Spotlight on Research Course – Final Project 2025

Part 2 : Article

Submitting: Simon Levy, Yaheli Levit, Eden Sokolov Gulko, Shira Abudi and Guy Nevo.

Our topic: Verifiable Mix-Nets for Voting

This article presents a set of concise lecture notes from MIT's course 6.879 "Special Topics in Cryptography", lectures 19-20 on Verifiable Mix-Nets.
Taken together, they offer a complete, rigorous, and accessible blueprint for constructing an electronic voting system that both guarantees ballot secrecy and allows for public verifiability.
The notes center on ElGamal based re-encryption mix nets, taking advantage of the malleability of the ElGamal encryption to anonymize ballots and resorting to zero-knowledge proofs to guarantee that no votes were added, modified, or changed in any way.

The system works as follows:

1) A public key is generated by jointly distrusting authorities.
2) The voters encrypt ballots using the public key.
3) A number of independent mix servers iteratively re-encrypt and shuffle the ciphertexts.
4) During the process, everyone can check that each server's output is a re randomized permutation of the ciphertext it received as its input.
5) A threshold of the authorities decrypt the last batch of ciphertexts to get the final tally of votes.

The adversarial side has two main components:

1) Voters may try to cheat the system prior to blending by creating ballots that negate or duplicate other ballots without discovering their messages.
2) Mix servers could attempt to manipulate the count or leak permutations (violating privacy).
   The notes contain the exact cryptographic protection against both concerns.

The mathematics in these lecture notes involve discrete mathematics and simple algorithms, and present each cryptographic subroutine as they are introduced, making it ideal for undergraduate computer science students to read and engage in as an introduction to more cryptography topics.

<u>Why this topic matters:</u>

Electronic voting presents a unique problem. To maintain fairness, voters must be anonymous, but the vote total must be publicly verifiable.
The solution to this problem is verifiable mix nets.
Mix nets solve by making public only encrypted ballots but using mathematics to ensure inputs and outputs are a perfect one to one match.
Correctness is then publicly verified without the public trusting the mix servers.
Naturally, the lectures also describe the edge of trust that zero knowledge proofs present: correctness of a vote count is computationally verifiable but still depends on at least one server keeping its permutation secret.
That is a clear realistic statement of assumptions that must be acknowledged and dealth with by any actual deployment.

<u>ElGamal encryption overview:</u>

ElGamal acts in a cyclic group G of prime order q with generator g.
A public key is $y = g^x$ for secret x. Encrypting a message $m \in G$ chooses random $t \in Z$ and outputs $C = (\alpha, \beta) = (g^t, m \cdot y^t)$.
A crucial property for mix-nets is re-encryption: given C, one may create $C' = (\alpha \cdot g^s, \beta \cdot y^s)$ for a random s. C' encrypts the same m but appears unlinkably different.
The notes build on the equivalence between "C and C' encrypt the same m " and "an equality of discrete logarithms holds" which is precisely what the Chaum–Pedersen protocol proves in zero knowledge.

Keeping Voters Honest: Proofs of Knowledge at Ballot Submission

Two subtle abuses—vote cancelling and copy-cat voting—can manifest if a malicious voter can inject related ciphertexts without "owning" them.
Lecture 19 proposes a simple, robust condition at submission time to solve this problem: when a voter presents an ElGamal ballot $C = (g^t, m \cdot y^t)$, they must prove knowledge of t in zero knowledge.
Intuitively, knowing t enables recovery of m, so a voter who cannot produce such a proof cannot submit "derivative" ballots that cancel or clone another voter's ballot.
The notes present an efficient 3-round HVZK protocol for knowledge of t.
The protocol works as follows: the first party, the prover, picks a random member of the group s, and then sends to the second party, the verifier, g^s.
The verifier then sends a random member of the group, c.
The prover then calculates r using $s, c,$ and t.
The verifier then checks that $g^r = g^{s+t*c}$.

There is an important design lesson to be learned, that being local proofs at ballot submission prevent global inconsistencies later, while also preserving privacy (the proof shows only that the voter knows t, not the vote m).

Keeping Mix Servers Honest: From a Naïve NP Statement to a Sound Strategy:

Formally, a mix server should prove the existence of a permutation π and re-encryption exponents $s_1, \ldots., s_n$ such that the output ciphertexts are exactly a re-randomized permutation of the inputs: $\alpha_{\pi(i),1} = \alpha_{i,0} g^{s_i}$ and $\beta_{\pi(i),1} = \beta_{i,0} y^{s_i}$ for every $1 \leq i \leq n$.
This is an NP statement, so in principle, a zero-knowledge proof exists.
The challenge is to do it efficiently and without leaking π.

Chaum–Pedersen equality-of-logs and HVZK:

Lecture 19 reviews the Chaum–Pedersen protocol for demonstrating to an honest verifier that two pairs $(a_1, b_1), (a_2, b_2)$ share the same discrete log.
In the ElGamal scenario, this guarantees that two ciphertexts are re-encryptions of the same plaintext, i.e., they only differ by new randomness.
The protocol works as follows: The first party, the prover, generates a random member of the group s, and sends a commitment - g^s and g^{r*s} (the rerandomization factor r is known to the prover, the message isn't).
Following that, the second party, the verifier, sends a random member of the group, c.
The prover then sends the verifier another member of the group t, calculated using the s, c, and r. Finally, the verifier checks if t matches the expected output.

The protocol is efficient (three-move), honest-verifier zero-knowledge (HVZK), and the standard "special soundness" is present and allows for witness extraction from two accepting transcripts with the same commitment and different challenges.

