A Mix-Net From Any CCA2 Secure Cryptosystem

Shahram Khazaei[1], Tal Moran[2], and Douglas Wikström[1]

[1] KTH Royal Institute of Technology
[2] IDC Herzliya

Abstract. We construct a provably secure mix-net from any CCA2 secure cryptosystem. The mix-net is secure against active adversaries that statically corrupt less than λ out of k mix-servers, where λ is a threshold parameter, and it is robust provided that at most $\min(\lambda - 1, k - \lambda)$ mix-servers are corrupted.
The main component of our construction is a mix-net that outputs the correct result if all mix-servers behaved honestly, and aborts with probability $1-O(H^{-(t-1)})$ otherwise (without disclosing anything about the inputs), where t is an auxiliary security parameter and H is the number of honest parties. The running time of this protocol for long messages is roughly $3tc$, where c is the running time of Chaum's mix-net (1981).

1 Introduction

A *mix-net*, introduced by Chaum in 1981 [2], is a tool to provide anonymity for a group of senders. The main application is electronic voting, in which each sender submits an encrypted vote and the mix-net then outputs the votes in sorted order. Mix-nets have also found applications in other areas, e.g., anonymous web browsing [6], payment systems [13] and even as a building-block for secure multiparty computation [10].

A mix-net is constructed as a cryptographic protocol by invoking a set of *mix-servers*, arranged in a series. The original mix-net proposed by Chaum works as follows. To set up, each mix-server publishes a public key for an encryption system. Each sender then publishes a "wrapped" message with several layers of encryption: starting with the innermost layer—an encryption of her plaintext message using the last mix-server's public key—and ending with the outermost layer, encrypted using the first mix-server's public key. Once all senders have published their encrypted inputs, the mixing stage begins. In turn, each mix-server receives the encrypted values output from the previous server, "peels off" a layer of encryption, i.e., decrypts the values using his private key, sorts the decrypted values and passes them on to the next mix-server in the chain. The output of the final mix-server is the sorted list of the senders' original inputs.

Chaum's mix-net hides the correspondence between the input ciphertexts and the output plaintexts, but even a single mix-server can undetectably modify the output or refuse to take part in the protocol (forcing the protocol to abort without output). These drawbacks have been addressed in previous work. The most widely researched line of work is based on the idea of *re-encryption mixes* (originally proposed by Park, Itoh and Kurosawa [19]); these rely on homomorphic encryption schemes whose ciphertexts can be "re-randomized". Using the homomorphic properties of the encryption scheme, it is possible to generate very efficient zero-knowledge proofs that the mixing was performed correctly (e.g., Neff [16] or Furukawa and Sako [4]). While the state-of-the-art re-encryption mixes are both provably secure and efficient for short inputs, their reliance on homomorphic properties limits them to a few specific encryption schemes.

1.1 Our Contribution

In this paper, we propose a new, efficient mix-net protocol that satisfies several highly-desirable properties:

- *Minimal Cryptographic Assumptions*. Our protocol can be based on any CCA2-secure cryptosystem, without requiring additional assumptions. In particular, we do not require the underlying encryption to have homomorphic properties.
  While interesting from a theoretical standpoint, this also has clear advantages in practice, as it gives greater flexibility in the choice of encryption scheme. For example, all currently practical homomorphic encryption schemes are susceptible to attacks from quantum computers. Although we do not currently know how to build quantum computers, it is important to take this vulnerability into account when using a mix-net as part of an electronic election scheme: ballot privacy is often required to be preserved for decades—these timeframes may be long enough for the development of a working quantum computer.
  Furthermore, the flexibility in the choice of encryption scheme makes it easy to deal efficiently with long inputs, while there do exist mix-nets that can deal with long inputs efficiently [11,17,5], these mixes require even more specialized encryption schemes tailored specifically to that purpose.
- *Provable Security.* Many of the existing mixing protocols do not have formal proofs of security. This may seem like a purely theoretical concern, but the history of cryptographic protocols, and mix-nets in particular, shows that there is good reason to distrust heuristic approaches. A notable example of this is the *Randomized Partial Checking* (RPC) scheme of Jakobsson, Juels and Rivest [12] (our main "competitor" in the field of generic CCA2-based mixes). The RPC scheme (and related constructions) have been around for over a decade, and have already been used in binding elections; however, recent work by Khazaei and Wikström [14] shows that RPC contains a subtle but serious security flaw, which was consistently missed in implementations. Other examples abound (see Section 1.2 for more).
  In contrast, our protocol is proven secure in the Universal Composability framework [1], a very strong notion of security that holds even when arbitrary additional protocols are run concurrently. (If a cryptosystem which allows recovering the randomness from a ciphertext using the secret key is used to implement the (non zero-knowledge) proof of correct decryption, then the result only holds in the stand-alone setting.)
- *Full Security.* The RPC scheme gains efficiency by relaxing slightly the security requirements. It prevents corrupt mix-servers from undetectably modifying *many* inputs of honest senders, but a malicious server can succeed in changing a constant number of inputs with non-negligible probability. For some uses, this may

not be acceptable. RPC also relaxes the privacy guarantees: while the exact correspondence between senders and their inputs is hidden, some information may still be leaked. Our protocol, with comparable or better efficiency, provides full simulation-based security.

Our protocol is based on a new technique we call *Trip-Wire Tracing* (TWT). Our main idea is to do away with zero-knowledge proofs (that would be costly for a generic cryptosystem) used by existing protocols to guarantee correctness and replace them with a virtual "trip wire" system: we insert "fake" inputs into the mix to act as trip wires for catching misbehaving mix-servers (for more details, see Section 2).

Security Guarantees and Assumptions. The protocol preserves privacy and correctness against active adversaries that statically corrupt less than λ mix-servers, where λ is a threshold parameter, and it is robust provided that at most $\min(\lambda - 1, k - \lambda)$ mix-servers are corrupted, where k is the number of mix-servers. As for all other mix-nets in the literature, we assume the existence of an ideal bulletin board functionality (this is equivalent to a broadcast channel). We also need an ideal functionality for shared key generation. In the general case (when we can only assume a generic CCA2-secure cryptosystem without any additional structure), this functionality would have to be implemented using general MPC. However, if the chosen cryptosystem does have a more efficient shared-key-generation protocol, it can be used instead (in any case, the bulk of the work can always be carried out offline, in a preliminary key generation phase).

Finally, we need a functionality for proving that a ciphertext is correctly decrypted, but it suffices that this protocol hides the secret key. This functionality can be realized trivially if the cryptosystem allows recovery of the randomness (used to form the ciphertext) using the secret key. In any case this protocol is only used to identify corrupted parties and mix-servers, so during normal operation it is not used at all.

Limitations of our protocol. Our construction essentially uses privacy to ensure correctness (by hiding the "trip-wires" from malicious mix servers). Because a threshold coalition of malicious servers can always violate privacy, our protocol loses correctness as well in this case. This implies that our protocol cannot be "universally verifiable" (i.e., verifiable by third parties who do not trust any of the mix servers). In comparison, the state-of-the-art mix-nets based on homomorphic cryptosystems can provide integrity (but not privacy) even if all mix-servers are corrupt.

We remark that RPC only allows a restricted form of universal verifiability, i.e., its relaxed correctness degrades further and allows an adversary that controls all mix-servers to undetectably replace a notable number of ciphertexts.

1.2 Related Work

The literature on mix-nets and verifiable shuffling is extensive. Below, we mention a small sample of particularly relevant works. Park, Itoh and Kurosawa [19] introduced re-encryption mixes as a way to improve efficiency—the size of the ciphertexts and the amount of work performed by senders does not depend on the number of mix-servers. Sako and Kilian constructed the first universally-verifiable mix-net [22], where senders can verify that the entire shuffle was performed correctly (and not just that their own input was included in the output). Sako and Kilian's construction was based on cut-and-choose zero-knowledge proofs; Neff [16] and Furukawa and Sako [4] gave much more efficient zero-knowledge proofs of shuffle for homomorphic cryptosystems. Many of the works in the field aim to improve the efficiency of the mix-net. Our construction includes ideas that appear in several previous papers: Jakobsson used the idea of "dummy inputs" [9] and "repetition" [8] to increase correctness (although in a different way than we do). Golle, Zong, Boneh, Jakobsson and Juels [7] considered mix-nets that are "optimistic" (i.e., can be much more efficient in the case that no errors occur).

