Advances in Cryptographic Voting Systems

by

Ben Adida

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[September 2006]

August 2006

Author ....
Department of Electrical Engineering and Computer Science
August 31st, 2006

Certified by ....
Ronald L. Rivest
Viterbi Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Advances in Cryptographic Voting Systems

by

Ben Adida

Submitted to the Department of Electrical Engineering and Computer Science
on August 31st, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Democracy depends on the proper administration of popular elections. Voters should receive assurance that their intent was correctly captured and that all eligible votes were correctly tallied. The election system as a whole should ensure that voter coercion is unlikely, even when voters are willing to be influenced. These conflicting requirements present a significant challenge: how can voters receive enough assurance to trust the election result, but not so much that they can prove to a potential coercer how they voted?

This dissertation explores cryptographic techniques for implementing verifiable, secret-ballot elections. We present the power of cryptographic voting, in particular its ability to successfully achieve both verifiability and ballot secrecy, a combination that cannot be achieved by other means. We review a large portion of the literature on cryptographic voting. We propose three novel technical ideas:

1. a simple and inexpensive paper-base cryptographic voting system with some interesting advantages over existing techniques,

2. a theoretical model of incoercibility for *human* voters with their inherent limited computational ability, and a new ballot casting system that fits the new definition, and

3. a new theoretical construct for shuffling encrypted votes in full view of public observers.

Thesis Supervisor: Ronald L. Rivest
Title: Viterbi Professor of Electrical Engineering and Computer Science

Acknowledgments

Ron Rivest was already my Master's Thesis Advisor in 1998-1999.He welcomed me back into his group after my 4-year leave of absence and found research funding for me by my second semester. His top-notch advice, endless encouragement, and bottomless well of knowledge were invaluable. His constant dedication to the field of cryptography, to scientific rigor, and to teaching, amaze me still every day. My committee members, Shafi Goldwasser and Srini Devadas, were so encouraging that my defense felt more like a family get-together than an exam (okay, not quite.)

The Knight Foundation funded my work with the Caltech/MIT Voting Technology Project. Stephen Ansolabehere helped me get started with the group and provided precious advice along the way. His unrivaled expertise in election matters was crucial to our debates. His humor made our meetings seem much shorter than they actually were.

Hal Abelson has been a close mentor for 10 years. His predictions seem outlandish only because they are, conservatively, 2 years ahead of their time. His dedication and unequaled talent for teaching will always be among my greatest inspirations. His positive impact on the world is exceeded only by his humility, and I am honored to have the opportunity to work with him in any fashion.

Andy Neff, my long-distance collaborator, possesses a strength in the face of absurd adversity that I have rarely seen. His ideas are groundbreaking and elegant, and I struggle to keep up. He is maybe the ultimate unspoken hero of voting. If we ever achieve secure verifiable voting, it is hard to imagine that one of his ideas wouldn't be part of the solution.

Douglas Wikström, my other long-distance collaborator, smacked me around for incomplete proofs (I deserved it.) Mike Bond, Jolyon Clulow, and Ross Anderson hacked my ATM card with a British accent. John Herzog, Paul Youn, Amerson Lin, provided the American (and Singaporian) counter-part.

Susan Hohenberger and I finally got our anti-phishing paper published, though it was a book chapter before it was a conference paper (go figure.) She and my other officemates Steve Weis, Seth Gilbert, and David Liben-Nowell, suffered my incessant yapping about policy, privacy, copyright reform, and free software.

Chris Peikert reviewed large portions of this work as a last-minute favor, catching bugs I would surely have missed, and suggesting brilliant improvements. He, along with Rafael Pass, Alon Rosen, abhi shelat, and Guy Rothblum were always willing to explain the theoretical crypto concepts I had forgotten, missed, or just not understood.

David Chaum, the man behind a disproportionate number of the major cryptographic breakthroughs of the last 30 years, got me involved in voting standards work and provided endless entertainment at CRYPTO. Every voting crypto idea seems to have its root in one of David's early proposals.

Danny Weitzner, Ralph Swick, and Eric Miller were my guides in the wild world of the W3C. Zak Kohane, Pete Szolovitz, and Ken Mandl were my guides in the land of medical informatics, bioinformatics, and genomic medicine.

Philip Greenspun helped me return to MIT by offering me a TAship in his course for my first semester back.

Gerald Ruderman provided unbeatable advice and encouragement.

My parents, Pierre and Yvette, eventually stopped asking questions about the detail of my work—long past the point I expected—but never stopped encouraging me, congratulating me on successes big and small, and dismissing my failures as other people's faults. I could not have asked for better parents. Claire regaled me with her stories of far-away lands and adventures, and Juliette found her path to success while helping me plan my wedding from across the Atlantic. I could not have asked for better sisters.

Speaking of wedding... I met my wife Rita the first week I returned to MIT. The moment I met her, I knew. The rest, all of this, is details.

This work is dedicated to my grandparents, Marguerite, Adolf, Salomon, and Marcelle.
They encouraged me to explore, and I went looking for things to decrypt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the most important tasks of a democratic government is the planning and execution of the election that designates its successor. Not surprisingly, it is also one of its most challenging tasks, one whose requirements and constraints are remarkably strict. Thus, dubious results, failing technology, and ingenious methods of fraud have been noted throughout election history. Lever machine counters have been rigged, ballot boxes have been lost or magically found, and dead citizens have voted. Legitimate voters have been coerced in various ingenious ways, including chain voting, spreading false election date and location information, and instilling false fears regarding the consequences of showing up to vote (arrests, jury duty, ...).

In the United States, the 2000 Presidential Election caused much stir: Bush lost the popular vote but won the Electoral College, including a win in Florida by a margin of 500 votes [67]. Numerous complaints were aired: the "butterfly ballot" in Broward County was misleading, the punchcard system failed to record a number of votes, and more than 50,000 absentee ballots went missing [10]. This debacle served as a public wake-up call that elections are far from perfect.

