obtained by \(Q\) in a BTraceIn call satisfy \(I^* = \{i \in I | \forall i \in \{j \in [n] | \epsilon_i \subseteq \epsilon_{j-1}\}\} \) and \(I^*_c = \{i \in I | \forall i \in \{j \in [n] | \epsilon_i \not\subset \epsilon_{j-1}\}\} \). Also, the correctness of Mix implies that for each \(i \in I \subseteq [n], \{i \in \{j \in [n] | \epsilon_i \subseteq \epsilon_{j-1}\}\}, \{i \in \{j \in [n] | \epsilon_i \not\subset \epsilon_{j-1}\}\} \) and \(J \subseteq \{n\}\), \(I^* \cup I^*_c = \{i \in I | \forall i \in \{j \in [n] | \epsilon_i \subseteq \epsilon_{j-1}\}\} \) and \(I^*_c \cup J = \{i \in I | \forall i \in \{j \in [n] | \epsilon_i \not\subset \epsilon_{j-1}\}\} \) for any \(J \subseteq \{n\}\). Thus, \(I^* \cup I^*_c = \emptyset\) and \(I^* \cup J = \emptyset\), which implies that \(Q\) does not abort and outputs a \(\epsilon_{I^*}\) that satisfies the first condition of Exp_completeness (Figure 3). By a similar argument using Lemma 2, it follows that \(Q\) does not abort in a BTraceOut call and outputs a \(\epsilon_{I^*_c}\) that satisfies the second condition of Exp_completeness.

**Lemma 1.** If inputs \((pk_{EG}, y, \epsilon_i, I, J, (M_k)_{k \in [m]}^{sk_{sk_{EG}}}, \pi(k), \nu(k), r(k))_{k \in [m]}\) of a DB-SM invocation satisfy the preconditions mentioned in Figure 8 and all the parties are honest then \(Q\) outputs \(I^* = \{i \in I | \forall i \in J : \sum_{k \in [m]} \nu(k)_{i} = \nu'(k)\}.

**Proof.** Note that for honest mix-servers, the \(i^{th}\) DPK passes iff \(M_k\) uses \(g_i^{(s_j, r_j, b_i^{(k)})}\) such that \((u_{i}, r_i, b_i) := (\sum_{k \in [m]} \nu(k), \sum_{k \in [m]} r_i^{(k)}, \sum_{k \in [m]} b_i^{(k)})\) satisfy the predicate \(PB_{\text{DB}}\). The equation \(y_i = \frac{g_i^{\nu_{i} + r_i^{(k)}}}{PB_{\text{DB}}}\) is trivially satisfied by the correctness of \((u_{i}, r_i, b_i)\). Next, note that \(\tilde{\sigma}_i = (\nu'_{i} + \pi_{i} - \nu_{i}) \sum_{k \in [m]} b_i^{(k)}\), by the homomorphism of \(E^*_G\). Let \(j \in [n]\) be the index to which index \(i\) is mapped after permutation, i.e., \(i = \pi(j)\) or equivalently \(\pi^{-1}(i) = j\).

Correctness of input conditions implies \(\nu'_{i} = \sum_{k \in [m]} \nu_{i}^{(k)} = \sum_{k \in [m]} \nu_{i}^{(k)}\) under \(E_G\). Thus, \(\tilde{\sigma}_i = (\nu'_{i} + \pi_{i} - \nu_{i}) \sum_{k \in [m]} b_i^{(k)}\), which equals \(g_i^{\nu_{i} + r_i^{(k)}}\), if \(j \in J\) and \(g_i^{\sum_{k \in [m]} b_i^{(k)}}\), if \(j \not\in J\). In the first case, the second equation of \(PB_{\text{DB}}\) passes; in the second case, it fails. Since the DPK is run only for \(i \in I, I^*\) is exactly as claimed.

**Lemma 2.** If inputs \((pk_{EG}, pk_{pa}, y, \epsilon_i, I, J, (M_k)_{k \in [m]}^{sk_{sk_{EG}}}, \nu(k), r(k))_{k \in [m]}\) of a DB-RSM invocation satisfy the preconditions mentioned in Figure 9 and all the parties are honest then \(Q\) outputs \(J^* = \{j \in [n] | \forall j \in J : \sum_{k \in [m]} \nu_{i}^{(k)} = \nu'(k)\}.

**Proof.** As in Lemma 1, let \(i\) be s.t. \(i = \pi(j)\). By correctness of inputs, \(\nu'_{i} = \sum_{k \in [m]} \nu_{i}^{(k)}\), \(\epsilon_{I^*_c} = E_{Pa} \text{Enc}(pk_{pa}, \sum_{k \in [m]} r_i^{(k)})\).

Then, it can be inspected that the following equalities hold:

- \(S'_{j} = (\sum_{k \in [m]} \nu_{i}^{(k)} r_i^{(k)} + \sum_{k \in [m]} r_i^{(k)} \frac{1}{\nu_{i}^{(k)}} r_i^{(k)} \sum_{k \in [m]} b_j^{(k)})\) if \(i \in I\) else \(f_i^{(k)}\),
- \(\nu'_{j} = \nu_{i}^{(k)} + \sum_{k \in [m]} b_j^{(k)}\),
- \(\nu'_{j} = \nu_{i}^{(k)} + \sum_{k \in [m]} b_j^{(k)}\)

Therefore, \((S'_{j} g_i^{r_i^{(k)}} b_j^{(k)} e_{I^*_c} - b_j^{(k)} e_{I^*_c})\) satisfies the BBS+ verification equation \(e(S'_{j} g_i^{r_i^{(k)}} b_j^{(k)} e_{I^*_c} - b_j^{(k)} e_{I^*_c}) = e(f_i^{(k)} g_i^{r_i^{(k)}} b_j^{(k)} e_{I^*_c})\) if \(i \in I\), where
Thus, with the mix-servers holding shares of \((b_{Sj}, b_{Cj}, b_{Rj}) = (\sum_{k \in [m]} b_{Sj}^{(k)}, \sum_{k \in [m]} b_{Cj}^{(k)}, \sum_{k \in [m]} b_{Rj}^{(k)})\). So:

\[
e(S_j, y_j^f, y_j^e) = e(f_S g_S^{r_j} h_{b_j}^{r_j}, f_j)
\]

\[
\equiv e(S_j, y_j^f, \tilde{y}_j^e) = e(f_S g_S^{r_j} h_{b_j}^{r_j}, f_j)
\]

\[
\equiv e(S_j, y_j^f, y_j^e) = e(f_S g_S^{r_j} h_{b_j}^{r_j}, f_j)
\]

\[
\equiv 32 = g_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
\]

Thus, with the mix-servers holding shares of \((b_{Sj}, b_{Cj}, b_{Rj})\) and \(\delta = b_{Sj} b_{Cj} b_{Rj}\), we have that all equations of the predicate \(\beta_{BBS+j}\) of the DPK are satisfied if \(i \in I_i\). If \(i \notin I\), the last equation of \(\beta_{BBS+j}\) is not satisfied. The claim follows. □

\section*{C.2 Proof for Theorem 2}

Suppose for contradiction that there is a PPT adversary \(\mathcal{A}\) such that \(\text{Exp}_{\text{soundness}}\) (Figure 4) outputs 1 with non-negligible probability. Note first that \(Q\) always outputs \(i^* \in I\) in DB-SM and \(j^* \in J\) in DB-RSM. Thus, \(c_{i^*} \in c_I\) and \(v_{j^*} \in v_J\). We now consider the possible cases:

Case 1: \(c_{i^*} \notin \{c_i \mid c_i \in c_I\}\). This leads to the following sub-cases:

- Case 1.1: \(\exists c_i \in c_I : v_i \notin v_J\). As per Figure 7, \(c_i\) contains \(y_i = g_1^{r_j} h_j^{r_j}\) for some \(r_j \in Z_q\). Since \(c_i \in c_I\), \(v_i \in v_J\). Thus, by Lemma 3, a PPT extractor can extract a tuple \((j^*, r^*)\) such that \(y_j = g_1^{r_j} h_j^{r_j}\) and \(v_j \in v_J\). The requirement \(v_j \notin v_J^c\) implies that \(v_i \neq v_j\). This allows producing two different openings \((v_i, r_i)\) and \((v_j, r_j)\) for Pedersen commitment \(y_i\), which is a contradiction under the discrete logarithm assumption in \(G_2\).

- Case 1.2: \(\exists c_i \in c_{I^c} : v_i \in v_J\). Note that since \(Q\) produces a \(c_I\) and does not abort, it must be that \(I^c \cap I^* = I\) during the BTraceOut call. Thus, \(I^c = I \setminus I^c\). Further, since all \(v_j\)s are distinct, \(v_i \in v_J \implies v_i \notin v_J^c\). Thus, this case can be restated as follows: \(\exists c_i \in c_{I^c} : v_i \notin v_J^c\). Thus, by applying Lemma 3 for the second DB-SM call in BTraceOut and proceeding as the previous case, we conclude that this case is not possible.