On the importance of formal proofs. A recurring tale in the history of mix-net design is the proposal of a mix-net construction followed by discovery of security flaws. Following Chaum's seminal paper [2], Pfitzmann and Pfitzmann pointed out that Chaumian mixes are vulnerable to attack if the encryption scheme used is malleable [21]. The mix-net of Park et al. [19] was also shown to be vulnerable to similar attacks [20]. Jakobsson's scheme of [8] was broken in [3]. His other scheme [9], was broken by Mitomo and Kurosawa [15], who also suggested a fix; this in turn, in addition to the schemes of Jakobsson and Juels [11], of Golle, Zong, Boneh, Jakobsson and Juels [7] were all shown to be vulnerable (to various attacks) by Wikström [23].

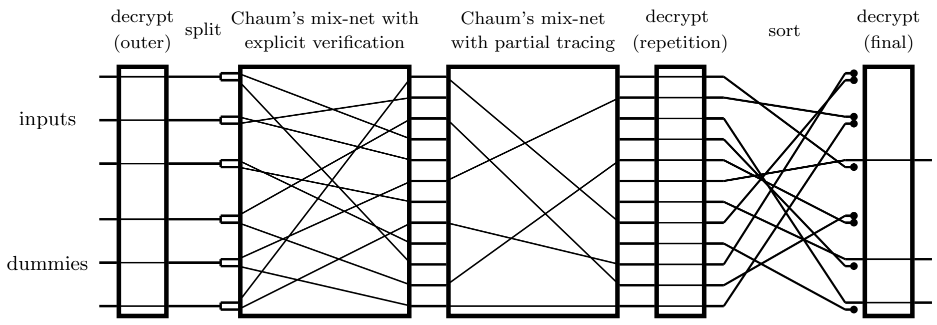
While a formal proof of security is not an iron-clad guarantee that no vulnerabilities will ever be found (proofs may have subtle errors, and assumptions may be shown to be wrong), they do significantly improve the trust in the security of a cryptographic scheme. In fact, the need for some of the components of our protocol only became evident during the analysis of the protocol.

2 Informal Description of Our Protocol

We begin with an overview of our mix-net protocol and some intuition for why this protocol is secure. The main component of our construction is a mix-net that outputs the correct result if all mix-servers behave honestly, and aborts with overwhelming probability otherwise—without disclosing anything about the inputs. At a high level, our mix-net with abort protocol is a Chaumian mix-net with added verification. It is parametrized with an auxiliary security parameter t and uses two Chaumian mix-nets in sequence (one with "explicit verification" and one with "partial tracing") and three additional layers of encryption (labeled as "final", "repetition" and "outer"). Figure 1 presents a schematic of our protocol.

Each sender encodes her message as a bundle of t ciphertexts: First, she encrypts her plaintext message using the public key of the "final" layer of encryption and makes t identical copies of it. Next, each copy is further encrypted using the public key of the "repetition" encryption layer and then under the public keys of the mix-servers in the two Chaumian mix-nets. Finally, the t encryptions are concatenated and encrypted using the public key of the "outer" encryption layer. To generate the final list of inputs to the mix-net, each mix-server adds a "dummy" encryption of zero to the list of inputs submitted by the senders (the dummy input is constructed using the same operations as the real inputs).

Once all parties have submitted their bundles, the decryptions proceed in the reverse order. If all the parties are honest, there will be t identical copies of each innermost ciphertext before the final decryption takes place. In this case the dummies are traced and removed, the duplicates are ignored, and only one instance of each sender's innermost ciphertext is decrypted. We stress that this is only an outline of the protocol. Additional measures are taken for ensuring correctness and privacy.



decrypt (outer)
split
Chaum's mix-net with explicit verification
Chaum's mix-net with partial tracing
decrypt (repetition)
sort
decrypt (final)
inputs
dummies

Fig. 1. Execution of Protocol 4 with $N = 3$ senders, $k = 3$ mix-servers, and $t = 2$ repetitions, where all parties are honest. Each party submits a bundle of two ciphertexts containing identical innermost ciphertexts. The bundle is decrypted and split into two ciphertexts. All ciphertexts are then individually shuffled in the two instances of Chaum's mix-net. Then the first is verified explicitly (revealing the permutation), the dummy ciphertexts are traced in the second (revealing the paths of the dummies) and the output is decrypted and verified to contain t copies of each ciphertext. If all tests passed, then a final round of decryption recovers the plaintexts.

To help give the intuition for our construction, we will describe a sequence of attacks on the Chaumian mix-net and our corresponding modifications to the protocol that prevent them. The final protocol is a composition of all these modifications. We start with a "core" Chaumian mix, which ends up—after slight modifications—as the box labeled "Chaum's mix-net with partial tracing" in Figure 1. We call a set of ciphertexts containing identical innermost ciphertexts a *copyset*.

1. *Elementary Error Handling*. The first type of attack we consider is the introduction of "simple" errors that are publicly detectable. Invalid ciphertexts are simply ignored. If there are duplicates of a ciphertext in the input to a mix-server, then exactly one copy is considered part of the input and the rest is ignored.
2. *Replication*. In a Chaumian mix-net, any corrupt mix-server can change the output undetectably by replacing an output ciphertext with a new one generated by the malicious server (this new ciphertext can be completely valid, except for not being a decryption of any input ciphertext). To prevent this attack, each sender submits t independently formed ciphertexts of her message to the Chaumian mix-net.
   To see why this replication technique helps prevent replacement attacks, consider a corrupt mix-server that appears between two honest mix-servers in the mix-net chain. In this case, the corrupt mix-server cannot identify which of the ciphertexts encrypts the same messages due to the following two reasons.

(a) He does not know the secret key of the succeeding honest mix-sever, and hence he can not fully decrypt the received ciphertexts and distinguish the copysets based on the final decrypted values.

(b) The preceeding honest mix-server randomly permuted all of the ciphertexts and hence he does not know which ciphertext originated from which sender.

We prove that, if a CCA2 secure cryptosystem is used, t is sufficiently large, and all messages are randomly chosen, then no efficient adversary between two honest mix-servers can replace a proper subset of senders messages without detection.

3. *Replication Cryptosystem*. In a Chaumian mix-net, the last mix-server learns the final output before anyone else. Thus, even with the replication trick, the final mix-server can clearly cheat, since he can identify all copies of a plaintext. To prevent this attack we modify the protocol by adding an additional "repetition" layer of encryption, using a public key for which the secret key is shared between the mix-servers.

   Think of this as running the Chaumian mix-net on *encrypted* inputs rather than plaintexts, i.e., each sender makes t encryptions of her input with the shared public key of the "repetition" layer, and then uses the encrypted values as her "plaintexts". The output of the Chaumian mix-net is a list of ciphertexts encrypted with the shared public key, which prevents the last mix-server from identifying identical plaintexts and replacing all copies of a subset of the plaintexts. At the end of the mixing, the shared secret key is recovered and decryption is performed publicly. In Figure 1, this decryption step is the box labeled "decrypt (repetition)" right after the Chaum's mix-net with partial tracing.

4. *Additional Mix-net with Explicit Verification*. The first mix-server knows how to partition the input messages into copysets (since he receives the messages directly from the senders), hence he can replace all copies of a given plaintext undetectably. To prevent this attack, we add a new, unmodified Chaumian mix-net (the box labeled "Chaum's mix-net with explicit verification" in Figure 1) between the senders and the first mix-server in the "main" mix-net. Recall that the Chaumian mix-net does not give any correctness guarantees, but it does guarantee privacy if even a single mix-server is honest. This is exactly what we need to put the first mix-server in the Chaumian mix-net with partial tracing on an equal footing with the others in the chain.

   We rely on the privacy of the first Chaumian mix-net only to obtain *correctness* via replication. Therefore, once the second Chaumian mix-net finishes his process of mixing, the mix-servers can reveal the secret keys for the first Chaumian mix-net and verify its correctness completely (hence the name "mix-net with explicit verification"). If the verification fails, the guilty mix-server is publicly evident.