Equipment failures were well known to election officials long before Gore vs. Bush [9]. However, these failures had not previously been implicated in such a close-call election. The fallout from the drawn-out results-certification process of 2000 caused many States to reconsider their voting equipment, often including hastened moves to newer, more computerized solutions so as to be "not like Florida" [113]. These changes raised questions of their own, notably among a number of computer scientists who feared that fully computerized voting would complicate or completely prevent the election verification process [11, 163]. The controversy and debate are ongoing, with much discussion about what will happen in the upcoming 2006 mid-term elections.

In this dissertation, we present recent advances in *cryptographic voting systems*, a type of election system that provides mathematical proofs of the results—rather than of the machines. We begin with a review of the functional requirements of voting, the specific issues that make voting fairly complex, and an overview of the basic technical concepts behind cryptographic voting. We note with interest that, to this day, there is no known way to execute a truly secure and verifiable election without some elements of cryptography.

1.1 A Brief History of Voting

We begin with a history of election practices and technology, focusing specifically on the introduction of the secret ballot. Details can be found in the work of Jones [99] and in the recent book by Saltman [148].

Early Voting. The first accounts of voting are from Ancient Greece, where male landowners voted in "negative elections": any politician receiving more than 6000 votes was exiled for ten years. Votes were recorded on broken pieces of porcelain called ostraca [176]. In the 13th century, Medieval Venice introduced approval voting, where each candidate received a thumbs-up or thumbs-down from each voter, with the winner designated as the candidate with the highest overall approval rating.

In the United States, the first elections were viva-voce: voters were sworn in and simply called out their preferences. Multiple clerks recorded the votes separately and in parallel to prevent error. In the early 1800s, paper ballots were introduced, though these were generally produced by the voters themselves or by political parties. It is this practice which led to the "party ticket," where a voter could easily pick up a pre-printed, all-Republican or all-Democrat ballot, and cast it as is. To this day, voting systems in the US strive to recreate the simplicity of "party ticket" voting.

The Secret Ballot. In 1858, Australia introduced a new ballot mechanism: ballots were to be printed by the state, kept in a safe place until election day, and distributed, one at a time, to each eligible voter, who then voted in an isolation booth. This method was first imported to the United States in 1888, following significant concerns of voter fraud. It was first used widely in the 1892 Presidential elections.

One of the most dramatic changes introduced by the Australian ballot was secrecy. No longer could voters be influenced by external sources, at least not openly and systematically. As we will see, it is also this change which radically altered the auditability of elections.

Mechanization of Election Equipment. The standardization of the ballot enabled new equipment for ballot casting and counting. Lever machines were first introduced in 1892 in New York and became widespread in medium-to-large voting communities in the mid 1900s. To this day, they are still used in a number of communities, notably in New York.

A lever machine isolates the user inside a booth with a privacy curtain. Levers are organized in a grid, where columns generally indicate the political party and rows indicate the election race. The voter simply turns the levers of her choice. When satisfied, some final action—pulling a larger lever or opening the privacy curtain—causes the ballot to be cast and the levers to reset. Mechanical counters keep track of the total count for each candidate. Consequently, only these aggregates, not individual ballots, are preserved.

In the 1960s, punch card systems were introduced in a few states. A clever design allows a single small card to offer more than 200 possible voting "positions," i.e. candidates. By carefully aligning the questions on flip-cards so that the proper column of the punch card is exposed, a voter can simply punch the hole next to desired candidate and use a single punch card for an entire election consisting of multiple races. Individual punch cards are then cast and tabulated centrally using either electro-mechanical readers or a typical computer-based punchcard reader. Though punch cards preserve individual ballots, the imprecision of certain hole punches was known to be problematic long before the highly visible exposition of this weakness during the 2000 elections. In addition, the clever question layout—used to maximize the number of punch holes on the small punchcard—led directly to the "butterfly" layout that received much criticism in the 2000 aftermath.

Computerization of Election Equipment. The 1960s also saw the introduction of optical-scan machines for voting. Using a pencil or a pen, the voter marks up a paper ballot in the appropriate locations, either by connecting two arrows when choosing a given candidate or, in more modern systems, filling in the appropriate bubble—much like a standardized multiple-choice test. An optical scanner is then used to tally the ballots.

Two major types of optical-scan voting systems exist: central-count and precinct-based. In a central-count optical-scan system, filled-in ballots are collected, unprocessed, at the precinct, and later delivered to a central location where they are scanned and counted. In a precinct-based system, voters feed their own ballot into a scanner, which immediately validates the ballot, either rejecting it if improperly marked, or passing it straight into a locked ballot box if it is correct.

It has been shown that precinct-based counting prevents a significant fraction of mistakes, including ballot management (ballot for the wrong precinct) and human error (voting for two candidates in the same race, i.e. overvoting) [36]. Optical scan machines are used quite extensively in modern US elections, usually in precinct-based mode, with approximately 30% market share in the 2000 and 2004 elections, and a predicted 40% share in 2006 [61].

DREs. In recent years, a new type of computerized voting equipment has appeared: the Direct Recording by Electronics (DRE) machine. These machines are generally personal-computer-type equipment running special-purpose voting software, often on a generic operating system like Windows. Ideally, the machines are physically hardened, preventing access to the typical personal-computer connectors, e.g. USB ports. DREs are particularly interesting because they solve a number of complex operational problems:

- ballots can easily be offered in different languages,
- voters with vision impairment can magnify the screen or use a headset that provides auditory feedback,
- ballot management is vastly simplified using memory cards instead of paper.

At the same time, these machines have come under significant criticism because they lack a tamper-proof audit-trail. Voting activists and computer scientists are worried that these machines could produce erroneous results, either because of bugs or malicious code, that would go undetected [169, 111]. In particular, the worry is that a voter's choice would be incorrectly recorded at casting time. Since the only feedback a voter obtains is from the voting machine itself, a mistake at ballot casting time would be completely unrecoverable and undetectable.