Case 2: \(v_j \neq \{v_j \in v_J \mid v_j \in v_J\}\). This leads to the following sub-cases:

- Case 2.1: \(\exists v_j \in v_J : v_j \notin v_J\). Since \(v_j \in v_J\), by Lemma 4, a PPT extractor can extract a tuple \((i^*, r^*)\) such that \(y_j = g_1^{r_j} h_j^{r_j}\) and \(i^* \in I\). Since \(i^* \in I\), the requirement \(v_j \notin v_J\) implies \(v_j \neq v_i\). As per Figure 7, \(y_i \leftarrow g_1^{r_j} h_j^{r_j}\) for some \(r_j \in Z_q\). This allows producing two different openings \((v_i, r_i)\) and \((v_j, r_j)\) for \(y_i\), which leads to a contradiction.

- Case 2.2: \(\exists v_j \in v_J \setminus v_I : v_j \notin v_I\). Note that since \(Q\) produces a \(v_j\) and does not abort, it must be that \(J^c \cap J^* = J\) during the BTraceOut call. Thus, \(J^c = J \setminus J^c\). Further, since all \(v_i\)s are distinct, \(v_j \in v_I \implies v_j \notin v_I\). Thus, this case can be restated as follows: \(\exists v_j \in v_J \setminus v_I : v_j \notin v_I\). Thus, by applying Lemma 3 for the second DB-SM call in BTraceOut and proceeding as the previous case, we conclude that this case is not possible.
\[ r_i \text{ (line 6) and forwards the quasi-signatures and their encrypted versions to } \mathcal{A}, \text{ along with invalid quasi-signatures for } i \not\in I \text{ — similar to } Q \text{ (lines 5-10). } \mathcal{A} \text{ responds with blinded permuted signatures } \hat{\sigma}' \text{ (line 11), as } (M_k)_{k \in |m|} \text{ do in the real protocol at the end of stage 1. In stage 2, for each } j \in J, \mathcal{E} \text{ extracts blinding factors for the blinded signature } \hat{\sigma}' \text{ using extractor } \mathcal{E}_2 (\text{lines 13-14}), \text{ from which it obtains an unblinded signature } \sigma' \text{ (line 15). } \mathcal{E} \text{ then attempts to find an opening } \nu_i \text{ extracted by } \mathcal{E}_1 \text{ for some commitment } y_i \in y \text{ such that } \nu_i = \sigma'_i, \text{ and returns the corresponding tuple } \langle i, r_i \rangle \text{ (lines 16-18). If no such } \nu_i \text{ exists, it outputs the message-signature tuple } \langle \sigma'_i, \sigma'_j \rangle. \]

Note that \( Q \) produced some \( j' \) and did not abort. In this case, the view produced by \( \mathcal{E} \) to \( \mathcal{A} \) is identical to that produced by \( \mathcal{Q} \) to \( (M_k)_{k \in |m|} \). Further, for some \( \sigma'_i \in \sigma'_j \), if \( \nu_i = \sigma'_i \) for some \( i \in I \), \( \langle i, r_i \rangle \) output in line 17 is a desired tuple since \( y_i = g_{\nu_1} h_{\nu_1} g_{\nu_2} h_{\nu_2} \) and \( i \in I \) (the first equality follows by the soundness of the proof of knowledge of commitment openings; the second because \( \nu_i = \sigma'_i \)). If no such \( \nu_i \) exists, we show that the tuple \( \langle \sigma'_i, \sigma'_j \rangle \) output in line 19 is a valid BBS+ signature forgery:

- For all \( i \in I \), \( \nu_i \neq \sigma'_i \), a BBS+ signature for \( \sigma'_j \) was not queried from \( C \) in line 3.
- For \( \sigma'_i \neq \sigma'_j \), the DPK for \( p_{\text{BBS}+} \) must have passed. Thus, by the soundness of DPK:
  \[ \mathcal{g}_1 = b_{\delta_1} b_{\delta_2} \quad (5) \]
  \[ \mathcal{g}_2 = b_{\delta_1} b_{\delta_2} \quad (6) \]
  \[ \mathcal{g}_3 = b_{\delta_1} b_{\delta_2} b_{\delta_1} b_{\delta_2} \quad (7) \]

From Equations 5 and 6, \( b_{\delta_1} b_{\delta_2} b_{\delta_1} b_{\delta_2} = b_{\delta_1} b_{\delta_2} \). It must be that \( \delta_1 = b_{\delta_1} b_{\delta_2} \), otherwise two different openings (\( \delta_1, \delta_2 \)) and \( (b_{\delta_1} b_{\delta_2}, \delta_1 b_{\delta_2}) \) for the Pedersen commitment \( b_{\delta_1} b_{\delta_2} \) can be produced. Equation 7 thus implies \( \mathcal{g}_3 = b_{\delta_1} b_{\delta_2} b_{\delta_1} b_{\delta_2} \), which implies that \( \sigma'_j = (S_j g_{\nu_1}^{-1}, \hat{c}_j, \nu_2 - \nu_1) \) satisfies the BBS+ signature verification equation on message \( \sigma'_j \) under public key \( y: e(S_j g_{\nu_1}^{-1}, \hat{c}_j, \nu_2 - \nu_1) = e(\nu_1, \hat{c}_j, \nu_2 - \nu_1) \) (see Equation 4; Lemma 2).

Since all \( n \) signature queries could have been made, forging a BBS+ signature is not possible under the \( n \)-Strong Diffie Hellman assumption in \( \langle G_1, G_2 \rangle \) [4]. Thus, for each \( \sigma'_i \neq \sigma'_j \), some \( i \in I \) such that \( \nu_i = \sigma'_i \) must exist and a desired tuple \( \langle i, r_i \rangle \) must have been produced.

\[ E_3 \text{ (Figure 15): In } E_3, \text{ shares of } \nu_i, r_i \text{ for } i \in \{i_0, i_1\} \text{ are drawn as } (g_{\nu_i}^{-1}, r_i)_{k \in \mathbb{Z}_q} \text{ and } \sigma'_i \text{ is obtained from } E_2 \text{ because the additive secret sharing is information-theoretically secure.} \]

\[ E_4 \text{ (Figure 16): In } E_4, \text{ instead of decrypting } \mathcal{E}(\nu_i, \sigma'_i) \text{ for } i \in \{i_0, i_1\} \text{ during Mix, } \sigma'_i (r_i) \text{ are obtained by directly using their corresponding values in the Enc call for } i_0, i_1. \]

\[ E_5 \text{ (Figure 17): In } E_5, \text{ instead of decrypting } \sigma'_i, \text{ the } \mathcal{E}(\nu_i, \sigma'_i) \text{ for } i \in \{i_0, i_1\} \text{ are decrypted by both encoders to extract blinding factors for the } \nu_i \text{ in the Enc calls for } i_0, i_1 \text{ by replacing by encryptions of 0. } E_6 \text{ is distinguishable from } E_5 \text{ by the IND-CPA security of } \mathcal{E}. \]

\[ E_7 \text{ (Figure 19): In } E_7, \text{ responses of } M_k \text{ in } E_{\mathcal{E}_C}^T \text{, } E_{\mathcal{E}_C}^T \text{ and } E_{\mathcal{E}}^T \text{ protocols are simulated by first obtaining the correct decryption } m \text{ of the given ciphertext } c \text{ using ideal decryption oracles } \mathcal{E}_{\mathcal{E}_C}^T \text{ and } \mathcal{E}_{\mathcal{E}}^T \text{ and then simulating } M_k \text{’s responses using } c \text{, } m \text{ and secret keys for } (M_k). \]

\[ E_8 \text{ (Figure 20): In } E_8, \text{ instead of using } \sigma' \text{ given by } \mathcal{E}_{\mathcal{E}}^T \text{ during Mix, it is set as } \sigma' \leftarrow (\sigma'_{\nu(i)} e_n). \text{ Here, } \pi \leftarrow \pi^{(m)} \cdot \cdots \cdot \pi^{(1)} \text{ is obtained using the experimenter-selected } \pi_k \text{ (and } \hat{\pi}_k \text{) obtained from the random tape issued to } \mathcal{A}, \text{ and } (\nu_i)_{i \in \{i_0, i_1\}} \text{ are obtained from their values in Enc calls for } i_0, i_1 \text{ and } (\sigma'_i)_{i \in \{n\}} \text{ are obtained from } \mathcal{A} \text{’s input for senders } (S_i). \text{ In the correctness of Shuffles } (E_9 \text{ by the correctness of } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{ protocols.} \]