5. *Dummy Values*. If a corrupt mix-server in the second Chaumian mix-net wishes to replace a proper subset of senders' messages, he must guess the positions of the copysets, but he can still undetectably replace *all* of the inputs with his own values. To prevent this, we have every mix-server add a "dummy" value to the inputs of the mix-net. These dummy values are treated identically to the senders' inputs. Thus, any mix-server attempting to replace the entire list of inputs would also be replacing all dummy values. The mix-servers can "trace forward" the dummy values and remove them from the final decrypted list if the trace completes successfully. There

is no privacy requirement for the dummy values; therefore, each mix-server can simply reveal all the randomness used in the encryption of the initial dummy values. This reveals all the internal layers of encryption in a verifiable way, allowing everyone to find the corresponding ciphertexts in each stage of the mix-net.

6. *Replication Verification and Error Tracing*. We need to handle the case where some of the final values, after recovering the shared secret key of the "repetition" encryption layer, do not have exactly t copies.

   We now add another step to the protocol after decryption using the shared secret: replication verification (this occurs in the part labeled "sort" in Figure 1) and error tracing (this is not shown in Figure 1 since we assume all parties are honest). In the replication verification step, the (honest) mix-servers verify that there are exactly t duplicates of every output value. This clearly is the case if all servers and senders are honest. If the verification fails, however, we need to figure out who is to blame so that we can continue the protocol if it was just a corrupt sender. To do this, we need to trace errors through the system in two ways:

   (a) *Backwards Tracing*. After determining the messages with more or less than t duplicates, we trace them backwards to identify their original senders. Since each mix-server knows his own permutation, the backwards trace is easy to do: each mix-server in turn (starting from the last one and going backwards) publishes the "paths" taken by the traced messages along with a proof that the decryption was performed correctly. If a broken copyset being traced contains ciphertexts that were introduced by a cheating mix-server (i.e., ciphertexts that are not valid decryptions of the mix-server's inputs), the mix-server will not be able to provide a valid trace and will be identified as a cheater at this point.

   (b) *Forward Tracing*. If all the broken copysets were successfully traced back to their sender, there are still two remaining possibilities for casting blame:

      i. The mix-servers behaved honestly, and bad copysets were submitted by corrupt senders.

      ii. At least one ciphertext submitted by an honest party was replaced by a corrupted mix-server. (This could be the case even if no cheating was discovered during backwards tracing. To see this, consider the case that a corrupt mix-server arbitrarily chooses t ciphertexts from honest senders and replaces them with a valid copyset.)

      To distinguish these two cases, we identify the senders from which the broken copysets originated, and "trace forward" *all* the messages of these senders. This is done similarly to the backwards tracing, but in reverse: starting from the first mix-server and going forwards, each one in turn publishes the paths taken by the traced messages along with a proof of correct decryption. If a mix-server cheated, he will not be able to provide a valid trace—hence he will be fingered as the culprit. On the other hand, if only the identified senders were cheating (e.g., by not encrypting a valid copyset in the first place), we will be able to trace the messages all the way to the output.

   If the backwards and forward tracings complete successfully without identifying a mix-server as culprit, the ciphertexts of the corrupt senders are removed from the output (otherwise, the protocol outputs the identity of a guilty mix-server and aborts).

7. *Final Cryptosystem.* As we have described in Step 6, to catch a misbehaving mix-server we must sometimes trace messages of honest users through the system. Although we abort the protocol in this case, we must still preserve the honest senders' privacy. Therefore, we protect the senders' messages with an additional layer of encryption (the last box labeled "decrypt (final)"). That is, a sender first encrypts her message under the "final" public key and uses this encrypted message as an input to the protocol as described so far. This innermost encryption layer is jointly decrypted only if the protocol does not abort. If the protocol does abort, only the encrypted values are revealed and privacy is protected by the final layer of encryption. The "final" layer of encryption also guarantees that the "plaintexts" of the protocol we have sketched so far (without the "final" layer) are distinct for all honest senders (and different from corrupt senders) with overwhelming probability.
8. *Outer Cryptosystem.* The protocol is still vulnerable to a subtle attack that uses the error-tracing mechanism itself to violate sender privacy. The problem is that tracing occurs in two additional indistinguishable cases:
   (a) Corrupt senders collude to create "colliding" ciphertexts (i.e., after removing some layers of encryption, the resulting ciphertexts are identical).
   (b) Corrupt mix-server(s) collude with corrupt sender(s) to copy some of an honest sender's ciphertexts.

   In both cases tracing will complete successfully (since no inputs were replaced in the middle of the mix-net). Because in the first case the mix-servers are all honest, we cannot simply abort if this situation occurs. On the other hand, in the second case, we may be forced to trace an honest ciphertext from beginning to end (we trace a broken ciphertext back to a corrupt sender, then trace forward all of that sender's inputs, which include a copy of an honest ciphertext). Since the corrupt sender knows the identity of the sender from whom the ciphertext was copied, if we decrypt that value the honest sender's privacy is violated.

   To prevent this, we add an "outer" layer of encryption (the box labeled "decrypt (outer)"): under a public-key whose secret key is shared by all the mix-servers, each sender formes a single "bundled" ciphertext. After all the ciphertext bundles are received, the mix-servers recover the secret key of the outer cryptosystem and the bundles are publicly decrypted and "split" into the separate copyset ciphertexts. This countermeasure works due to the CCA2 security of the cryptosystem: CCA2 security ensures that no corrupt coalition of mix-servers and senders can make partial copies of an honest sender's copyset: either they copy a bundle in its entirety (in which case they are removed due to being duplicates) or they create a bundle that is completely independent of the honest senders' bundles (in which case the probability of a collision is negligible).

3 Notation

For an integer e, we denote the set $\{1, \ldots, e\}$ by $[1, e]$. The security parameter, n, represented in unary, is an implicit input to all protocols and functionalities. Whenever we say a quantity ε is negligible, we mean that it is negligible in the security parameter, i.e., for every $c > 0$ we have $\varepsilon(n) < n^{-c}$ for all but finitely many n. We write $x \in a$

for a list $a = (a_1, \ldots, a_e)$ if and only if $x \in \{a_1, \ldots, a_e\}$. The length of a is denoted by $|a|$. For any index set $I \subset [1, e]$ of size ℓ, we write $(a_i)_{i \in I} = (a_{i_1}, \ldots, a_{i_\ell})$, where $I = \{i_1, \ldots, i_\ell\}$ with $i_1 < i_2 < \cdots < i_\ell$. We say that a list $b = (b_1, \ldots, b_\ell)$ is a subset of a and write $b \subset a$, if and only if $\{b_1, \ldots, b_\ell\} \subset \{a_1, \ldots, a_e\}$ (with multiplicity). We use $\mathsf{Sort}(a)$ to denote the lexicographically sorted list of elements from a (with multiplicity). We write $a \setminus b$ for $\mathsf{Sort}(\{a_1, \ldots, a_e\} \setminus \{b_1, \ldots, b_\ell\})$ (with multiplicity in the set difference). We also write $a \circ b$ for the concatenated list $(a_1, \ldots, a_e, b_1, \ldots, b_\ell)$. We denote by $\mathsf{Unique}(a)$ the sorted list where each element of a appears exactly once.

We denote a cryptosystem by $\mathcal{CS} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, where Gen, Enc, and Dec denote the key generation algorithm, the encryption algorithm, and the decryption algorithm respectively. To deal with nested encryption as needed in a Chaumian mix-net, we simply assume that a plaintext of any length can be encrypted, but that indistinguishability only holds for plaintexts of the same length. We write $c = \mathsf{Enc}_{pk}(m, r)$ for the encryption of a plaintext m using randomness r, and $\mathsf{Dec}_{sk}(c) = m$ for the decryption of a ciphertext c. We often view Enc as a probabilistic algorithm and drop r from our notation. We assume that malformed ciphertexts are decrypted to a special symbol different from all normal plaintexts.