The VVPAT. To thwart this problem, some have supported the Voter-Verified Paper Audit Trail, first proposed by Mercuri in 1992 [115]. In VVPAT-based voting machines, once the voter has finished filling out her ballot using the computer interface, the machine prints out an audit of the entire ballot on a scrolling receipt visible to the voter behind glass. The voter then gets to confirm or cancel her vote. The audit trail effectively short-circuits the machine's possible mistakes. Ideally, in the case of a recount, the paper trail would be used instead of the electronic record. VVPAT machines are only just beginning to appear in the voting equipment market: November 2006 will likely mark the first time they are used on a significant basis in the United States, with 5 states expected to implement it [98].

1.2 What Makes Voting so Hard?

To illustrate the complexities of voting, it is useful to consider a precise hypothetical scenario with the following characters:

- **Alice** and **Adrienne**, two voters,
- **Carl**, a coercer who wishes to influence Alice,
- Blue and Red, two options between which voters are deciding in the election.

Alice wants to vote for Blue, while Adrienne wants to vote for Red. Carl wants to coerce Alice so that she votes for Red instead, thereby swinging the election. It is worth noting that Carl, the coercer, may be an election official. This setup is illustrated in Figure 1-1.

1.2.1 Verifiability vs. Secrecy

In elections, there is a functional conflict between verifiability and secrecy. On the one hand, Alice wants to verify that the entire voting process happened correctly, in particular that her individual vote was counted appropriately as Blue. However, if Alice obtains enough information from the voting process that she can convince Carl of how she voted, then vote selling becomes a threat: Carl can offer Alice money in exchange for her voting Red instead of Blue.

Somehow, we want Alice to obtain enough information to *personally verify* that her vote was indeed recorded as Blue, but not so much information that she could convince Carl. More concretely, if Alice votes Blue and Adrienne votes Red, both should receive assurance that their vote was cast according to their preference and counted accordingly. In addition, both can tell Carl equally valid stories about how they allegedly voted Red. Alice is lying,

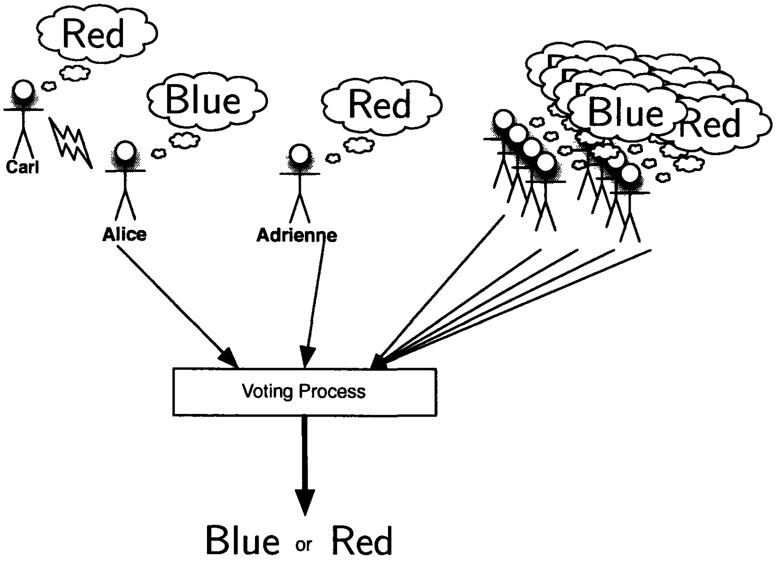


Figure 1-1: Participants in an election process. Carl wants to coerce Alice into voting Red instead of her natural choice, Blue. Carl may be an election official.

and Adrienne is telling the truth, but Carl cannot tell the difference. Thus, Carl has no incentive to pay for votes, since he cannot tell if his money is going to "good" use. It is not immediately clear that resolving this conflict is possible!

1.2.2 Threat Modeling: Planes, Banks, and Voting Machines

A common, recent criticism of voting system failures compares voting to existing complex systems, like the operation of airplanes or transaction processing by banks [117]. At first glance, the criticism seems warranted: if we can build large aluminum cylinders, load them with hundreds of people, project tens of thousands of them every day at half the speed of sound and at an altitude of 6 miles, land them at their intended destination, all with fewer than one fatal crash a year, even in the face of malicious adversaries, then surely we can build reliable voting systems! Similarly, if banks process millions of transactions a day, recording every dollar in and out of each customer's bank account, with receipts allowing customers to audit their account on their own, then surely we can reliably record 100 million votes once every 4 years and provide adequate auditing!

These analogies make three notable mistakes, the last and most important of which was noted by renowned security expert Bruce Schneier in 2001 [156]. First, the *incentive* to throw a federal election is grossly underestimated. Second, the *adversarial model* for airplanes and ATMs are, somewhat surprisingly, less demanding than for a federal election. Third, the *failure detection and recovery process* in the case of an airplane or bank failure is generally well understood: thanks to full auditing, appropriate recovery actions can be taken. In the case of elections, it isn't clear that failures can always be detected, and, if they are, recovery is often expensive or even impossible.

Incentive. Influencing the outcome of a federal US election is worth quite a bit of money. The presidential campaign budget for both parties in 2004 reached $1 billion [34], and there is, of course, no certainty of winning. A more certain mechanism for influencing the election outcome might be worth even more, especially considering the influence wielded by a US president. Though the counter-incentive is significant—voting fraud is a felony—one cannot ignore this significant incentive to commit fraud in the first place. Even with currently established stiff penalties and when the gains are relatively small, there is strong empirical evidence that voting fraud is a regular occurrence [9].