The 2×2 Building Block: Verifiable 2x2 mixes:

Lecture 20 takes a modular approach. rather than show a monolithic $n \times n$ statement directly, it constructs the mixture from many 2×2 verifiable mixes.
A 2×2 mixture takes inputs C_1, C_2 and returns re-encryptions C'_1, C'_2 either in the same order or reversed order.
The server needs to prove the disjunction:

$$(C_1 \approx C'_1 \wedge C_2 \approx C'_2) \vee (C_1 \approx C'_2 \wedge C_2 \approx C'_1)$$

where ≈ denotes "encrypts the same plaintext" (i.e., re-encryption).

The Lecture shows how to reduce this problem to four disjunctions and then illustrates an efficient protocol for proving each of the disjunctions, involving a special version of Chaum–Pedersen's equality-of-logs proof as the subroutine for "$C \approx C'$", and obtaining an OR-proof by splitting the verifier's challenge as $c = c1 \oplus c2$.
This special protocol works similarly to chaum Pederson.
The initial steps are similar, but the verification is changed. The new verification checks if $g^r = g^{s+v*c}$ and $y^r = A_2 * (\frac{m'_1}{m_1} * y^v)^c$.
The prover simulates a Chaum–Pedersen transcript for the false branch and provides a real response for the true branch, binding them together with the relation $c = c1 \oplus c2$.
From the properties of Chaum–Pedersen and the algebra of the split challenge we get completeness, special soundness, and HVZK.

Notably, a byproduct of this protocol is witness indistinguishability. Because the protocol is HVZK, the transcripts do not disclose the actual branch (swap or not) that was taken and from the properties of Chaum–Pedersen and the algebra of the split challenge compose well in parallel, something that may be helpful when a lot of 2×2 mixes are proven at the same time.

From 2×2 to $n \times n$ - Sorting Networks as the Skeleton:

To scale to n ballots, the lecture wires these 2×2 gadgets into a sorting network. Replace each comparator gate (a network of wires with 2 inputs x, y which outputs the max of the 2 values to the top wire, and the min to the bottom wire) by a 2×2 verifiable mix. Since sorting networks realize all permutations, certifying that each local gadget is a valid re-encryption implies that the whole network implements a permutation.
This is exactly what is needed for a verifiable shuffle that leaks nothing about π.
From this construction, 3 practical notes appear.
First, there is a trick that can simplify both the mix and the proof, which involves assigning each input a random tag temporarily, sorting the network based on the tags, and erasing the tags at the end.
Second, the proofs can be generated either by the commit-and-decommit shares of a master challenge or via Fiat–Shamir.
Lastly, putting off the proof until all the servers have completed the mix helps in parallelism.

An enticing, yet incorrect, shortcut for verification is to establish only that "every input matches some output" and "every output matches some input." Lecture 19 illustrates why that is not enough: a cheating server may alter the votes (e.g., changing the amount of votes a voter received without changing the total amount of cast votes) and yet meet those per-element criteria.

The solution to this problem is to force a structure that guarantees a bijective correspondence without revealing which bijection occurred.

<u>Efficiency and Deployment Issues:</u>

The notes give back-of-the-envelope calculations for precinct-scale elections: using $nlog^2n$ comparators for the sorting network and roughly 10 modular exponentiations per comparator, a set of 1,000 ballots represents around 10^6 exponentiations.
On a standard PC, preforming 50-100 modular exponentiations per second, the overall time per precinct is roughly 3–4 hours, a computably acceptable time for election night reporting.
They also mention quicker shuffle proofs (e.g., Neff's) that support 100,000 ballots around 20 hours.
Operationally, the lectures advise grouping in precincts or counties, as very-large batches slow the mixing down, and very-small ones decrease anonymity to a small set.
Such a trade-off is within the system design and not an afterthought.

<u>Detailed order of operations:</u>

<u>1) Ballot submission and creation</u> - Voters encrypt ballots with a public ElGamal key and pair each ciphertext with a zero-knowledge proof of knowledge of the encryption randomness t.
It foils the cancelling and copying attack but reveals no information about the vote.

<u>2) Mixing phase</u> - The mix server re-encrypts each of the ciphertexts with new randomness and shuffles the list.
It then demonstrates, without showing the permutation, that the result is a re-ordered permutation of the input, i.e. the result is indeed a re- encryption of the ciphertexts.
The lecture maps this overall requirement to a composition of 2×2 OR-checks synchronized with a sorting network.

<u>3) Public verification</u> - Anybody may verify all the posted proofs against the public bulletin board.
Since the proofs are zero-knowledge and witness indistinguishable, the verification does not disclose anything about individual vote-paths through the network.

<u>4) Final decryption and count</u> - After all the shuffles are validated, the trustees jointly decrypt the ciphertexts and count the ballots. Privacy reductions boil down to the assumption that at least one mix server maintains its permutation secret, while correctness is provided by the hardness of forging the posted proofs.

Conclusion:

Verifiable mix-nets achieve the rare combination: private protections for the individual and public verifiability at scale, built from well-understood pieces.
Lecture 19 argues the architecture and demonstrates why voter-side proofs prevent subtle misuses at the source, and Lecture 20 provides an efficient, modular proof that each mixing stage is a permutation plus re-encryption without disclosing the identity of the permutation.
In the resulting system, anyone can verify by checking against a public bulletin board but from which no one learns how anyone at all voted (assuming at least one honest mixer).