We extend our notation to lists of plaintexts, ciphertexts and keys as follows. For a plaintext $m = (m_1, \ldots, m_e)$ and a key pair (pk, sk) with $pk = (pk_1, \ldots, pk_\ell)$ and $sk = (sk_1, \ldots, sk_\ell)$ we write $c = \mathsf{Enc}_{pk}(m)$, where $c = (c_1, \ldots, c_e)$ with $c_i = \mathsf{Enc}_{pk_1}(\mathsf{Enc}_{pk_2}(\cdots \mathsf{Enc}_{pk_\ell}(m_i) \cdots))$. Similarly, $m = \mathsf{Dec}_{sk}(c)$ is defined by $m_i = \mathsf{Dec}_{sk_\ell}(\mathsf{Dec}_{sk_{\ell-1}}(\cdots \mathsf{Dec}_{sk_1}(c_i) \cdots))$. We stress that when we use Enc as a probabilistic algorithm with a list of messages or public keys, we assume that the random values used in each encryption are chosen randomly and independently. We use the notation $a \| b$ for the concatenation of two bitstrings. We define the function $\mathsf{Split}_t(a)$ to divide a bitstring a, whose length is a multiple of t, into t chunks of equals lengths and turn it into a list, i.e., $(a_1, \ldots, a_t) = \mathsf{Split}_t(a_1 \| \ldots \| a_t)$ when $|a_i|$s are equal.

4 Definitions and Conventions

We consider a mix-net employing k mix-servers $\mathcal{M}_1, \ldots, \mathcal{M}_k$ that provide anonymity for a group of N senders $\mathcal{P}_1, \ldots, \mathcal{P}_N$. Throughout, $\overline{\mathcal{M}}$ and $\overline{\mathcal{P}}$ denote the sets of all mix-servers and senders respectively. We let $J_{\mathcal{M}} \subset [1, k]$ and $I_{\mathcal{P}} \subset [1, N]$ denote the index sets of corrupted mix-servers and senders respectively. We let $J^* \subset J_{\mathcal{M}}$ denote the index set of mix-servers identified as corrupted so far. This set may grow throughout an execution.

We present and analyze the main components of our mix-net in the universal composability framework [1], with *non-blocking* adversaries, i.e., adversaries that do not block the delivery of messages indefinitely. We use superscripts to distinguish different functionalities and protocols, for example $\mathcal{F}^{\mathrm{bb}}$ for a bulletin board and π^{c} for Chaum's mix-net. The ideal adversary (simulator) of the ideal model is denoted by $\mathcal{S}$. When there is no ambiguity, we use the same notation for dummy parties and real parties.

We use a number of conventions to simplify the exposition. Whenever we say a party "hands" a message to a functionality, we mean that the party sends the message to the corresponding dummy party who will then forward it to the functionality. All our functionalities capture distributed protocols where messages sent to more than one party can be delayed arbitrarily by the adversary, and all such messages are also given to the adversary. Thus, when we say that a functionality hands a message to more than one party, we mean that the message is passed to the adversary, who then schedules the delivery of the message to the parties. When a party "inputs" a message to a subprotocol, we mean that he executes the algorithm of the corresponding party with the same message. A party or protocol is said to "wait for" an input of a given form if any other input is immediately returned to the sender. Similarly, a party can wait for a message to appear on the bulletin board. In practice this would be implemented using a time-out, after which some default value is taken to be the message. Some of our functionalities give an output before receiving any input, which makes no sense in an event-driven model like the universal composability framework where execution starts by activating the environment. This is merely a useful convention, since we can easily fix this problem by allowing parties to request the given data.

In all of our protocols, security holds only as long as the adversary corrupts less than λ mix-servers, where $1 \leq \lambda \leq k$ is a parameter of the protocol. All our functionalities and protocols may fail to give an output if more than $\min(\lambda - 1, k - \lambda)$ mix-servers are corrupted. To capture the case $\lambda \geq k/2$ with minimal notational overhead, we simply assume that even a non-blocking simulator can block messages indefinitely in this case.

We use the subroutine $\mathsf{Agree}(\mathit{Tag})$, parameterized by a label *Tag*, to simplify the description of some of our ideal functionalities. The subroutine waits until each mix-server $\mathcal{M}_j$ has submitted a pair (Tag, m_j) for some message m_j. If at least λ mix-servers submitted identical m_j, then the subroutine returns this value and otherwise it halts the complete ideal functionality, e.g., the functionality could hand $\perp$ to all parties and ignore inputs from then on. The message m_j can be an empty string in which case the subroutine is only used to capture the robustness property of the functionality. In Appendix A we give a formal definition of the subroutine $\mathsf{Agree}(\mathit{Tag})$. We use the same convention for protocols, i.e., if an ideal functionality used by the protocol aborts, then the protocol aborts as well. These conventions allow us to capture the robustness of a protocol by requiring a non-blocking simulator for a non-blocking adversary.

4.1 Useful Functionalities

Our results are given in a hybrid model with distributed key generation functionalities of two types, a bulletin board functionality, and a proof of correct decryption functionality. In Appendix B, formal descriptions of these functionalities are presented. The first key generation functionality, $\mathcal{F}_j^{\mathrm{kg}}$, generates a public key pk_j such that only the jth mix-server knows the corresponding secret key sk_j. The second functionality, $\mathcal{F}^{\mathrm{dkg}}$, differs only in that no mix-server learns the secret key sk corresponding to the generated public key pk. In both functionalities, any subset of λ mix-servers can recover the secret key. The bulletin board functionality, denoted by $\mathcal{F}^{\mathrm{bb}}$, is used by parties to announce their messages. That is, a message can be posted by any party and read by any other one. To simplify the exposition, we simply say that a message is "published" when it appears on the public bulletin board. The published message can not be deleted or modified once posted. The proof of correct decryption functionality, $\mathcal{F}_j^{\mathrm{pd}}$, is used to prove that the jth mix-server has correctly decrypted a known ciphertext into a known plaintext. A subset of λ mix-servers must agree on the pair of plaintext and ciphertext. Assuming that the underlying cryptosystem allows the jth mix-server to recover the randomness used during encryption from the ciphertext itself, the realization of the functionality becomes trivial. In other words, the proof of correct decryption simply consists of revealing the randomness used for encryption. Our main result, Theorem 1, holds if this solution is employed, but only in a standalone model (see the full version of this paper for details). In any case, proofs of correct decryption are only used to trace ciphertexts to identify corrupted senders or mix-servers.

4.2 Mix-Nets

We use ideal mix-net functionalities similar to that in [24], but in a slightly simplified form in that we assume that each sender submits exactly one input. Functionality 1 presents a natural mix-net. Our results are easy to generalize to the case where senders can submit more than one input (this holds also for Functionality 2 and Functionality 3).

The protocol we construct does not quite implement the natural mix-net. Thus, we present a relaxed mix-net (Functionality 2) which we are able to securely realize and then argue that it still provides sufficient guarantees. The relaxed functionality first hands the adversary (simulator) a public key. Then it waits for inputs from all the senders, encrypts the messages of the honest senders, and then hands the resulting ciphertexts in sorted order to the adversary. The adversary is then asked to provide his own inputs in encrypted form on behalf of corrupted senders. The final output is the sorted decryption of the union of the ciphertexts computed by the functionality and those provided by the adversary (after duplicates are removed). For technical reasons the functionality uses several public keys and encrypts the messages under all keys.

This functionality provides *unconditional privacy* for honest senders. The relaxation lies in the ability of an *unbounded* adversary to *adaptively choose* the messages of the corrupted senders based on the *set of inputs* of the honest senders, but a CCA2-secure cryptosystem prevents this for efficient adversaries.

We define a mix-net with abort (Functionality 3) that either gives a proper output or aborts after identifying a mix-server as culprit (with no information about the submitted messages at all). A relaxed mix-net can be constructed using such a mix-net with abort. The mix-net with abort waits for inputs from all the senders and then outputs these messages in encrypted form (as in the relaxed mix-net). Then it allows the mix-servers to agree on a list of known corrupted mix-servers. Finally, the adversary decides if the mix-net should abort or not. In the former case, the adversary must provide the index of a *previously unknown* corrupted mix-server, and this is forwarded to all mix-servers. In the latter case, the mix-net outputs the result like in the relaxed mix-net.

In the full version of this paper we describe a protocol using Functionality 3 that securely realizes Functionality 2. The idea is to use λ instances of Functionality 3. Each sender submits a copy of his input to all functionalities. The mix-servers then run them sequentially until one produces an output without aborting. To ensure that this scheme eventually gives an output, the mix-servers jointly keep track of the identified corrupted mix-servers.

Functionality 1 (Natural Mix-Net). The *natural mix-net* functionality $\mathcal{F}^{\mathrm{mn}}$ executing with dummy senders $\overline{\mathcal{P}}$, dummy mix-servers $\overline{\mathcal{M}}$ and ideal adversary $\mathcal{S}$ proceeds as follows.