Adversaries. The threat model for safe aviation is well defined: it is assumed that passengers may be adversaries, which leads to the numerous security checks and the recent implementation of the Transportation Safety Authority. By contrast, it is generally assumed that there is significant time and resources available long before the flight to ensure that *pilots are not adversarial.* By the time they board the plane, pilots are assumed to be honest. The presence of a co-pilot indicates planning for random failures, though it is hardly a defense against malicious attacks: significant damage can be inflicted by a rogue pilot, as some historical cases have suggested [177]. Fortunately, few legitimate pilots are ever motivated to commit such acts.

In the case of personal banking, the threat model to the individual customer is also well defined: adversaries are most likely outsiders—identity thieves, ATM bandits, etc. Importantly, all data is available to honest participants: both bank officers and the customer can view the transaction trail and account balance. It is also a feature of the system that the customer can prove her bank balance to a third party, e.g. when applying for a mortgage. [1]

By contrast, in an election, any participant, including especially insiders, might want to throw off the results. Voters might be corrupt. Election officials might be corrupt. No assumptions of honesty can be made of any participant, and all participants are potentially highly motivated to perform fraud, since the outcome is a single, extremely important result for all participants.

Failure Detection and Recovery. In aviation, failures are difficult to miss: planes malfunction or crash in tragically obvious ways. Extensive records of passenger and crew names, on-board equipment, and flight data are kept, including the use of "black boxes" to recover this data in the case of a crash. If a crash does occur, an extensive and expensive investigation usually ensues, and any technical problems immediately lead to change recommendations from the National Transportation Safety Board [123].

In banking, the situation is quite similar. There is significant investment and emphasis on detecting and resolving failures: customers are offered receipts of every transaction, which

[1] Note that the threat model for the bank is quite different, as the bank must be far more worried about insider attacks. That said, this is not the threat model against which we're comparing here: we're concerned about the comparison often made between voting system audit trails and personal ATM receipts.

they can reconcile against regular account statements. Duplicates of lost records can be requested and obtained with ease. Video records of ATM transactions are kept. Electronic records of online activity are maintained. If a banking customer or the bank itself finds a discrepancy, a review of all audit trails almost always leads to discovery of the malfunction and rectification of the problem with little overhead.

By contrast, detecting election failure using today's voting protocols is quite difficult. It is highly conceivable that successful fraud might go completely undetected, given that a significant portion of the audit information is *voluntarily destroyed* to ensure ballot secrecy. Years after the 2000 and 2004 elections, there isn't even consensus on whether fraud occurred [175, 101]. If an error *is* detected, it is unclear how one might go about resolving the issue: the only solution may be to re-run the election, which could, in and of itself, vastly change the election result.

Fixing the Analogies. If voting is compared to banking, then one should imagine a banking system where the bank cannot know the customer's balance, and even the customer cannot prove her balance to her spouse, yet somehow she receives enough assurance that her money is safe. If voting is compared to aviation, then one must imagine that pilots are regularly trying to crash the plane, and that we must ensure that they are almost always unsuccessful, even though, in this imaginary world, plane crashes are particularly difficult to detect. These significant additional constraints lead to a clearer appreciation of the challenges faced by voting system designers.

1.2.3 Auditing a Partially Secret Process

Voting is particularly difficult because it requires a public audit of a process which must ensure a significant amount of secrecy. This secrecy cannot be guaranteed by trusting an all-powerful third party: even the auditors cannot be made aware of how individual citizens voted. In addition, this audit must be convincing to mutually distrusting observers.

Such apparently conflicting requirements often call for cryptography. However, before we explore the solutions offered by cryptographic techniques, let us consider the security properties of classic voting systems, those in use around the world today.

1.3 Classic Voting Systems

In the conflict between auditability and secrecy, election systems must often favor one or the other in a compromise. All election systems used in the United States since the introduction of the secret ballot in 1892 have favored secrecy over auditability. Typically, secrecy is ensured by forced physical dissociation of identity and ballot, e.g. dropping an anonymized ballot in a physical ballot box. On the other hand, election auditing generally depends on a properly enforced chain of custody of election equipment and ballots.

1.3.1 Chain-of-Custody Security

Current voting solutions use some type of voting machine to assist voters in preparing and casting a ballot. The machines are built by private companies, according to various state-specific standards. Independent testing agencies (ITAs) are called upon to evaluate and certify the various voting equipment, though their lack of power and limited means has often been criticized [51]. Election officials generally perform additional tests, usually in the weeks prior to an election, and, sometimes, in parallel with the election [58], to provide assurance that the machines are working as expected. On election day, voters cast their ballots via these machines, and the resulting votes end up in some ballot box, either physical or digital. Election officials then transport these ballot boxes to a counting facility, where they are tallied. The aggregate results are finally posted for all to see. This approach presents three major limitations:

1. **Voters must verify by proxy**: only election officials can ensure that various testing procedures are adequate to begin with. Voters receive some indirect indication of test results, such as the machine testing receipts posted on the wall at the start and end of election day. However, voters cannot directly verify that the ballot boxes themselves (digital or physical) are handled appropriately throughout election day.

2. **Verification strongly depends on chain-of-custody**: election officials must maintain a well-audited chain-of-custody to defend against malicious problems. For an election to run correctly, every transition must be performed correctly, and the chain of custody of machines and ballots must be strictly respected at all times. A single failure can open the door to significant corruption of the election results.

3. **Recovery is very difficult**: the error detection mechanisms are few and coarse-grain, able only to notice honest mistakes, rarely malicious attacks. If an error is detected, some voting systems allow for ballots to be recounted. Such a recount can only address failures in the tallying process, however: recovering from an integrity breach on ballot boxes or voter intent capture requires that the election be re-run, as it is usually impossible to tell the legitimate ballots from the fraudulent ones, or properly cast votes from improperly cast votes.