\[ E_9 \text{ (Figure 21): In } E_9, \text{ instead of using } \sigma' \text{ given by } \mathcal{E}_{\mathcal{E}}^T \text{ during DB-SM, it is set as } \sigma' \leftarrow (\sigma'_{\nu(i)} b_{\mathcal{E}}) e_n. \text{ Here } \sigma' \text{ denotes BB signatures sent by } \mathcal{A} \text{ at the beginning of DB-SM, } \pi \text{ is as obtained in } E_8 \text{ and } b_{\mathcal{E}} := \sum_{k \in |m|} b_{\mathcal{E}} k \text{ is obtained using the experimenter-selected } \hat{\pi}_k \text{ and } (\hat{\pi}_k, b_{\mathcal{E}} k) \text{ obtained from } \mathcal{A} \text{’s random tape. } \mathcal{E}_9 \text{ is indistinguishable from } E_9 \text{ by the correctness of Shuffles and } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{ protocols.} \]

\[ E_{10} \text{ (Figure 22): In } E_{10}, \text{ instead of using } \hat{\sigma}' \text{, } \hat{\sigma}' \text{, } \hat{\sigma}' \text{ given by } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{ during } \text{DB-RSM}, \text{ they are set as } \hat{\sigma}' \leftarrow (S_{\nu(i)} b_{\mathcal{E}}) e_n. \text{ Here, } \hat{\sigma}' \text{ is identical to that produced by } \mathcal{A} \text{ at the beginning of DB-RSM. } \mathcal{E}_{10} \text{ is indistinguishable from } E_{10} \text{ in the correctness of Shuffles and } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{, } \mathcal{E}_{\mathcal{E}}^T \text{ protocols.} \]
(S, c, r) denote BBS+ quasi-signatures sent by A at the beginning of DB-RSM, π is as obtained in Eq. bS_j ← \sum_{k \in \{m\}} bS_j^k, b_{c_j} ← \sum_{k \in \{m\}} b_{c_j}^k \mod N, b_{r_j} ← \sum_{k \in \{m\}} b_{r_j}^k \mod N are obtained using experimenter-selected bS_j^k, b_{c_j}^k, b_{r_j}^k := b_{c_j}^k + qX_j + b_{r_j}^k := b_{r_j}^k + qX_j. Components \(b_j\) are encrypted using a random tape. (r)_{\ell}(\pi_j) \in \{a, b\} are obtained from the random tape given to A. \(E_\ell\) is indistinguishable from \(E_0\) by the correctness of Shuffle, \(E_{\mathcal{F}}\), TDec and \(E_{\mathcal{P}}\) TDec protocols.

- For \(j \in \{\ell_0, \ell_1\}\), the DPK is simulated completely if \(\ell_0 = \pi_j \in \ell\) otherwise the first two equations do not pass, their first two equations do pass.

- For \(j \in \{\ell_0, \ell_1\}\), the DPK is not simulated, i.e., values \((b_{s_j}^k, b_{c_j}^k, b_{r_j}^k)\) used exactly as \(\mathcal{M}_\ell\) does.

- For \(j \in \{\ell_0, \ell_1\}\), the DPK is simulated completely if \(\ell_0 = \pi_j \in \ell\) otherwise the blinded signature is invalid, \(b\) by the assent condition in the OTraceOut call, the DPK for \(j_0\) passes iff the DPK for \(j_0\) passes, and \(c\) even when these DPKs do not pass, their first two equations do pass.

- In E15 (Figure 28): In \(E_{15}\), \(S_{\mathcal{F}}\) is chosen uniformly at random from \(\mathcal{Z}_{\mathcal{F}}\). \(E_{15}\) is indistinguishable from \(E_{14}\) because of perfect blindings using \(\delta(z)\).

- In E16 (Figure 29): In \(E_{16}\), shares \((\ell_0, \ell_1, v_0, v_1, r_0, r_1)\) during the Enc calls for \(i_0, i_1\) are set to \(0, 0\). \(E_{16}\) is indistinguishable from \(E_{15}\) because these shares are not used anymore.

- In E17 (Figure 30): In \(E_{17}\), commitments \(y_{i_0}, y_{i_1}\) in the Enc calls commit to \(\ell_0, \ell_1\) is indistinguishable from \(E_{16}\) because Pedersen commitments are perfectly hiding and the committed values are not used anywhere.

In E18 (Figure 31): In \(E_{18}\), \(\bar{\sigma}_{\ell_0}, \bar{\sigma}_{\ell_1}\) during DB-SM are replaced by randomly drawn elements from \(\mathcal{G}_i\). \(E_{18}\) is indistinguishable from \(E_{17}\) because \((\ell_0, \ell_1, v_0, v_1, r_0, r_1)\) used in computing \(\bar{\sigma}_{\ell_0}, \bar{\sigma}_{\ell_1}\) are chosen uniformly at random from \(\mathcal{Z}_{\mathcal{F}}\).

- In E19 (Figure 32): In \(E_{19}\), \(\check{S}_{\ell_0}, \check{S}_{\ell_1}\) during DB-SM are replaced by randomly drawn elements from \(\mathcal{G}_i\). \(E_{19}\) is indistinguishable from \(E_{18}\) because \((\ell_0, \ell_1, v_0, v_1, r_0, r_1)\) used in computing \(\check{S}_{\ell_0}, \check{S}_{\ell_1}\) are chosen uniformly at random from \(\mathcal{Z}_{\mathcal{F}}\) and \((\ell_0, \ell_1, v_0, v_1, r_0, r_1)\) used in computing \(\check{S}_{\ell_0}, \check{S}_{\ell_1}\) are chosen uniformly at random from an exponentially larger space than the corresponding messages \(c_{\ell_0}(\pi), c_{\ell_1}(\pi)\).
C.5 Secrecy in the malicious model

In this section, we sketch a proof that when the additional steps of Appendix B are applied then our construction protects secrecy (Definition 4) even against general malicious adversaries. We assume that secure MPC protocols are used for Beaver triples generation and for \( E_{th}^{EG} \) and \( E_{th}^{pa} \) distributed key generation.

The proof essentially follows the structure of the proof in Appendix C.3 with the following differences:

- In \( E_5, \rho_i^{(k)} \) and \( t_i^{(k)} \) for \( i \in [n] \setminus \{i_0, i_1\} \) are obtained by extracting them from proofs \((\rho_{c_1}, \rho_{c_2}, k_{\in [m]} \): the corresponding proofs for \( i \in \{i_0, i_1\} \) are simulated. \( E_5 \) is indistinguishable from \( E_4 \) by knowledge soundness of \( P_{c_1}^{(k)}, P_{c_2}^{(k)} \) for \( i \in [n] \setminus \{i_0, i_1\} \) and their zero-knowledge for \( i \in \{i_0, i_1\} \).

- In \( E_6, (\pi(k))_{k \in k^*} \) are obtained by extracting them from proofs of shuffle produced by \( (M_k)_{k \in k^*} \) and \( (m_i)_{i \in [n]} \) are obtained by extracting them from proofs \( \rho_i \) for \( i \in [n] \setminus \{i_0, i_1\} \). Proofs of shuffle for \( M_k \) and proofs \( \rho_i \) for \( i \in \{i_0, i_1\} \) are simulated. \( E_6 \) is indistinguishable from \( E_7 \) by a) knowledge soundness of proofs of shuffles for \( (M_k)_{k \in k^*} \), b) knowledge soundness of \( \rho_i \) for \( i \in [n] \setminus \{i_0, i_1\} \), c) zero-knowledge of proofs of shuffle for \( M_k \), d) zero-knowledge of \( \rho_i \) for \( i \in \{i_0, i_1\} \), and e) correctness of threshold decryption protocol \( E_{th}^{pa, TDec} \).

- In \( E_{10}, (b_j(k))_{k \in k^*} \) are obtained by extracting them from \( (\rho_{\sigma_j, i}^{(k)})_{k \in k^*} \). The corresponding proofs for \( k = k' \) are simulated. \( E_9 \) is indistinguishable from \( E_8 \) by a) verification of correct construction of \( \alpha'_e \) from \( \sigma' \), b) knowledge soundness of proofs of shuffle that prove that each \( (M_k)_{k \in k^*} \) shuffled encrypted signatures in this phase correctly using the inverse of the permutation \( \pi(k) \) it applied during mixing, c) knowledge soundness of \( (\rho_{\sigma_j, i}^{(k)})_{k \in k^*} \), d) zero-knowledge of proofs of shuffle for \( k = k' \), e) zero-knowledge of \( \rho_{\sigma_j, i}^{(k)} \), and f) correctness of threshold decryption protocol \( E_{th}^{EG, TDec} \).