1. Let $I = [1, N]$. While $I \neq \emptyset$:
   a) Wait for a message $(\textit{Message}, m_i)$ with $m_i \in \{0,1\}^n$ from some dummy sender $\mathcal{P}_i$ with $i \in I$.
   b) Set $I \leftarrow I \setminus \{i\}$ and hand $(\textit{MessageReceived}, i)$ to $\mathcal{S}$.
2. Hand $\big(\textit{Mixed}, \mathsf{Sort}(m_1, \ldots, m_N)\big)$ to $\mathcal{S}$ and $\overline{\mathcal{M}}$.

Functionality 2 (Relaxed Mix-Net). The *relaxed mix-net* functionality $\mathcal{F}^{\mathrm{mn}}$ executing with dummy senders $\overline{\mathcal{P}}$, dummy mix-servers $\overline{\mathcal{M}}$ and ideal adversary $\mathcal{S}$ proceeds as follows.

1. Hand $\big(\textit{PublicKeys}, (pk_\ell)_{\ell=1}^{\lambda}\big)$ to $\mathcal{S}$, where $(pk_\ell, sk_\ell) = \mathsf{Gen}(1^n)$.
2. Let $I = [1, N]$. While $I \neq \emptyset$:
   a) Wait for a message $(\textit{Message}, m_i)$ with $m_i \in \{0,1\}^n$ from some dummy sender $\mathcal{P}_i$ with $i \in I$.
   b) Set $I \leftarrow I \setminus \{i\}$ and hand $(\textit{MessageReceived}, i)$ to $\mathcal{S}$.
3. Let $L_\ell = \mathsf{Sort}\big(\big(\mathsf{Enc}_{pk_\ell}(m_i)\big)_{i \in [1,N] \setminus I_{\mathcal{P}}}\big)$. Hand $\big(\textit{HonestCiphertexts}, (L_\ell)_{\ell=1}^{\lambda}\big)$ to $\mathcal{S}$ and wait to get back $(\textit{CorruptCiphertexts}, L', \ell^*)$, where $|L'| \leq |I_{\mathcal{P}}|$ and $1 \leq \ell^* \leq \lambda$.
4. Hand $(\textit{SecretKey}, sk_{\ell^*})$ to $\mathcal{S}$ and $\big(\textit{Mixed}, \mathsf{Sort}\big(\mathsf{Dec}_{sk_{\ell^*}}(\mathsf{Unique}(L_{\ell^*} \circ L'))\big)\big)$ to $\overline{\mathcal{M}}$.

Functionality 3 (Mix-Net With Abort). The *mix-net with abort* functionality $\mathcal{F}^{\mathrm{mna}}$ executing with dummy senders $\overline{\mathcal{P}}$, dummy mix-servers $\overline{\mathcal{M}}$, and ideal adversary $\mathcal{S}$ proceeds as follows.

1. Generate $(pk, sk) = \mathsf{Gen}(1^n)$ and hand $(\textit{PublicKey}, pk)$ to $\mathcal{S}$.
2. Let $I = [1, N]$. Then while $I \neq \emptyset$:
   a) Wait for a message $(\textit{Message}, m_i)$ with $m_i \in \{0,1\}^n$ from some dummy sender $\mathcal{P}_i$ with $i \in I$.
   b) Set $I \leftarrow I \setminus \{i\}$, let $v_i = \mathsf{Enc}_{pk}(m_i)$, and hand $(\textit{MessageReceived}, i)$ to $\mathcal{S}$.
3. Wait for a common input $J^* \subset J_{\mathcal{M}}$ from dummy mix-servers, i.e., $J^* \leftarrow \mathsf{Agree}(\textit{Culprits})$.
4. Let $L = \mathsf{Sort}\big((v_i)_{i \in [1,N] \setminus I_{\mathcal{P}}}\big)$ and wait for a message *EncryptPlaintexts* from $\mathcal{S}$. Then hand $(\textit{HonestCiphertexts}, L)$ to $\mathcal{S}$ and wait to receive $(\textit{CorruptCiphertexts}, L')$ where $|L'| \leq |I_{\mathcal{P}}|$, or $(\textit{Culprit}, d)$ where $d \in J_{\mathcal{M}} \setminus J^*$. In the latter case, hand $(\textit{Culprit}, d)$ to $\overline{\mathcal{M}}$ and halt.
5. Hand $(\textit{SecretKey}, sk)$ to $\mathcal{S}$ and $\big(\textit{Mixed}, \mathsf{Sort}\big(\mathsf{Dec}_{sk}(\mathsf{Unique}(L \circ L'))\big)\big)$ to $\overline{\mathcal{M}}$.

5 Chaum's Mix-Net

Consider Chaum's original mix-net [2] with λ mix-servers in the chain. Each mix-server $\mathcal{M}_j$ generates a key pair (pk_j, sk_j) and a sender wraps her message m_i in λ layers of encryptions and submits a ciphertext $c_i = \mathsf{Enc}_{pk_1}\big(\mathsf{Enc}_{pk_2}\big(\cdots \mathsf{Enc}_{pk_\lambda}(m_i)\cdots\big)\big)$. Then the mix-servers form an initial list $L_0 = (c_i)_{i\in[1,N]}$, and sequentially peel off layers of encryptions after removing the duplicates. That is, for $j = 1,\ldots,\lambda$, the jth mix-server computes $L_j = \mathsf{Sort}\big(\mathsf{Dec}_{sk_j}(\mathsf{Unique}(L_{j-1}))\big)$. Thus, $\mathsf{Unique}(L_\lambda)$ is the sorted list of plaintexts without duplicates. This mix-net is neither secure against active adversaries nor robust, but it nevertheless forms the basis of our constructions. We formalize this in Protocol 1 below and later extend it in two different ways in Protocol 2 and Protocol 3. We assume that the main protocol (Protocol 4) keeps track of the set J^* of indices of identified corrupted mix-server so far, see Step 3 of Protocol 1 below.

Protocol 1 (Chaum's Mix-Net, π^{c}).
Mix-servers. The jth mix-server $\mathcal{M}_j$ proceeds as follows when executing with functionalities $\mathcal{F}^{\mathrm{bb}}$, and $\mathcal{F}^{\mathrm{kg}}_1,\ldots,\mathcal{F}^{\mathrm{kg}}_\lambda$.

1. Wait for $(\mathit{PublicKey}, pk_\ell)$ from $\mathcal{F}^{\mathrm{kg}}_\ell$ for $\ell = 1,\ldots,\lambda$. Let $pk = (pk_1,\ldots,pk_\lambda)$ and output $(\mathit{PublicKey}, pk)$. Wait for $(\mathit{SecretKey}, sk_j)$ from $\mathcal{F}^{\mathrm{kg}}_j$ if $j \in [1,\lambda]$.
2. Wait for an input $(\mathit{Culprits}, J^*)$. For $\ell = 1,\ldots,\lambda$: if $\ell \in J^*$, then hand *Recover* to $\mathcal{F}^{\mathrm{kg}}_\ell$ and wait for a response $(\mathit{SecretKey}, sk_\ell)$.
3. Wait for an input $(\mathit{Ciphertexts}, L_0)$. For $\ell = 1,\ldots,\lambda$ do the following and output $\big(\mathit{Mixed}, \mathsf{Unique}(L_\lambda)\big)$:
   (a) If $\ell \in J^*$ or $\ell = j$, then set $L_\ell = \mathsf{Sort}\big(\mathsf{Dec}_{sk_\ell}(\mathsf{Unique}(L_{\ell-1}))\big)$, and publish $(\mathit{Decryption}, L_\ell)$.
   (b) Otherwise, wait until $\mathcal{M}_\ell$ publishes $(\mathit{Decryption}, L_\ell)$ (or we published L_ℓ, since sk_ℓ was recovered), where $|L_\ell| = |\mathsf{Unique}(L_{\ell-1})|$.

Protocol 2 and Protocol 3 formalize the two nested mix-nets used in our main protocol. Recall from Section 2 that the first protocol is an *optimistic* execution of Chaum's mix-net. The privacy of this mix-net is only required to *temporarily randomize* the input to the second mix-net. This is needed to argue that it is hard to replace all ciphertexts submitted by a non-empty proper subset of the honest senders without being identified as a cheater. When Protocol 3 has completed, the optimistic execution is verified explicitly by simply recovering the secret keys of all mix-servers.