In other words, in order to implement the secret ballot, current voting systems resort to a trust-by-proxy, chain-of-custody-based verification mechanism, with a "point-of-no-return" ballot hand-off, beyond which recovery is limited. This approach is both highly constraining and prone to significant error. Whether paper, optical-scan, or touch-screen, classic election methods significantly compromise verifiability in order to achieve ballot secrecy. This chain-of-custody verification process is diagrammed in Figure 1-2.

1.3.2 The Erosion of the Secret Ballot

As a result, a number of new election proposals have tipped the scales in the other direction: in order to address the lack of verifiability in current election systems, they propose to

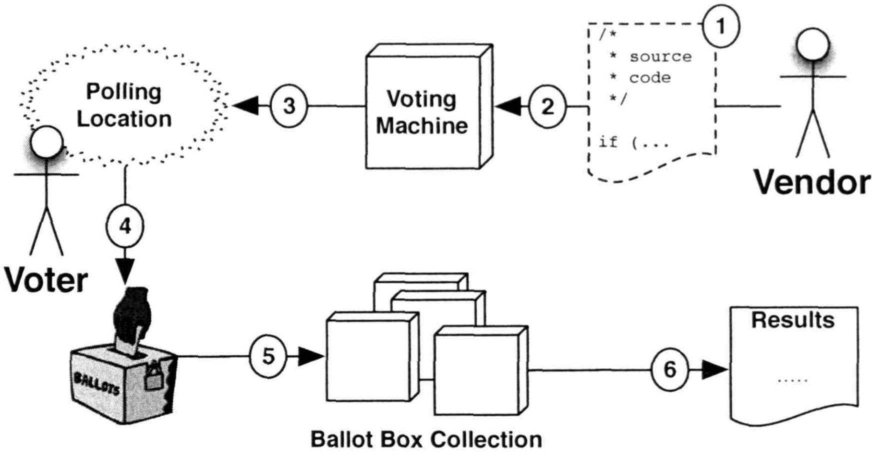


Figure 1-2: **Chain-of-Custody Voting** - every step must be verified. (1) The source code for voting machines is read and checked. (2) The installation of the voting machine is verified to ensure that the verified software is indeed installed. (3) The voting machines are sequestered and sealed prior to the election, and must be secured against physical attacks (e.g. installation of malicious hardware components). (4) Election officials ensure that only eligible voters cast a ballot. (5) Ballot boxes are sealed and collected with care. (6) Tallying occurs in a secured area, ensuring that no ballots are maliciously lost or injected.

compromise ballot secrecy. In some cases, this erosion of ballot secrecy occurs without anyone noticing. There is, in some election communities, a surprising belief that election integrity can be achieved even if the system is coercible.

Internet Voting (Switzerland). Switzerland holds multiple referenda per year. In order to improve convenience and increase voter turnout, the Swiss have chosen to introduce Internet-based voting using any home computer [35]. Unfortunately, internet voting is inherently coercible [50], as there is no guaranteed privacy during ballot casting.

Vote By Mail (Oregon and Beyond). Oregon has long offered voting by mail to its residents, in part due to the large distances voters have to travel to reach a polling location. Recently, a grassroots effort has emerged to promote the same vote-by-mail approach in other states [172]. This group strives to make elections more convenient and bypass the complexity and cost of in-person elections. Most recently, a task force in San Diego suggested a move to permanent vote-by-mail for all [153]. Unfortunately, much like Internet voting, vote-by-mail is inherently susceptible to coercion, with detection almost impossible.

Return to Public Voting? Among the voting activism groups, some individuals have recently proposed explicitly doing away with the secret ballot altogether. They claim that "auditability and secrecy are incompatible" [112], propose that auditability is more important than secrecy, and conclude that we should simply give up ballot secrecy.

Unfortunately, though the secret ballot may seem inconsequential, it is, historically, one of the more important developments in democratic elections. A recent study of Chilean election data shows that ballot secrecy, introduced for the first time in Chile's 1958 elections, has "first order implications" on the election results and subsequent policy decisions [97]. Coercion is notoriously difficult to detect, and, though not currently rampant, may return in force if the secret ballot is compromised. We have reached a critical juncture, where the public outcry for verifiability must be addressed, for fear that the pendulum will swing too far in the other direction, jeopardizing the secret ballot and thus the ability to honestly gauge voter intent.

1.3.3 The Voter-Verified Paper Audit Trail

As previously described, one proposed solution for verifiability is the Voter-Verified Paper Audit Trail, abbreviated VVPAT. Though there are some concerns regarding the practicality of VVPAT machines [166], there is no question that a properly operating VVPAT machine would significantly simplify the verification chain. VVPAT effectively short-circuits the voting equipment: voters get the ease-of-use associated with computer voting, while the paper trail provides a direct mechanism for verification of the voting machine's output.

However, it is worth noting that even VVPAT does not change the *nature* of the verification process: a chain of custody, albeit a shorter one, must still be maintained and audited, so that the following questions can be answered:

- Do the accepted paper trails get properly deposited in the ballot box? Do the rejected paper trails get properly discarded?

- Are the ballot boxes of paper trails appropriately safeguarded during election day?

- Are the ballot boxes of paper trails appropriately collected and safeguarded after the polls close? What are the safeguards against the introduction of extraneous ballot boxes?

- Are the paper trails properly tallied? Using what process?

In other words, the VVPAT short-circuits the custody chain prior to ballot casting. The verification process for everything that follows the ballot hand-off, however, remains a chain of custody that must be properly enforced at all times.

1.4 Cryptographic Voting Schemes

We now consider cryptographic voting systems at a high level. We pay specific attention to the end-to-end nature of the verification provided by such systems, and how, consequently, any observer can verify the proper operation of a cryptographic election completely.