- In \( E_{13}, \) the correctness of blinded signatures is ensured through the additional verification that signatures \( \sigma_j^* \) for each \( j \in J \) are valid BB signatures on \( \sigma_j^* \) and are invalid signatures for \( j \notin J \).

- In \( E_{14}, \) the correctness of blinded signatures is ensured through the additional verification that quasi-signatures \( (S_i, e_i, f_i) \) for each \( i \in I \) are valid BB+ quasi signatures using commitment \( y_i \) and invalid for \( i \notin I \). Note that correctness of quasi-signatures implies correctness of full BB+ signatures for \( i \in \{i_0, i_1\} \) because encryptions \( e_i \) are generated by honest senders. Further, in this case, \( (\delta_0^{(k)}, \delta_1^{(k)}, \delta_2^{(k)})_{k \in k^*} \) are obtained by extracting \( \delta_0^{(k)} \) from proofs of knowledge of the opening of \( (\delta_0^{(k)}, \delta_1^{(k)})_{k \in k^*} \) and computing \( (\delta_0^{(k)}, \delta_1^{(k)})_{k \in k^*} \) using the Mult algorithm on inputs \( (\delta_0^{(k)}, b_s^{(k)}) \) and \( (\delta_0^{(k)}, b_c^{(k)}) \) respectively, where \( b_s^{(k)}(k), b_c^{(k)}(k) \equiv (b_{c_i}(k) mod q) \) are the ones obtained in \( E_{10} \) (note that this requires Beaver triples held by \( (M_k)_{k \in k^*} \) which are provided by the ideal functionality for the MPC protocol for Beaver triple generation).

D PRIVACY RISK ANALYSIS OF TRACEIN/TRACEOUT QUERIES

In this section, we provide formal rules to analyse the privacy risk impact of allowing a given set of Tracen/TraceOut queries to a querier in an application. Since the goal of the analysis is to apriori decide whether to allow the queries or not, we assume that we only have query inputs and not their outputs. Further, we assume that no query’s input depends on other queries’ outputs. Given this, we adopt a static analysis approach where we conservatively estimate the information potentially leaked by the queries. If the actual information leaked by a query depends on its output, we conservatively assume that the query potentially leaks the information corresponding to both the outputs.

Consider a mixnet with input ciphertexts \( (c_i)_{i \in [n]} \) and output plaintexts \( (\nu_j')_{j \in [n]} \). Let \( Q \) denote the set of proposed allowed queries containing elements of the form (Tracein, \( i, j \)) and (TraceOut, \( i, j \)) for \( i, j \in [n] \) and \( I, J \subseteq [n] \) representing the Tracein(i, j) and TraceOut(i, j) queries respectively (BTracein/BTraceOut queries can be compiled to this format). Let \( K(i, j) \) denote the fact that the querier potentially knows that ciphertext \( c_j \) encrypts a plaintext in set \( \nu_j' \). Let \( K(I, j) \) (with an index set as the first argument and an index as the second argument) denote the fact that the querier potentially knows that plaintext \( \nu_j' \) is encrypted in some ciphertext in set \( c_j \). We therefore have the following rules:

- R0.1: \( \forall i \in [n] : K(i, [n]). \)
- R0.2: \( \forall j \in [n] : K([n], j). \)
- R1.1: \( \forall i \in [n], J \subseteq [n] \setminus \{i\} : (\text{Tracein}, i, j) \in Q \implies K(i, j) \land K(i, [n] \setminus J). \)
- R1.2: \( \forall j \in [n], I \subseteq [n] \setminus \{j\} : (\text{TraceOut}, i, j) \in Q \implies K(i, j) \land K([n] \setminus I, j). \)
- R2.1: \( \forall i \in [n], J, J' \subseteq [n] : K(i, J) \land K(i, J') \implies K(i, J \cap J'). \)
- R2.2: \( \forall j \in [n], I, I' \subseteq [n] : K(i, J) \land K(I', j) \implies K(I \cap I', j). \)
- R3.1: \( \forall i, j \in [n], J, I \subseteq [n] : K(i, J) \land K(I, j) \land i \notin I \land j \notin J \implies K(I \setminus \{i\}, j). \)
- R3.2: \( \forall j \in [n], I, J \subseteq [n] : K(i, J) \land K(I, j) \land i \in I \land j \notin J \implies K(I \setminus \{i\}, j). \)

Rules R0.1 and R0.2 encode the fact that the querier initially knows that each input ciphertext encrypts a plaintext in the output list and each output plaintext is encrypted in a ciphertext in the input list. R1.1 (symmetrically R1.2) encodes that a Tracein(i, j) query leads to either the querier learning that \( c_j \) encrypts a plaintext in \( \nu_j' \) (if the query passed) or that \( c_j \) encrypts a plaintext in \( \nu_j' \) (if the query failed). Thus, potentially, the querier may learn both.
R2.1 (symmetrically R2.2) encodes that if the querier knows that \( c_i \) encrypts a value in \( v_j' \) and in \( v_j'' \), then it knows that \( c_i \) encrypts a value in \( v_{j/j''} \). R3.1 (symmetrically R3.2) encodes that if the querier knows that \( c_i \) encrypts a value in \( v_j' \) but there exists a \( j \in J \) that is known to be encrypted in a ciphertext in a set \( I \) that does not include \( i \), then it knows that \( c_i \) cannot encrypt \( v_j' \) because of distinctness of encrypted values.

Given these rules, the goal of our analysis is to output a) for each \( i \in [n] \), the smallest set \( J_{min} \) such that \( K(i, J_{min}) \) holds; and b) for each \( j \in [n] \), the smallest set \( I_{min} \) such that \( K(I_{min}, i) \) holds. Note that if for some \( i \), no TraceIn(i, J) query exists in \( Q \) then \( J_{min} = [n] \) (similarly for \( I_{min} \)). The \( J_{min} \), \( I_{min} \) outputs tell exactly what information is potentially leaked by the queries in \( Q \). This information can directly feed to the application-level security analysis. Since each implication rule monotonically decreases the size of either the \( J \) set for \( K(i, J) \) or the \( I \) set for \( K(I, j) \), \( J_{min} \) and \( I_{min} \) can both be obtained in finite time for a finite set \( Q \) using a fixed-point algorithm.
\( E_i(1^k, k', i_0, i_1, b) : \)

// Keygen:
E → \( \mathbf{pk}_\mathbf{E}(k') \leftarrow \text{KGen}(2^k) \)
E → \( \mathbf{sk}_\mathbf{E}(k') \leftarrow \text{EKeyGen}(1^t, \langle \mathbf{M}_{\mathbf{E}} \rangle_{k \leftarrow k'} \rangle) \)
E → \( \mathbf{sk}_{\mathbf{P}a}(k') \leftarrow \text{P}a_{\mathbf{E}} \cdot \text{KeyGen}(1^t, \langle \mathbf{M}_{\mathbf{E}} \rangle_{k \leftarrow k'} \rangle) \)
E → \( \mathbf{pk}_{\mathbf{P}a}(k') \leftarrow \text{EKeyGen}(1^t, \langle \mathbf{M}_{\mathbf{E}} \rangle_{k \leftarrow k'} \rangle) \)
E → \( (\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{pk}_{\mathbf{P}a}) \)
E → \( \mathbf{u}_0, \mathbf{v}_0, (\mathbf{e}_i)_i \in [1, t] \)
E: if \( b = 0 \) then \((\mathbf{e}_i, \mathbf{v}_i) := (\mathbf{u}_0, \mathbf{v}_0)\) else \((\mathbf{e}_i, \mathbf{v}_i) := (\mathbf{v}_i, \mathbf{u}_0)\)

// Enc calls for \( i_0, i_1 \):
E: for \( i \in [1, t] \):
\( e_i \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
\( \mathbf{y}_i \leftarrow \mathbf{y}_i^t \)
\( \mathbf{z}_i \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
\( \mathbf{c}_i \leftarrow (\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i) \in [1, t] \)
E → \( \mathbf{c}_i \)

// Mix:
E: π(dx) \leftarrow \text{Perm}(n)
E → \( \mathbf{c}_e \leftarrow \text{Perm}(\mathbf{c}) \)
E: \( \mathbf{c}_e \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
E → \( \mathbf{c}_e \)
E: \( \mathbf{c}_f \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
E → \( \mathbf{c}_f \)
E: endfor