Protocol 2 (Chaum's Mix-Net with Explicit Verification, π^{cev}).
Mix-servers. The jth mix-server $\mathcal{M}_j$ when executing with functionalities $\mathcal{F}^{\mathrm{bb}}$, and $\mathcal{F}^{\mathrm{kg}}_1,\ldots,\mathcal{F}^{\mathrm{kg}}_\lambda$, first runs Chaum's mix-net (Protocol 1) and then proceeds as follows.

4. Wait for an input *Verify*. Then for $\ell = 1,\ldots,\lambda$, where $\ell \notin J^*$:
   (a) If $\ell = j$, then publish $(\mathit{SecretKey}, sk_j)$.
   (b) If $\ell \neq j$ and $\ell \notin J^*$, then wait until $\mathcal{M}_\ell$ publishes $(\mathit{SecretKey}, sk_\ell)$, and halt with output $(\mathit{Culprit}, \ell)$ if sk_ℓ does not correspond to pk_ℓ or if $L_\ell \neq \mathsf{Sort}\big(\mathsf{Dec}_{sk_\ell}(\mathsf{Unique}(L_{\ell-1}))\big)$.
5. Halt with output $(\mathit{SecretKey}, sk)$, where $sk = (sk_1,\ldots,sk_\lambda)$.

In our second variant of Chaum's mix-net (Protocol 3), the mix-servers proceed optimistically, but in contrast to Protocol 2 they do not later verify the complete execution explicitly. Instead, they *trace* a subset of ciphertexts backwards and forwards through the mix-net and reveal how they are decrypted in the process. In the main protocol a small subset of *dummy ciphertexts* (submitted by the mix-servers) are always traced forward to show that these were processed correctly. As explained in Section 2, the idea is that starting from the randomly permuted output of Protocol 2, the adversary must avoid modifying the traced ciphertexts to avoid detection. In other words, to cheat without detection, a corrupted mix-server can not simply replace all ciphertexts. However, tracing starts by tracing any ciphertexts that do not have exactly t copies backwards to distinguish the case where a corrupted sender submits a malformed set of ciphertexts from the case where a corrupted mix-server processes his input incorrectly. Only then are the dummies, and possibly additional ciphertexts, traced forwards through the mix-net.

Protocol 3 (Chaum's Mix-Net with Partial Tracing, π^{cpt}).
Mix-servers. The jth mix-server $\mathcal{M}_j$, when executing with functionalities $\mathcal{F}^{\mathrm{bb}}$, $\mathcal{F}^{\mathrm{kg}}_1, \ldots, \mathcal{F}^{\mathrm{kg}}_\lambda$, and $\mathcal{F}^{\mathrm{pd}}_1, \ldots, \mathcal{F}^{\mathrm{pd}}_\lambda$, runs Chaum's mix-net (Protocol 1), hands $(\mathit{SecretKey}, sk_j)$ to $\mathcal{F}^{\mathrm{pd}}_j$ if $j \in [1, \lambda]$, and then proceeds as follows.

4. *Backward Tracing.* Wait for an input $(\mathit{TraceB}, B_\lambda)$, where B_λ is the list of ciphertexts to be traced backwards. For $\ell = \lambda, \ldots, 1$ do the following and then output (Traced, B_0):
   (a) Expand B_ℓ to a list B'_ℓ by adding the removed duplicates, i.e., the expanded list B'_ℓ includes all copies in L_ℓ of every ciphertext occurring in B_ℓ.
   (b) If $\ell \in J^*$ or $\ell = j$, then identify $B_{\ell-1} \subset L_{\ell-1}$ such that $B'_\ell = \mathsf{Dec}_{sk_\ell}\big(B_{\ell-1}\big)$ and publish $(\mathit{TracedB}, B_{\ell-1})$. Otherwise, wait until $\mathcal{M}_\ell$ publishes $(\mathit{TracedB}, B_{\ell-1})$ with $B_{\ell-1} \subset L_{\ell-1}$.
   (c) If $\ell \notin J^*$, then hand $(\mathit{Verify}, B'_\ell, B_{\ell-1})$ to $\mathcal{F}^{\mathrm{pd}}_\ell$ and halt with $(\mathit{Culprit}, \ell)$ if it returns *False*.
5. *Forward Tracing.* Wait for an input (TraceF, F_0), where F_0 is the ciphertexts to be traced forward. For $\ell = 1, \ldots, \lambda$ do the following and then halt with output $(\mathit{Traced}, F_\lambda)$:
   (a) Let $F'_{\ell-1} = \mathsf{Unique}(F_{\ell-1})$.
   (b) If $\ell \in J^*$ or $\ell = j$, then let $F_\ell = \mathsf{Dec}_{sk_\ell}(F'_{\ell-1})$ and publish $(\mathit{TracedF}, F_\ell)$. Otherwise, wait until $\mathcal{M}_\ell$ publishes $(\mathit{TracedF}, F_\ell)$ with $F_\ell \subset L_\ell$.
   (c) If $\ell \notin J^*$, then hand $(\mathit{Verify}, F_\ell, F'_{\ell-1})$ to $\mathcal{F}^{\mathrm{pd}}_\ell$ and halt with $(\mathit{Culprit}, \ell)$ if it returns *False*.

Forward tracing of the dummy ciphertext list F_0, a subset of the input list L_0 submitted by the mix-servers, is done in the natural way. For $\ell = 1, \ldots, \lambda$, the ℓth mix-server computes $F_\ell = \mathsf{Dec}_{sk_\ell}\big(\mathsf{Unique}(F_{\ell-1})\big)$ and proves that he did so correctly. The other mix-servers verify the proof and that $F_\ell \subset L_\ell$.

Backward tracing of a list B_λ, a subset of the output list $\mathsf{Unique}(L_\lambda)$, is more complicated in that we must invert the process of duplicate removal. For $\ell = \lambda, \ldots, 1$, all mix-servers first expand B_ℓ into a list B'_ℓ by including all copies in L_ℓ of each ciphertext in B_ℓ, and then the ℓth mix-server computes $B_{\ell-1} \subset L_{\ell-1}$ such that $B'_\ell =$

$\mathsf{Dec}_{sk_\ell}(B_{\ell-1})$ and proves that this relation holds. Thus, the expansion is the inversion of how Unique removed duplicates of traced ciphertexts during processing.

The correctness of the decryption for the jth mix-server is verified using the proof of correct decryption functionality $\mathcal{F}_j^{\mathrm{pd}}$. Notice that for the dummies, it suffices that each mix-server simply reveals the randomness used to encrypt his own dummy inputs. However, for senders' inputs this may not be possible for a general cryptosystem since the randomness is chosen by the corresponding sender and may not be known to the decrypting mix-server. Nevertheless, one possible incarnation of our protocol uses a cryptosystem that allows recovering the randomness used during encryption from the ciphertext itself during decryption. In this case, the proof of correct decryption used during tracing simply consists of revealing the randomness.

6 Constructing a Mix-Net With Abort

We are now ready to present the details of our mix-net with abort in Protocol 4. We use two nested instances of Chaum's mix-net: one with explicit verification (Protocol 2), and one with partial tracing (Protocol 3). The lists of public keys of these mix-nets are denoted by pk^{cev} and pk^{cpt}, each of which contains λ keys. Each sender encrypts her message m_i once using the additional joint "final" public key pk^{f} to form a ciphertext v_i. This layer of encryption hides the inputs of the honest senders if the execution aborts. The ciphertext v_i is then encrypted independently t times with the additional joint "replication" public key pk^{r}. Recall that this prevents the last mix-server in Chaum's mix-net with partial tracing (Protocol 3) from identifying all ciphertexts submitted by the same sender. The resulting ciphertexts are then encrypted using the lists pk^{cpt} and pk^{cev} of public keys of the two instances of Chaum's mix-net. Finally, the t encryptions are concatenated to form one plaintext chunk and then encrypted using the "outer" public key pk^{o}, which prevents a dishonest sender (with the collusion of some dishonest mix-servers) from partially copying an honest sender's submission to break his privacy. In addition to the ciphertexts submitted by senders, each mix-server submits a dummy encryption of the zero message computed like a sender's ciphertext. These ciphertexts prevent a corrupt mix-server from replacing all ciphertexts instead of guessing the positions of all ciphertexts submitted by a subset of the senders.