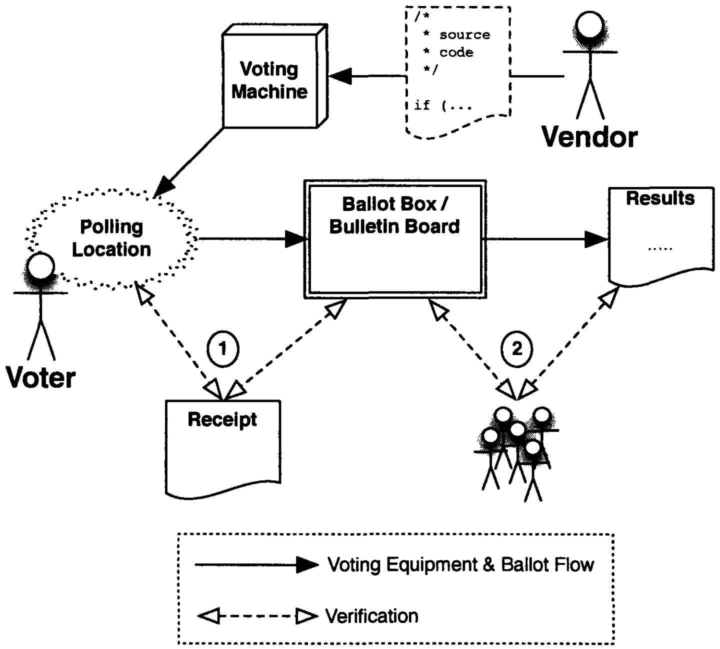


Figure 1-3: **End-to-End Voting** - only two checkpoints are required. (1) The receipt obtained from a voter's interaction with the voting machine is compared against the bulletin board and checked by the voter for correctness. (2) Any observer checks that only eligible voters cast ballots and that all tallying actions displayed on the bulletin board are valid.

1.4.1 End-to-End Verifiability

When dealing with complex systems, software engineering has long relied on the "end-to-end" principle [152], where, in order to keep the system simple, the "smarts" of the system are kept at higher levels of abstraction, rather than buried deep in the stack. For example, when routing packets on the Internet, very few assumptions are made about the underlying transport mechanism. Instead, checksums are performed by sender and recipient to ensure end-to-end integrity in the face of random mistakes, and digital signatures are applied to thwart malicious modifications of the data. No details of traffic routing need to be verified in either case; instead, a certain property is preserved from start to finish, regardless of what happens in between.

Though not all systems are amenable to such a design, *voting systems are.* Rather than completely auditing a voting machine's code and ensuring that the voting machine is truly running the code in question, end-to-end voting verification checks the voting machine's *output* only. Rather than maintain a strict chain-of-custody record of all ballot boxes, end-to-end voting checks tally correctness using mathematical proofs. Thus, the physical chain of custody is replaced by a mathematical proof of end-to-end behavior. Instead of verifying the *voting equipment*, end-to-end voting verifies the *voting results* [147].

As an immediate consequence, *one need not be privileged to verify the election.* In a chain-of-custody setting, one has to keep a close eye on the process to be certain of correct execution ; only election officials can do this directly. In an end-to-end verifiable setting, anyone can check the inputs and outputs against the mathematical proofs. The details of the internal processes become irrelevant, and *verifiability becomes universal*, as diagrammed in Figure 1-3.

Cryptography makes end-to-end voting verification possible. At a high level, cryptographic voting systems effectively bring back the voting systems of yore, when all eligible citizens voted publicly, with tallying also carried out in public for all to see and audit. Cryptographic schemes augment this approach with:

1. *encryption* to provide ballot secrecy, and

2. *zero-knowledge proofs* to provide public auditing of the tallying process.

1.4.2 A Bulletin Board of Votes

Cryptographic voting protocols revolve around a central, digital *bulletin board*. As its name implies, the bulletin board is public and visible to all, via, for example, phone and web interfaces. All messages posted to the bulletin board are authenticated, and it is assumed that any data written to the bulletin board cannot be erased or tampered with. In practice, implementing such a bulletin board is one of the more challenging engineering aspects of cryptographic voting, as one must worry about availability issues beyond data corruption, such as denial-of-service attacks for both data publication and access. There are, however, known solutions to this problem [110].

On this bulletin board, the names or identification numbers of voters are posted in plaintext, so that anyone can tell *who* has voted and reconcile this information against the public registry of eligible voters. Along with each voter's name, the voter's ballot is posted in encrypted form, so that no observer can tell *what* the voters chose. Two processes surround the bulletin board. The *ballot casting process* lets Alice prepare her encrypted vote and cast it to the bulletin board. The *tallying process* involves election officials performing various operations to aggregate the encrypted votes and produce a decrypted tally, with proofs of correctness of this process also posted to the bulletin board for all observers to see.

Effectively, the bulletin board is the verifiable transfer point from identified to de-identified ballots. Votes first appear on the bulletin board encrypted and attached to the voter's identity. After multiple transformations by election officials, the votes end up on the bulletin board, decrypted and now unlinked from the original voter identity. Whereas classic voting schemes perform a complete and blind hand-off—i.e. the ballot is dropped into a box –, cryptographic voting performs a *controlled hand-off*, where individual voters can trace their vote's entry into the system, and any observer can verify the processing of these encrypted votes into an aggregate, decrypted tally. This process is illustrated in Figure 1-4.