// OTraceln call:
E: \( \mathbf{c}_o \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
E → \( \mathbf{c}_o \)
E: \( \mathbf{b}_e \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
E: \( \mathbf{b}_e \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
E → \( \mathbf{b}_e \)
E: \( \mathbf{b}_f \leftarrow \text{EEnt}(\mathbf{pk}_{\mathbf{E}}, \mathbf{sk}_{\mathbf{E}}, \mathbf{r}_i, \mathbf{y}_i) \)
E → \( \mathbf{b}_f \)
E: endfor

Figure 13: \( E_i \): The original \( \text{Exp}_{
\text{secret}} \) experiment instantiated for our construction.
$E_2(1^t, k^*, b_0, b_1, b_2) :$

// Keygen:
E: $\mathbf{pk}_{\mathbf{E}_2}, sk_{\mathbf{E}_2} \leftarrow \mathbf{E}_{\text{Keygen}}(1^t)$
E $\Rightarrow$ A $\mathbf{pk}_{\mathbf{E}_2}, sk_{\mathbf{E}_2} \leftarrow \mathbf{E}_{\mathbf{E}_2, \mathbf{E}}_{\text{Keygen}}(1^t, (M_{\mathbf{E}_2} \mathbf{E}_2))$
E $\Rightarrow$ A $\mathbf{pk}_{\mathbf{E}_2}, sk_{\mathbf{E}_2} \leftarrow \mathbf{E}_{\mathbf{E}_2, \mathbf{E}}_{\text{Keygen}}(1^t, (M_{\mathbf{E}_2} \mathbf{E}_2))$
E $\Rightarrow$ A $\mathbf{pk}_{\mathbf{E}_2}, sk_{\mathbf{E}_2} \leftarrow \mathbf{E}_{\mathbf{E}_2, \mathbf{E}}_{\text{Keygen}}(1^t, (M_{\mathbf{E}_2} \mathbf{E}_2))$

// for $i \in [t]_0$:
E: $\mathbf{c}_i \leftarrow \mathbf{E}_{\mathbf{E}_2, \mathbf{E}}_{\text{Dec}}(\mathbf{sk}_{\mathbf{E}_2}, (T_{\mathbf{E}_2} \mathbf{E}_2))$
E $\Rightarrow$ A $\mathbf{c}_i \leftarrow \mathbf{E}_{\mathbf{E}_2, \mathbf{E}}_{\text{Dec}}(\mathbf{sk}_{\mathbf{E}_2}, (T_{\mathbf{E}_2} \mathbf{E}_2))$

// OTraceOut call:
A $\Rightarrow$ E $\mathbf{OTraceOut}(1, i)$
E: assert $c_i \in c_t \iff c_i \in c_t$
A $\Rightarrow$ E $y_i (S, E \cdot \mathbf{c}_i \cdot e_j) / \mathbf{DB}-\mathbf{SM}$ call (repeat for $[n] \setminus i$)
E: $e_i \leftarrow \mathbf{e}_i$
A $\Rightarrow$ E $\mathbf{e}_i^{(k_i)} : \mathbf{e}_i^{(k_i)} : \mathbf{e}_i^{(k_i)} : \mathbf{e}_i^{(k_i)}$ / shuffling
E $\Rightarrow$ A $\mathbf{e}_i^{(k_i)} : (\mathbf{E}_{\mathbf{E}_2, \mathbf{E}}_{\text{Dec}}(\mathbf{sk}_{\mathbf{E}_2}, (\mathbf{M}_{\mathbf{E}_2} \mathbf{E}_2)))$ / threshold decryption
E: $\mathbf{e}_i^{(k_i)} : \mathbf{e}_i^{(k_i)} : \mathbf{e}_i^{(k_i)} : \mathbf{e}_i^{(k_i)}$
A $\Rightarrow$ E $\mathbf{b}_i^{(k_i)} : \mathbf{b}_i^{(k_i)} : \mathbf{b}_i^{(k_i)} : \mathbf{b}_i^{(k_i)}$

\[ \text{Figure 14: } E_2: \text{Simulating } \rho_\mathbf{y}_{\mathbf{b}_0} \text{ and } \rho_\mathbf{y}_{\mathbf{b}_1}. \]
$E(t, k', z, y, b, h)$:

// Keygen:
E : $pk(k'), sk(k') \rightarrow E$ Keygen($t$)
E $\rightarrow A$ : $pk_{EG}, sk_{EG} \leftarrow E_{EG}$ Keygen($t^3, (M_{k})_{k\neq k'}$)
E $\rightarrow A$ : $pk_{EG}, sk_{EG} \leftarrow E_{EG}$ Keygen($t^1, (M_{k})_{k\neq k'}$)
E $\rightarrow A$ : $pk(k'), sk(k')$
A $\rightarrow E$ : $(pk(k'))_{k\neq k'}$
A $\rightarrow E$ : $v_0, v_1, (c_1)_{i\in[n]}, \{i\}_{i=0}^n$
E : if $(b=0)$ then $(a_0, a_1) := (v_0, v_1)$ else $(a_0, a_1) := (v_1, v_0)$
// Enc calls for $b_0, b_1$:
E : for $i \in \{b_0, b_1\}$:
$e_i \leftarrow E_{EG}(pk_{EG}, c_i)$
$r_i \leftarrow z_0, y_i \leftarrow g_{i1}^r_i, \{p_j, q_j\} \leftarrow SNIZPK(y_i)$
$c_i \leftarrow E_{EG}(pk_{EG}, r_i)$
$(g^{(k')}_{i})_{k\neq k'}, r_i^{(k')}_{i \neq k'} \leftarrow Z^k_q$
$s^r_i \leftarrow \sum_{i=0}^{k+1} r_i^{(k')} - \sum_{i=0}^{k+1} s_i^{(k)}$
$(ev^r_i) \leftarrow c_i \leftarrow \sum_{i=0}^{k+1} r_i^{(k')} - \sum_{i=0}^{k+1} s^{(k)}$
$\{ev^r_i\}_{i\in[m]} \leftarrow (E_{EG}(pk_{EG}, s^r_i))_{i\in[m]}$
\endfor

// Mix:
E : $\pi_{(k')} \leftarrow \text{Perm}(n)$
A $\rightarrow E$ : $e_0^{\pi_{(k')}}, c_0^{\pi_{(k')}} \leftarrow E_{EG}(pk_{EG}, e_0^{(k')}), (M_{k})_{k\neq k'}$
E $\rightarrow A$ : $e^{(k')} \leftarrow (E_{EG}(pk_{EG}, e^{(k')})))_{i\in[n]}$
E $\rightarrow A$ : $e'$
A $\rightarrow E$ : for $j \in \{m\}$:
$p_j^{(k')} \leftarrow E_{EG}(pk_{EG}, c_j)$
$(M_{k})_{k\neq k'}$
\endfor

// OTtraceOut call:
A $\rightarrow E$ : OTTraceOut($l, f$)
E : assert $c_i \in e_i \iff e_i \in c_i$
A $\rightarrow E$ : $y, \sigma, e_i, \ell_1, \ell_2 / DB$-SM call (repeat for $[n] \setminus f$)
E $\rightarrow A$ : $e_0^{\pi_{(k')}}, c_0^{\pi_{(k')}} / $shuffling
A $\rightarrow E$ : $e_0^{(k')} \leftarrow E_{EG}(pk_{EG}, e_0^{(k')})$
\endfor
E : $b^{(k')} \leftarrow Z^k_q$ // homomorphic binding
E $\rightarrow A$ : $b^{(k')} \leftarrow E_{EG}(pk_{EG}, b^{(k')})$
\endfor
E $\rightarrow A$ : $\sigma \leftarrow E_{EG}(pk_{EG}, (M_{k})_{k\neq k'}) / $threshold decryption
E : for $i \in [k]$:
\endfor
E $\rightarrow A$ : DPK$(y, \sigma, \gamma, \bar{b}_{\gamma})$ with $\bar{b}_{\gamma} = (y_f = g_{y}^{i_1 \gamma_i \gamma}, \sigma_i y = (g_1, g_2) \in c_0)$
E : $\text{endfor}$

Figure 15: $E_5$: Send random values as shares to $(M_{k})_{k\neq k'}$. 