To process the ciphertexts, the mix-servers first remove the "outer" layer of encryption by jointly recovering the corresponding secret key sk^{o}. Then they execute the two instances of Chaum's mix-net in sequence. We stress that the t ciphertexts of each sender are processed independently at this stage. Then the secret keys in sk^{cev} (corresponding to pk^{cev}) are recovered and the mix-servers verify the execution of the first mix-net explicitly. The "replication" secret key sk^{r} corresponding to pk^{r} is then recovered and all ciphertexts are decrypted. Finally, the processing in the second mix-net is verified for: (1) all ciphertexts of which there are not exactly t copies (backward tracing), and (2) all dummy ciphertexts submitted by mix-servers and all ciphertexts intersecting with the ciphertexts traced backwards (forward tracing). If there is any inconsistency, the corrupted mix-server is identified and the execution aborts. If there is no inconsistency, then the "final" secret key sk^{f} corresponding to pk^{f} is recovered and the innermost layer of encryption is removed to reveal the plaintexts.

Theorem 1 captures security of Protocol 4. If we use a cryptosystem that allows recovering the randomness used for encryption, then our result still holds, but only in the standalone model where the simulator is allowed to rewind. The full version details this variation of the scheme.

Protocol 4 (Mix-Net with Abort π^{mna}). This protocol is executed with a bulletin board $\mathcal{F}^{\mathrm{bb}}$, a mix-net with explicit verification π^{cev}, a mix-net with partial tracing π^{cpt}, and distributed key generation functionalities $\mathcal{F}_{\mathrm{o}}^{\mathrm{dkg}}$, $\mathcal{F}_{\mathrm{r}}^{\mathrm{dkg}}$ and $\mathcal{F}_{\mathrm{f}}^{\mathrm{dkg}}$.

Senders. The ith sender $\mathcal{P}_i$ proceeds as follows on input $m_i \in \{0,1\}^n$.

1. Wait until λ of the mix-servers have published identical list $(\mathit{PublicKeys}, pk^{\mathrm{o}}, pk^{\mathrm{cev}}, pk^{\mathrm{cpt}}, pk^{\mathrm{r}}, pk^{\mathrm{f}})$. If no such list exists, then abort.
2. Let $v_i = \mathsf{Enc}_{pk^{\mathrm{f}}}(m_i)$.
3. Let $u_{i,s} = \mathsf{Enc}_{pk^{\mathrm{cev}}}(\mathsf{Enc}_{pk^{\mathrm{cpt}}}(\mathsf{Enc}_{pk^{\mathrm{r}}}(v_i)))$, for $s = 1, \ldots, t$.
4. Let $\overline{u}_i = \mathsf{Enc}_{pk^{\mathrm{o}}}(u_{i,1} \| \cdots \| u_{i,t})$ and publish $(\mathit{Ciphertext}, \overline{u}_i)$.

Mix-servers. The jth mix-server $\mathcal{M}_j$ proceeds as follows on input J_j^*.

1. *Public Keys.* Wait for public keys: $(\mathit{PublicKey}, pk^{\mathrm{o}})$ from $\mathcal{F}_{\mathrm{o}}^{\mathrm{dkg}}$, $(\mathit{PublicKey}, pk^{\mathrm{cev}})$ from π^{cev}, $(\mathit{PublicKey}, pk^{\mathrm{cpt}})$ from π^{cpt}, $(\mathit{PublicKey}, pk^{\mathrm{r}})$ from $\mathcal{F}_{\mathrm{r}}^{\mathrm{dkg}}$, and $(\mathit{PublicKey}, pk^{\mathrm{f}})$ from $\mathcal{F}_{\mathrm{f}}^{\mathrm{dkg}}$. Then publish $(\mathit{PublicKeys}, pk^{\mathrm{o}}, pk^{\mathrm{cev}}, pk^{\mathrm{cpt}}, pk^{\mathrm{r}}, pk^{\mathrm{f}})$. Wait until λ of the mix-servers have published the same list, or abort if no such list can be found.
2. *Input Ciphertexts.* Wait until every $\mathcal{P}_i$ has published her encrypted message $(\mathit{Ciphertext}, \overline{u}_i)$. Let $\overline{u}_{N+j}$ be an encryption of zero as computed by a sender and publish $(\mathit{Ciphertext}, \overline{u}_{N+j})$. Wait until every $\mathcal{M}_\ell$ has published $(\mathit{Ciphertext}, \overline{u}_{N+\ell})$ and let $L^{\mathrm{in}} = \mathsf{Unique}(\overline{u}_1, \ldots, \overline{u}_{N+k})$.
3. *Culprits Agreement.* Publish $(\mathit{Culprits}, J_j^*)$ and wait until λ of the mix-servers have published identical $(\mathit{Culprits}, J^*)$, or abort if no such set J^* can be found. Input $(\mathit{Culprits}, J^*)$ to π^{cev} and π^{cpt}.
4. *Decrypt and Split.* Hand *Recover* to $\mathcal{F}_{\mathrm{o}}^{\mathrm{dkg}}$ and wait for a response $(\mathit{SecretKey}, sk^{\mathrm{o}})$. Let $L^{\mathrm{o}} = \bigcirc_{\overline{u} \in L^{\mathrm{in}}} \mathsf{Split}_t\big(\mathsf{Dec}_{sk^{\mathrm{o}}}(\overline{u})\big)$.
5. *Chaum's Mix-Net.* Input $(\mathit{Ciphertexts}, L^{\mathrm{o}})$ to π^{cev} and wait for an output $(\mathit{Mixed}, L^{\mathrm{cev}})$.
6. *Chaum's Mix-Net.* Input $(\mathit{Ciphertexts}, L^{\mathrm{cev}})$ to π^{cpt}, and wait for an output $(\mathit{Mixed}, L^{\mathrm{cpt}})$.

This protocol is completed on the next page.

Theorem 1. *Let $\mathcal{CS}$ be a CCA2 secure cryptosystem. Then Protocol 4 securely realizes Functionality 3 with respect to static active adversaries that corrupt less than λ of the mix-servers and any number of senders, provided that t is chosen such that $H^{-(t-1)}$ is negligible, where $H > 1$ is the number of honest parties.*

Due to our conventions in Section 4 and the definition of Functionality 3, the theorem also captures the robustness of the protocol, i.e., it gives an output provided that at most $\min(\lambda - 1, k - \lambda)$ parties are corrupted.

Protocol 4 (Continued, including verifications.).

7. *Verifications.*
   (a) *Explicit Verification.* Input *Verify* to π^{cev}. If it outputs $(\textit{Culprit}, d)$, then halt with this output, and otherwise let $(\textit{SecretKey}, sk^{\mathrm{cev}})$ be the output.
   (b) *Replication Check.* Hand *Recover* to $\mathcal{F}_{\mathrm{r}}^{\mathrm{dkg}}$ and wait for a response $(\textit{SecretKey}, sk^{\mathrm{r}})$. Compute $L^{\mathrm{r}} = \mathsf{Dec}_{sk^{\mathrm{r}}}(L^{\mathrm{cpt}})$ and let B be the ciphertexts in L^{cpt} that do not have exactly t copies after decryption with sk^{r}.
   (c) *Backwards Tracing.* Input $(\textit{TraceB}, B)$ to π^{cpt}. If it outputs $(\textit{Culprit}, d)$, then halt with this output, and otherwise let $(\textit{TracedB}, B')$ be the output. Let L' be the list of all $\overline{u} \in L^{\mathrm{in}}$ such that π^{cev} on input $\big(\textit{Ciphertexts}, \mathsf{Split}_t(\mathsf{Dec}_{sk^{\mathrm{o}}}(\overline{u}))\big)$ would output $(\textit{Mixed}, B'')$ with $B' \cap B'' \neq \emptyset$.
   (d) *Forward Tracing.* Let F be the list such that π^{cev} on input $(\textit{Ciphertexts}, L' \circ L'')$, where $L'' = \bigcirc_{\ell\in[1,k]} \mathsf{Split}_t\big(\mathsf{Dec}_{sk^{\mathrm{o}}}(\overline{u}_{N+\ell})\big)$, would give an output $(\textit{Mixed}, F)$. Input $(\textit{TraceF}, F)$ to π^{cpt}. If it outputs $(\textit{Culprit}, d)$, then halt with this output. Otherwise, let $(\textit{TracedF}, F')$ be the output.
8. *Final Decryption.* Hand *Recover* to $\mathcal{F}_{\mathrm{f}}^{\mathrm{dkg}}$ and wait for a response $(\textit{SecretKey}, sk^{\mathrm{f}})$. Let $L^{\mathrm{r}\prime} = \mathsf{Unique}\big(\mathsf{Dec}_{sk^{\mathrm{r}}}(L^{\mathrm{cpt}} \setminus F')\big)$ and halt with output $\big(\textit{Mixed}, \mathsf{Sort}(\mathsf{Dec}_{sk^{\mathrm{f}}}(L^{\mathrm{r}\prime}))\big)$.