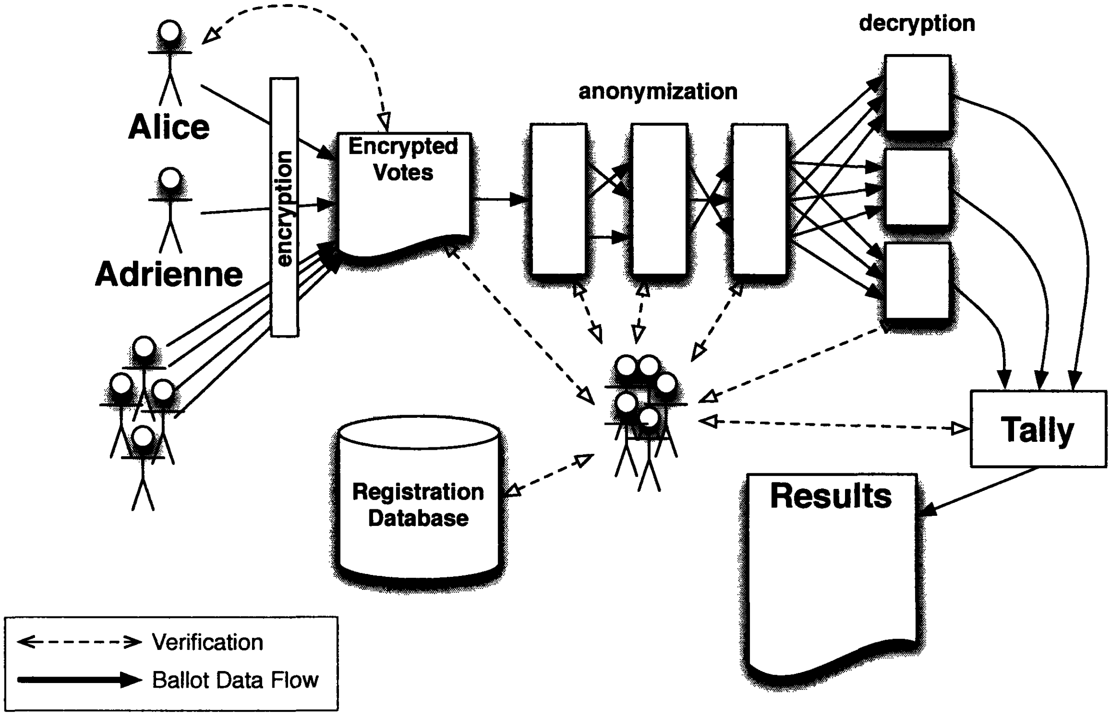


Figure 1-4: **Cryptographic Voting at a High Level** - voters cast an encrypted ballot on a bulletin board, where voter names can be checked by anyone against a public voter registration database. Then, election officials proceed to anonymize the votes and jointly decrypt them, providing proofs along the way that any observer can verify. The results are then posted for all to see.

1.4.3 A Secret Voter Receipt

Before Alice's encrypted ballot appears on the bulletin board, she must prepare it using some process that gives her personal assurance that her ballot has been correctly encrypted. Recall that, in providing Alice with this assurance, the system cannot enable her to transfer this same assurance to Carl, as this would enable Carl to influence her decision. For this purpose, all current cryptographic voting schemes require that voters physically appear at a private, controlled polling location: it is the only known way to establish a truly private interaction that prevents voter coercion.

In many proposed cryptographic schemes, Alice interacts privately with a computerized voting machine. She makes her selection much as she would using a classic voting machine, answering each question in turn, verifying her selections on screen, and eventually confirming her vote. The machine then produces an encryption of Alice's ballot, and begins a verification interaction with Alice.

This interaction is a type of *zero-knowledge proof*, where the machine proves to Alice that her encrypted vote indeed corresponds to her intended choice, without revealing exactly the details of this correspondence. In one particularly interesting scheme, Neff's MarkPledge

[29], Alice obtains a printed receipt that includes her encrypted vote and some confirmation codes. By comparing the codes on the receipt with those on the screen of the machine, Alice can be certain that the machine properly encoded her vote. In addition, since Alice can easily claim, at a later point, that a different confirmation code appeared on the screen, she cannot be coerced. This scheme is diagrammed in Figure 1-5 and explored and extended in Chapter 5.

Using this encrypted receipt, Alice can verify that her encrypted ballot appears correctly on the bulletin board. Given that the ballot is encrypted, she can even give a copy of her receipt to helper political organizations - e.g. the ACLU, the NRA, the various political parties—so that they may verify, on her behalf, the presence of her encrypted vote on the bulletin board.

Paper-Based Cryptographic Voting. In 2004, Chaum [40] was the first to propose a cryptographic scheme that uses paper ballots. The ballot in question is specially formed: after filling out the ballot, Alice physically splits it into two pre-determined halves, destroys one, and casts the other while taking a copy of this same half home with her as a receipt. This separation effectively encrypts Alice's choice: only election officials with the proper secret keys can recover Alice's choice during the tallying process.

In most of these paper-based schemes, the paper ballot must be verified *prior to voting* to ensure that the two halves are consistent with one another. Without this verification, a fraudulently created ballot could corrupt the proper recording of the voter's intent. In Chaum's latest version, Punchscan [66], a second, post-election verification can also verify the correctness of Alice's ballot. The details of the Chaum scheme are reviewed in Chapter 4, which also describes Scratch & Vote, our proposed evolution of Chaum's scheme, with methods to simplify the ballot face and pre-voting verification stage.

1.4.4 Tallying the Ballots

Once all encrypted votes are posted on the bulletin board, it is time to tally them. Note that no single election official can decrypt these individual votes: the secret key required for decryption is shared among a number of election officials, who must collaborate for any decryption operation. This is extremely important, as any decryption at this point would violate the ballot secrecy of the voters in question. The decryption process must be well controlled to protect privacy.

Two major techniques exist for just this purpose. The first uses a special form of encryption—called homomorphic encryption—that enables the aggregation of votes under the covers of encryption, so that only the aggregate tally needs decryption. The second uses a digital version of "shaking the ballot box," where individual votes are shuffled and scrambled multiple times by multiple parties, dissociated from the voter identity along the way, and only then decrypted.