Agrawal et al.
Traceable mixnets

\[ E_k(v, k', t_0, v_0, t_1, b) : \]

// Keygen:

\[ E \rightarrow A: \quad \text{pk}_{EG}, \text{sk}_{EG} \leftarrow \text{Keygen}^1() \]

\[ E \rightarrow A: \quad \text{pk}_{PA}, \text{sk}_{PA} \leftarrow \text{Keygen}^2() \]

\[ E \rightarrow A: \quad \text{pk}_{PA}, \text{sk}_{PA} \leftarrow \text{Keygen}^3() \]

// Encalls for \( v_0, t_0 \):

\[ E : \quad \text{for} \ e \in \{ 0, 1 \} : \]

\[ c_e \leftarrow \text{Enc}(\text{pk}_{PA}, v_0, r_i) \]

\[ \rho \leftarrow \text{r} \leftarrow E_{\text{NIZKP}}(Y) \]

\[ c_i \leftarrow \text{Dec}(\text{pk}_{EG}, c_i) \]

// Mix:

\[ E : \quad \pi^{\langle k \rangle} \leftarrow \text{Perm}(n) \]

\[ E \rightarrow A: \quad e^{\langle k \rangle} \leftarrow \pi^{\langle k \rangle} \cdot c_i \]

\[ E \rightarrow A: \quad e^{\langle k \rangle} \leftarrow \text{Enc}(\text{pk}_{PA}, e^{\langle k \rangle}) \]

\[ E \rightarrow A: \quad e^{\langle k \rangle} \leftarrow \text{Dec}(\text{sk}_{PA}, e^{\langle k \rangle}) \]

// OTTraceout call:

\[ A \rightarrow E: \quad \text{OTTraceout}(I, J) \]

\[ E : \quad \text{assert} \ c_0 \in I \iff c_1 \in J \]

\[ A \rightarrow E: \quad y, (s, e, f, (e^2, e_f)) // \text{DB-SM call (repeat for } n \text{ \& } J) \]

\[ E : \quad e_f \leftarrow e f \]

\[ A \rightarrow E: \quad e^{\langle k \rangle}_1, e^{\langle k \rangle}_2, e^{\langle k \rangle}_3 // \text{shuffling} \]

\[ A \rightarrow E: \quad e^{\langle k \rangle} \leftarrow \text{Dec}(\text{sk}_{PA}, e^{\langle k \rangle}) \]

\[ E \rightarrow A: \quad e^i_e, e^j_e, e^r_e \]

\[ E \leftarrow \text{DB-SM call (repeat for } n \text{ \& } J) \]

\[ E \rightarrow A: \quad \text{Enc}(\text{pk}_{EG}, e^i_e, e^j_e, e^r_e) \]

\[ E \rightarrow A: \quad \text{Dec}(\text{sk}_{PA}, e^{\langle k \rangle}_1) \]

\[ E \rightarrow A: \quad \text{Dec}(\text{sk}_{PA}, e^{\langle k \rangle}_2) \]

\[ E \rightarrow A: \quad \text{Dec}(\text{sk}_{PA}, e^{\langle k \rangle}_3) \]

Figure 16: Do not use \text{sk}_{EG}^{\langle k \rangle} for decrypting \text{ev}_i^{\langle k \rangle}, \text{er}_i^{\langle k \rangle} for \text{for } i \in \{ 0, 1 \}.

25
Figure 17: *E5: Do not use sk((k)) for decrypting ev^((k),i), er^((k),i) for i ∈ [n] \ {i_0, i_1}.
Traceable mixnets

$E_{\nu}(1^{k}, k, i_{0}, i_{1}, h):$

// Keygen:
\[ E \rightarrow \mathcal{A}: pk_{\nu}, sk^{(k)}_{\nu} \leftarrow \text{E.Keygen}(1^{k}) \]
\[ E \rightarrow \mathcal{A}: pk_{\nu}, sk^{(k)}_{\nu} \leftarrow \text{E.Keygen}(1^{k}, (M_{k}^{\nu})_{h_{sk}}) \]
\[ E \rightarrow \mathcal{A}: pk_{\nu}, sk^{(k)}_{\nu}, sk_{\nu} \leftarrow \text{E.Keygen}(1^{k}, (M_{k}^{\nu})_{h_{sk}}) \]
\[ \mathcal{A} \rightarrow E: (pk_{\nu}, sk^{(k)}_{\nu}) \]
\[ \mathcal{A} \rightarrow E: (pk_{\nu}, sk^{(k)}_{\nu}) \]
\[ \mathcal{A} \rightarrow E: (pk_{\nu}, sk^{(k)}_{\nu}) \]

// Enc calls for $i_{0}, i_{1}$:
\[ \text{for } i \in \{i_{0}, i_{1}\}: \]
\[ \text{c}_{i} \leftarrow \text{Enc}(pk_{\nu}, r_{i}) \]
\[ \text{c}_{i} \leftarrow \text{Enc}(pk_{\nu}, r_{i}) \]
\[ \text{c}_{i} \leftarrow \text{Enc}(pk_{\nu}, r_{i}) \]
\[ \text{c}_{i} \leftarrow \text{Enc}(pk_{\nu}, r_{i}) \]
\[ \text{c}_{i} \leftarrow \text{Enc}(pk_{\nu}, r_{i}) \]

// Mix:
\[ \mathcal{A} \rightarrow E: e_{i}^{(k-1)} \leftarrow \text{Enc}(pk_{\nu}, c_{i}^{(k-1)}) \]
\[ \mathcal{A} \rightarrow E: e_{i}^{(k-1)} \leftarrow \text{Enc}(pk_{\nu}, c_{i}^{(k-1)}) \]
\[ \mathcal{A} \rightarrow E: e_{i}^{(k-1)} \leftarrow \text{Enc}(pk_{\nu}, c_{i}^{(k-1)}) \]
\[ \mathcal{A} \rightarrow E: e_{i}^{(k-1)} \leftarrow \text{Enc}(pk_{\nu}, c_{i}^{(k-1)}) \]
\[ \mathcal{A} \rightarrow E: e_{i}^{(k-1)} \leftarrow \text{Enc}(pk_{\nu}, c_{i}^{(k-1)}) \]

// OTraceIn call:
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]

// OTraceOut call:
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]

OTraceIn call:
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceIn}(l, i_{j}) \]

OTraceOut call:
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]
\[ \mathcal{A} \rightarrow E: \text{OTraceOut}(l, i_{j}) \]

Figure 18: $E_{\nu}$: Replace $ev_{i}^{(k)}$, $er_{i}^{(k)}$ for $i \in \{i_{0}, i_{1}\}$ by encryptions of zeros.
$E_7(1^k, k^j, i_0, i_1, k)$:

// Keygen:
$E \rightarrow$ pk$_{EC}, sk_k \leftarrow E$ Keygen($1^k$)
$E \rightarrow \mathcal{A} \leftarrow$ pk$_{EC}, sk_k \leftarrow E$ Keygen($1^k$, $(M_{sk_{k_{LHK}}})$)
$E \rightarrow \mathcal{A} \leftarrow$ pk$_{EC}, sk_k \leftarrow E$ Keygen($1^k$, $(M_{sk_{k_{LHK}}})$)
$E \rightarrow \mathcal{A} \leftarrow (pk_k, pk_{EC})$
$\mathcal{A} \rightarrow E$ (pk$_k$) k LHK
$\mathcal{A} \rightarrow (\sigma_0, \sigma_1, \beta) \in (1^k, 1^k)$
$\mathcal{A} \rightarrow \gamma \in \{0, 1\}$ if ($\gamma = 0$) then $(\sigma_0, \sigma_1) \leftarrow (\sigma_0, \sigma_1)$ else $(\sigma_0, \sigma_1) \leftarrow (\sigma_0, \sigma_1)$

// Enc calls for $i_0, i_1$:
$E \rightarrow \mathcal{A} \leftarrow$ (pk$_{EC}$) $\leftarrow$ E Enc(pk$_{EC}$, $c$) for $i_0, i_1$

// Mix:
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)
$E \rightarrow \mathcal{A} \leftarrow (e^{(i_0)}, e^{(i_1)}) \leftarrow E$ Enc(pk$_{EC}$, $c$)

$\mathcal{A} \rightarrow \mathcal{A}_r \leftarrow (\sigma_0^0, \sigma_0^1, \sigma_0^2, \sigma_0^3, \sigma_0^4)$ from the Enc calls for $i_0, i_1$

// OTraceOut call:
$\mathcal{A} \rightarrow \mathcal{A}_r \leftarrow$ OTraceOut($i, f$)
$\mathcal{A} \rightarrow \gamma \in \{0, 1\}$
$\mathcal{A} \rightarrow \mathcal{A}_r \leftarrow$ DB-RSM call (repeat for $\lceil n \rceil \backslash \{f \}$)
$\mathcal{A} \rightarrow \mathcal{A}_r \leftarrow$ DB-SM call (repeat for $\lceil n \rceil \backslash \{f \}$)

// OTraceIn call:
$\mathcal{A} \rightarrow \gamma \in \{0, 1\}$
$\mathcal{A} \rightarrow \mathcal{A}_r \leftarrow$ DB-SM call (repeat for $\lceil n \rceil \backslash \{f \}$)

// Agrawal et al.