7 Conclusion

We construct a provably secure mix-net that unlike many other mix-nets in the literature do not require any homomorphic properties from the cryptosystem. This is a clear advantage for those concerned that quantum computers can be constructed in the future. In contrast to the only previous proposed mix-net based on any cryptosystem [12], our construction enjoys not only provable security but also full privacy and correctness. Our mix-net is fast there are many senders and plaintexts are large.

References

1. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
2. D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.
3. Y. Desmedt and K. Kurosawa. How to break a practical mix and design a new one. In B. Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2000.
4. J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer, 2001.
5. J. Furukawa and K. Sako. An efficient publicly verifiable mix-net for long inputs. *IEICE Transactions*, 90-A(1):113–127, 2007.
6. E. Gabber, P. B. Gibbons, Y. Matias, and A. J. Mayer. How to make personalized web browising simple, secure, and anonymous. In R. Hirschfeld, editor, *Financial Cryptography*, volume 1318 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1997.

7. P. Golle, S. Zhong, D. Boneh, M. Jakobsson, and A. Juels. Optimistic mixing for exit-polls. In Y. Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2002.
8. M. Jakobsson. A practical mix. In *EUROCRYPT*, pages 448–461, 1998.
9. M. Jakobsson. Flash mixing. In *PODC*, pages 83–89, 1999.
10. M. Jakobsson and A. Juels. Mix and match: Secure function evaluation via ciphertexts. In Okamoto [18], pages 162–177.
11. M. Jakobsson and A. Juels. An optimally robust hybrid mix network. In *PODC*, pages 284–292, New York, NY, USA, 2001. ACM Press.
12. M. Jakobsson, A. Juels, and R. L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In D. Boneh, editor, *USENIX Security Symposium*, pages 339–353. USENIX, 2002.
13. M. Jakobsson and D. M'Raïhi. Mix-based electronic payments. In S. E. Tavares and H. Meijer, editors, *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 1998.
14. S. Khazaei and D. Wikström. Randomized partial checking revisited. Cryptology ePrint Archive, Report 2012/063, 2012. `http://eprint.iacr.org/2012/063`.
15. M. Mitomo and K. Kurosawa. Attack for flash mix. In Okamoto [18], pages 192–204.
16. C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *CCS '01: Proc. of the 8th ACM conference on Computer and Communications Security*, pages 116–125, New York, NY, USA, 2001. ACM.
17. M. Ohkubo and M. Abe. A length-invariant hybrid mix. In Okamoto [18], pages 178–191.
18. T. Okamoto, editor. *Advances in Cryptology — ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3–7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*. Springer, 2000.
19. C. Park, K. Itoh, and K. Kurosawa. Efficient anonymous channel and all/nothing election scheme. In *EUROCRYPT*, pages 248–259, 1993.
20. B. Pfitzmann. Breaking efficient anonymous channel. In *EUROCRYPT*, pages 332–340, 1994.
21. B. Pfitzmann and A. Pfitzmann. How to break the direct rsa-implementation of mixes. In *EUROCRYPT*, pages 373–381, 1989.
22. K. Sako and J. Kilian. Receipt-free mix-type voting scheme — a practical solution to the implementation of a voting booth. In *EUROCRYPT*, pages 393–403, 1995.
23. D. Wikström. Five practical attacks for "optimistic mixing for exit-polls". In M. Matsui and R. J. Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2004.
24. D. Wikström. A universally composable mix-net. In M. Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 317–335. Springer, 2004.

A Agreement Subroutine

Subroutine 1 (Agree(***Tag***)).

```
1. Set $J \leftarrow \{1, \ldots, k\}$.
2. While $J \neq \emptyset$:
   a) Wait for a message $(Tag, m_j)$ from the dummy mix-server $\mathcal{M}_j$ with $j \in J$.
   b) Set $J \leftarrow J \setminus \{j\}$ and hand $(TagReceived, j, m_j)$ to $\mathcal{S}$.
3. Return the value in $(m_j)_{j \in [1,k]}$ that has been submitted by $\lambda$ of the mix-servers.
   If no such value exists, hand $\perp$ to $\overline{\mathcal{M}}$ and halt the main functionality.
```

B Functionalities Implemented by General MPC

Functionality 4 (Key Generation with VSS). The key generation with VSS functionality $\mathcal{F}_j^{\mathrm{kg}}$ executing with dummy mix-servers $\overline{\mathcal{M}}$ and ideal adversary $\mathcal{S}$ proceeds as follows.

1. Generate $(pk, sk) = \mathsf{Gen}(1^n)$, hand $(\textit{PublicKey}, pk)$ to $\mathcal{S}$ and $\overline{\mathcal{M}}$, and $(\textit{SecretKey}, sk)$ to $\mathcal{M}_j$, and wait until dummy mix-servers agree to recover, i.e., run $\mathsf{Agree}(\textit{Recover})$.
2. Hand $(\textit{SecretKey}, sk)$ to $\mathcal{S}$ and $\overline{\mathcal{M}}$.

Functionality 5 (Distributed Key Generation with VSS). The distributed key generation with VSS functionality $\mathcal{F}^{\mathrm{dkg}}$ executing with dummy mix-servers $\overline{\mathcal{M}}$ and ideal adversary $\mathcal{S}$ proceeds as follows.

1. Generate $(pk, sk) = \mathsf{Gen}(1^n)$.
2. Hand $(\textit{PublicKey}, pk)$ to $\mathcal{S}$ and $\overline{\mathcal{M}}$, and wait until dummy mix-servers agree to recover, i.e., run $\mathsf{Agree}(\textit{Recover})$.
3. Hand $(\textit{SecretKey}, sk)$ to $\mathcal{S}$ and $\overline{\mathcal{M}}$.

Functionality 6 (Bulletin board). Executing with dummy senders $\overline{\mathcal{P}}$, dummy mix-servers $\overline{\mathcal{M}}$ and ideal adversary $\mathcal{S}$, the bulletin board functionality $\mathcal{F}^{\mathrm{bb}}$ keeps a private and a public[a] database and proceeds as follows.

1. Upon receiving a message $(\textit{Tag}, m)$ from a party $P \in \overline{\mathcal{P}} \cup \overline{\mathcal{M}}$, hand $(P, \textit{Tag}, m)$ to $\mathcal{S}$ and write $(P, \textit{Tag}, m)$ on the private database. Ignore any further message $(\textit{Tag}, m')$ from the party P.
2. Upon receiving a message $(P, \textit{Tag}, m)$ from $\mathcal{S}$, see if $(P, \textit{Tag}, m)$ already exists in the private database. If so, then write $(P, \textit{Tag}, m)$ on the public database. Ignore any further message $(P, \textit{Tag}, m)$ from $\mathcal{S}$.

[a] The contents of the public database is known to all parites. In our protocols, parties need to wait until a specific party P publishes $(\textit{Tag}, m)$ on the bulletin board. This means that, they wait until $(P, \textit{Tag}, m)$ appears on the public database.

Functionality 7 (Proof of Correct Decryption). The proof of correct decryption functionality $\mathcal{F}_j^{\mathrm{pd}}$ executing with dummy mix-servers $\overline{\mathcal{M}}$ and ideal adversary $\mathcal{S}$ proceeds as follows.

1. Wait for an input $(\textit{SecretKey}, sk)$ from dummy mix-server $\mathcal{M}_j$ and then hand $(\textit{SecretKey}, j)$ to $\mathcal{S}$.
2. Wait for a common input (m, c) from dummy mix-servers, i.e., $(m, c) = \mathsf{Agree}(\textit{Verify})$, and send *True* or *False* to $\mathcal{S}$ and $\overline{\mathcal{M}}$ depending on if $m = \mathsf{Dec}_{sk}(c)$ or not.