Figure 19: $E_7$: Simulate threshold decryption protocols using ideal decryption oracles.
The page contains a detailed explanation of a cryptographic protocol, specifically focusing on the traceable mixnets algorithm. The text is dense and technical, involving equations, algorithms, and cryptographic notation. The page outlines the keygen process, encryption, and decryption steps, along with the shuffling and verification mechanisms. The text is formatted in a mathematical and programming-like style, typical of cryptographic literature.

---

**Figure 20: Eₘ:** Obtain $\sigma'$ without decryption.
\[ E_\theta(1, k, \theta_0, i_0, k) \]
Traceable mixnets

\[ E_{\text{mix}}(\{1, k^i, t_0, t_1, b\}) : \]

// Keygen:
\[ E \Rightarrow \mathcal{A} : \text{PK}_{\text{mix}}(k^i) \rightarrow E_{\text{keygen}}(t_0, \{\text{Mix}, \mathcal{A}_k, k^i\}) \]
\[ E \Rightarrow \mathcal{A} : \text{PK}_{\text{mix}}(k^i) \rightarrow E_{\text{keygen}}(t_0, \{\text{Mix}, \mathcal{A}_k, k^i\}) \]
\[ E \Rightarrow \mathcal{A} : \text{PK}_{\text{mix}}(k^i) \rightarrow E_{\text{keygen}}(t_0, \{\text{Mix}, \mathcal{A}_k, k^i\}) \]

// Enc calls for \( t_0, t_1 \):
\[ E \rightarrow \mathcal{A} : (c_0, c_1) \rightarrow E_{\text{mix}}(\{1, k^i, t_0, t_1, b\}) \]

// Mix:
\[ E \Rightarrow \mathcal{A} : \pi(k^i) \rightarrow \text{Perm}(n) \]
\[ E \Rightarrow \mathcal{A} : \pi(k^i) \rightarrow E_{\text{mix}}(\{1, k^i, t_0, t_1, b\}) \]

// OTTraceOut call:
\[ E \Rightarrow \mathcal{A} : \text{OTTraceOut}(I, J) \]
\[ E \Rightarrow \mathcal{A} : \text{OTTraceOut}(I, J) \]

Figure 22: E10: Obtain \( \tilde{S}', \tilde{e}'', \tilde{r}'' \) during DB-RSM without decryption.
Figure 23: $E_{11}$: Send encryptions of zeros in place of all $E_{Pa}$.
Traceable mixnets

\[ E_{\text{mix}}(t^k_i, \pi, h_1, h_2, \delta) : \]

// Keygen:
\[ E : \]
\[ E := E_{\text{Keygen}}(1^k) \]

\[ \mathcal{A} \rightarrow E_{\text{Keygen}}(1^k, (M^k_1)_k^k) \]

\[ E := E_{\text{Keygen}}(1^k, (M^k_1)_k^k) \]

\[ E := E_{\text{Keygen}}(1^k, (M^k_1)_k^k) \]

\[ E := E_{\text{Keygen}}(1^k, (M^k_1)_k^k) \]

// Enc calls for \( t_i, t_i' \):
\[ E \]
\[ \text{for } i \in [1, n] : \]
\[ E_i \leftarrow E_{\text{Enc}}(pk_\mathcal{A}) \]
\[ r_i \leftarrow \mathcal{R}_{\mathbb{Z}_q} \]
\[ \pi'_i \leftarrow \mathcal{R}_{\mathbb{Z}_q} \]
\[ \pi'_{i'} \leftarrow \mathcal{R}_{\mathbb{Z}_q} \]
\[ c_i \leftarrow E_{\text{Enc}}(pk_\mathcal{A}, \pi'_i) \]
\[ c_i \leftarrow E_{\text{Enc}}(pk_\mathcal{A}, \pi'_i) \]
\[ c_i \leftarrow E_{\text{Enc}}(pk_\mathcal{A}, \pi'_i) \]

\[ \text{endfor} \]

// Mix:
\[ E \]
\[ \pi(i)^k \leftarrow \mathcal{R}(\mathbb{Z}_q) \]
\[ \mathcal{A} \rightarrow E : \]
\[ \mathcal{A} \rightarrow E : \]
\[ \mathcal{A} \rightarrow E : \]
\[ \mathcal{A} \rightarrow E : \]
\[ \mathcal{A} \rightarrow E : \]

// \( O\text{TraceOut} \):
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]
\[ \mathcal{A} \rightarrow \mathcal{A} : \]

// endfor

Figure 24: \( E_{\text{EG}} \) Send encryptions of \( g_1^k \) in place of all \( E_{\text{EG}} \) encryptions.
\[ E \left( (1, k^*, \theta_0, \theta_1, b) \right) : \]

// Keygen:
\[ E \left( \text{pk}, s_k(k^*) \right) \leftarrow E\text{Keygen}(1^t) \]
\[ E \left( \text{pk}_{\text{EC}}, s_k(k^*) \right) 
\]

// simulate the entire DPK
\[ \text{pk}(k^*), \text{sk}(k^*) \leftarrow E\text{Keygen}(1^t, (\text{M}^k_k)_{k \leftarrow k^*}) \]

// simulate the first component of the DPK
\[ E \left( \text{pk}, s_k(k^*) \right) \leftarrow E\text{Keygen}(1^t, (\text{M}^k_k)_{k \leftarrow k^*}) \]

// simulate \( \text{pk}_{\text{EC}}, \text{pk}_{\text{PA}} \)
\[ E \left( \text{pk}(k^*), \text{sk}(k^*) \right) \]

// Encalls for \( b_0, b_1 \):
\[ \text{for } i \in [b_0, b_1]: \]
\[ E_i : \]
\[ \hat{e}_i \leftarrow \text{Enc}(\text{pk}_{\text{PA}}, 0) \]
\[ r_i \leftarrow \text{rand} \]
\[ y_i \leftarrow g_i^{\hat{e}_i} \cdot r_i \leftarrow \text{Enc}(\text{pk}_{\text{PA}}, 0) \]
\[ (\gamma_i, h_i)_{k \leftarrow k^*} \leftarrow \text{Enc}(\text{pk}(k^*), (\text{M}^k_k)_{k \leftarrow k^*}) \]
\[ \sigma_i := \gamma_i \cdot \Pi_{\text{Enc}(\text{pk}(k^*), (\text{M}^k_k)_{k \leftarrow k^*})} \]
\[ E \left( E_i, \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \right) \]
\[ \text{endfor} \]

// Mix:
\[ E \left( \pi(k^*) \right) \leftarrow \text{Perm} \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{PA}}, 0) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ \text{for } j \in [\pi_1, \pi_2]: \]
\[ E_j \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{PA}}, 0) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ \text{endfor} \]

// OTracecall:
\[ E \left( \pi(k^*) \right): \]
\[ \text{assert } \text{in} \in \pi_1 \iff \text{in} \in \pi_1 \]
\[ E \left( \pi(k^*) \right): \]
\[ \text{DB-SM call (repeat for } n \text{ \& } f) \]
\[ E \left( \pi(k^*) \right): \]
\[ \text{shuffle} \]
\[ E \left( \pi(k^*) \right): \]
\[ \text{Enc}(\text{pk}_{\text{PA}}, 0) \]
\[ \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]

// homomorphic binding
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]

// threshold decryption
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]

// simulate the entire DPK
\[ z_i(k^*) \leftarrow \gamma_i \cdot h_i \left( r_i \mid \text{Enc}(\text{pk}(k^*), (\text{M}^k_k)_{k \leftarrow k^*}) \right) \]
\[ \text{for } i \in [b_0, b_1]: \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{PA}}, 0) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]
\[ E \left( \pi(k^*) \right) \leftarrow \text{Enc}(\text{pk}_{\text{EC}}, \text{sk}(k^*)) \]

// simulate the first component of the DPK
\[ z_i(k^*) \leftarrow \gamma_i \cdot h_i \left( r_i \mid \text{Enc}(\text{pk}(k^*), (\text{M}^k_k)_{k \leftarrow k^*}) \right) \]
\[ \text{endfor} \]

---

Agrawal et al.
Enc(1, k*, i0, i1, b) -

// Keygen:
\[ E \rightarrow \mathcal{K} \rightarrow \mathcal{E}_{\text{Keygen}}^{(1)}(1) \]
\[ \mathcal{E}_{\text{Keygen}}^{(1)}(1, (\mathcal{M}_{\text{E}}^k)_{\text{enc}}) \]
\[ \mathcal{E}_{\text{Keygen}}^{(1)}(1, (\mathcal{M}_{\text{E}}^k)_{\text{enc}}) \]
\[ \mathcal{E}_{\text{Keygen}}^{(1)}(1, (\mathcal{M}_{\text{E}}^k)_{\text{enc}}) \]
\[ \mathcal{E}_{\text{Keygen}}^{(1)}(1, (\mathcal{M}_{\text{E}}^k)_{\text{enc}}) \]
\[ \mathcal{E}_{\text{Keygen}}^{(1)}(1, (\mathcal{M}_{\text{E}}^k)_{\text{enc}}) \]
\[ \mathcal{E}_{\text{Keygen}}^{(1)}(1, (\mathcal{M}_{\text{E}}^k)_{\text{enc}}) \]

// Enc calls for \( b \):
\[ \text{for } i \in \{1, \ldots, l\} \text{ } : \]
\[ e_i \leftarrow \mathcal{E}_{\text{Enc}}(\text{pk}_i, 0) \]
\[ r_i \leftarrow \mathcal{E}_{\text{Enc}}(\text{pk}_i, 0) \]
\[ r_i \leftarrow \mathcal{E}_{\text{Enc}}(\text{pk}_i, 0) \]
\[ r_i \leftarrow \mathcal{E}_{\text{Enc}}(\text{pk}_i, 0) \]
\[ r_i \leftarrow \mathcal{E}_{\text{Enc}}(\text{pk}_i, 0) \]
\[ r_i \leftarrow \mathcal{E}_{\text{Enc}}(\text{pk}_i, 0) \]

// Mix:
\[ \text{for } i \in \{1, \ldots, l\} \text{ } : \]
\[ \text{for } i \in \{1, \ldots, l\} \text{ } : \]
\[ \text{for } i \in \{1, \ldots, l\} \text{ } : \]
\[ \text{for } i \in \{1, \ldots, l\} \text{ } : \]
\[ \text{for } i \in \{1, \ldots, l\} \text{ } : \]

// OTTrace call:
\[ \mathcal{E}_{\text{TraceOut}(1,J)} \]
\[ \mathcal{E}_{\text{TraceOut}(1,J)} \]
\[ \mathcal{E}_{\text{TraceOut}(1,J)} \]
\[ \mathcal{E}_{\text{TraceOut}(1,J)} \]
\[ \mathcal{E}_{\text{TraceOut}(1,J)} \]

// OTTrace out:
\[ \text{else: } \text{simulate the first two components of the DPK} \]
\[ \text{else: } \text{simulate the first two components of the DPK} \]

\[ \text{endfor} \]
\(E_{\text{Mix}}(1^k, k, i_0, i_1, b)\):

- **Keygen:**
  \(E_{\text{Mix}}(1^k) \rightarrow \text{Pk} = (\text{Pk}_0, \text{Pk}_1)\)
  \(\text{Key}_0, \text{Key}_1 \leftarrow \text{KeyGen}(1^k)\)

- **Enc:**
  \(E_{\text{Mix}}(1^k) \rightarrow \text{Enc} = (\text{Enc}_0, \text{Enc}_1)\)
  \(\text{Enc}(m) \leftarrow \text{KeyGen}(1^k, (\text{Key}_0, \text{Key}_1))\)

- **Decrypt:**
  \(E_{\text{Mix}}(1^k) \rightarrow \text{Dec} = (\text{Dec}_0, \text{Dec}_1)\)
  \(\text{Dec}(c) \leftarrow \text{KeyGen}(1^k, (\text{Key}_0, \text{Key}_1))\)

**Figure 28:** \(E_{\text{Mix}}(1^k)\) uses random group element for \(\hat{z}_1^{(k)}\).
Figure 29: $E_{G}: \text{Set } v_{i}^{(k_{i})}, r_{i}^{(k_{i})}, v_{i}^{(c_{i})}, r_{i}^{(c_{i})}$ as zeros (also abstracted out the DPK simulation steps during DB-RSM).
Traceable mixnets

\[ E_{ij}(1^k, i_0, i_1, i) : \]

// Keygen:
\[ E \rightarrow \mathcal{A} \rightarrow \text{Keygen}(1^k) \]
\[ E \rightarrow \mathcal{A} \rightarrow \text{Enc}_{EG}(\text{Keygen}(1^k), (M_k^k)_{k \neq k}) \]
\[ E \rightarrow \mathcal{A} \rightarrow \text{Pk}_k \rightarrow \text{Enc}_i(1^k, (M_k^k)_{k \neq k}) \]

// Enc calls for \( i_0, i_1, k \):
\[ E : \]
\[ \text{for } r \in \text{Rand}(\text{Enc}(\text{pk}_k, 0)) : \]
\[ \begin{align*}
& \text{Get } (v_k, r_k) \text{ from the Enc calls for } i_0, i_1, k \text{ for } j \in [n] \setminus \{i, j\} ; \\
& \text{Get } v_j^j, r_j^j \text{ from } \mathcal{A}'s \text{ random tape and input for sender } S_j \end{align*} \]

// OTraceOut call:
\[ \mathcal{A} \rightarrow E \rightarrow \text{OTraceOut}(1, j) \]
\[ E \rightarrow \mathcal{A} \rightarrow \text{assert } \exists i : s_j \subseteq i \end{align*} \]
\[ E \rightarrow \mathcal{A} \rightarrow \text{Enc}_i(1^k) \rightarrow \text{DB-RSM call (repeat for } [n] \setminus j) \]
\[ E \rightarrow \mathcal{A} \rightarrow \text{Get } \theta \rightarrow \text{DB-RSM call (repeat for } [n] \setminus j) \]
\[ E \rightarrow \mathcal{A} \rightarrow \text{Get } \theta \rightarrow \text{DB-RSM call (repeat for } [n] \setminus j) \]

Figure 30: E17: Set \( y_{i_0}, y_{i_1} \) as commitments of zeros.
Agrawal et al.

Figure 31: $E_{\hat{T}}$: Replace $\hat{\sigma}_i$, $\hat{\sigma}_j$ by elements randomly drawn from $\mathbb{G}_1$. 

Traceable mixnets

\[ E_i \approx (1^i, k^i, b_i, i_0, i_1) \]

// Keygen:
\[
\begin{align*}
E &\leftarrow \mathcal{A} \\
&\quad \leftarrow \mathcal{P}_{\mathcal{E}} \mathcal{G}_k \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{G}_k \mathcal{G}_k \leftarrow \mathcal{E}_{\mathcal{D}} \mathcal{G}_k \mathcal{G}_k
\end{align*}
\]

// Encalls for \( i_0, i_1 \):
\[
\begin{align*}
&\text{for } i \in \{ i_0, i_1 \}:
&\quad c_i \leftarrow \mathcal{E}_i \mathcal{P}_{\mathcal{E}} \mathcal{P}_k \mathcal{P}_r
&\quad r_1 \leftarrow \mathcal{S}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \beta_i \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \gamma_i \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_i^k, r_i^k) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_i^{k^i}, r_i^{k^i}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_i^{k^i}, r_i^{k^i}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_i^{k^i}, r_i^{k^i}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
\end{align*}
\]

// Mix:
\[
\begin{align*}
&\text{for } j \in \{ i_0, i_1 \}: \\
&\quad c_j \leftarrow \mathcal{E}_j \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad r_1 \leftarrow \mathcal{S}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \beta_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \gamma_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^k, r_j^k) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
\end{align*}
\]

// OTTraceOut call:
\[
\begin{align*}
\mathcal{A} &\rightarrow \mathcal{E} \\
&\quad \text{OTTraceOut}(I, J)
\end{align*}
\]

For \( j \in \{ i_0, i_1 \} \):
\[
\begin{align*}
&\text{assert } c_j \in \mathcal{E}_j \leftarrow \mathcal{E}_j
&\quad \text{for } j \in \{ i_0, i_1 \}:
&\quad \beta_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \gamma_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^k, r_j^k) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
\end{align*}
\]

For \( j \in \{ i_0, i_1 \} \):
\[
\begin{align*}
&\text{assert } c_j \in \mathcal{E}_j \leftarrow \mathcal{E}_j
&\quad \text{for } j \in \{ i_0, i_1 \}:
&\quad \beta_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \gamma_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^k, r_j^k) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
\end{align*}
\]

For \( j \in \{ i_0, i_1 \} \):
\[
\begin{align*}
&\text{assert } c_j \in \mathcal{E}_j \leftarrow \mathcal{E}_j
&\quad \text{for } j \in \{ i_0, i_1 \}:
&\quad \beta_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad \gamma_j \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^k, r_j^k) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
&\quad (\psi_j^{k^j}, r_j^{k^j}) \leftarrow \mathcal{E}_{\mathcal{E}} \mathcal{E}_{\mathcal{D}} \mathcal{P}_{\mathcal{E}} \mathcal{P}_r
\end{align*}
\]

Figure 32: \( E_{ij} \): Replace \( S_j, c', c'' \) for \( j \in \{ i_0, i_1 \} \) by randomly drawn elements.