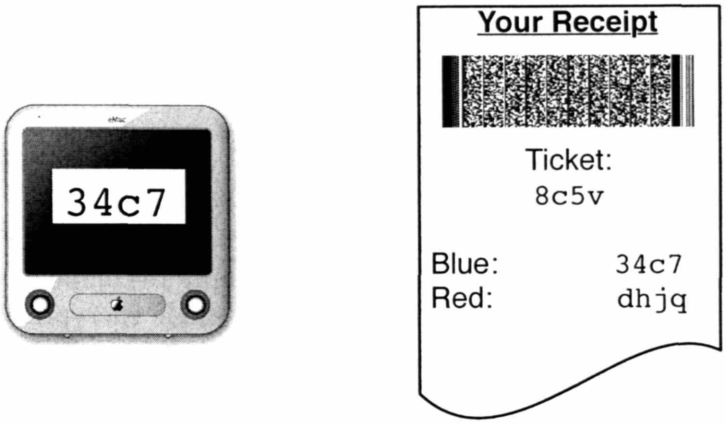


Figure 1-5: **A secret receipt in the Neff voter verification scheme** - The screen displays a code, which should match the voter's selected option on the receipt. In addition, the ticket number should match the voter's random challenge. Once the voter leaves the booth with only the receipt, it is impossible for her to provably claim that she saw `34c7` on the screen, and not `dhjq`.

Randomized Threshold Public-Key Encryption. Before we present either tallying method, it is important to note that all cryptographic voting systems use a special kind of encryption called *randomized threshold public-key encryption.* The public-key property ensures that anyone can encrypt using a public key. The threshold-decryption property ensures that only a quorum of the trustees (more than the "threshold"), each with his own share of the secret key, can decrypt.

In addition, using *randomized* encryption, a single plaintext, e.g. Blue, can be encrypted in many possible ways, depending on the choice of a *randomization value* selected at encryption time. Without this property, since most elections offer only a handful of choices, e.g. Blue or Red, an attacker could simply try to deterministically encrypt all possible choices to discover, by simple ciphertext comparison, how everyone voted. With the randomization value, an attacker would have to try all possible random factors. Selecting a cryptosystem with a large enough set of randomization values ensures that this is never possible.

Tallying under the Covers of Encryption. Using a special form of randomized public-key encryption called homomorphic public-key encryption, it is possible to combine two encryptions into a third encryption of a value related to the original two, i.e. the sum. For example, using only the public key, it is possible to take an encryption of x and an encryption of y and obtain an encryption of $x + y$, all without ever learning x or y or $x + y$.

In a homomorphic voting scheme, as first proposed by Benaloh [43, 19], votes are encrypted as either 0 for Blue or 1 for Red. Then, all resulting encrypted votes are homomorphically added, one at a time, to yield a single encryption of the number of votes for Red. The trustees can then decrypt this single encrypted value to discover this tally for Red. The difference between the total number of votes and the Red tally is then the Blue tally. The approach can be extended to more than two options by encoding multiple "counters" within a single ciphertext, a technique we review in chapter 2. In addition, a zero-knowledge proof is typically required for each submitted vote, in order to ensure that each vote is truly the encryption of 0 or 1, and not, for example, 1000. Otherwise, a malicious voter could easily throw off the count by a large amount with a single ballot.

Homomorphic voting is particularly interesting because the entire homomorphic operation is publicly verifiable by any observer, who can simply re-compute it on his own using only the public key. The trustees need only decrypt a single encrypted tally for each election race. Unfortunately, homomorphic voting does not support write-in votes well: the encrypted homomorphic counters must be assigned to candidates before the election begins.

Shaking the Virtual Ballot Box. A different form of tallying is achievable using a *mixnet*, as first described by Chaum [39]. Mixnets typically use a *rerandomizable encryption scheme*, which allows anyone to take an encryption of a message and produce a new encryption of the same message with altered randomness. This is particularly interesting because, if one were simply to take a set of encryptions and reorder them, it would be trivial to compare the shuffled encryptions to the pre-shuffling encryptions. As randomized encryptions are effectively unique, the rerandomization property is necessary to perform true, indistinguishable shuffling.

In a mixnet, a sequence of mix servers, each one usually operated by a different political party, takes all encrypted votes on the bulletin board, shuffles and rerandomizes them according to an order and a set of randomization values kept secret, and posts the resulting set of ciphertexts back to the bulletin board. The next mix server then performs a similar operation, and so on until the last mix server. Then, all trustees cooperate to decrypt the individual resulting encryptions, which have, by now, been dissociated from their corresponding voter identity.

It is reasonable to trust that at least one of the mix servers will "shake the ballot box well enough" so that privacy is ensured. However, it is *not reasonable* to trust that no single mix server will replace an encrypted vote in the mix with a vote of its own. In other words, we trust the mix servers with the privacy of the vote, but we do not trust them with its correctness. Thus, each mix server must provide a zero-knowledge proof that it performed correct mixing, never removing, introducing, or changing the underlying votes. These types of proof are rather complicated, but a number of efficient schemes are known and implemented.

Mixnet-based voting is more difficult to operate than homomorphic-based voting, because the re-encryption and shuffle processes must be executed on a trusted computing base, keeping the details of the shuffle secret from all others. However, mixnets present two important advantages: the complete set of ballots is preserved for election auditing, and free-form ballots, including write-ins, are supported. As a result, mixnet-based voting schemes offer the most promise in real-world, democratic election implementation, even if they are operationally more complex.

1.4.5 The Promise of Cryptographic Voting

With cryptographic voting promising significantly improved auditability, one might question why real elections have shunned these techniques to date. In fact, it is only recently that cryptographic voting schemes have become reasonably usable by average voters. Much necessary research remains to ensure that the extra voter verification processes can be made to work in a realistic setting. Most importantly, a significant education effort will be required, because the power of a cryptographically verified election is far from intuitive.

1.5 Contributions of this Work

This dissertation contributes to the teaching, practice, and theory of cryptographic voting. Each contribution attempts to make cryptographic voting more useful and realistic.

1.5.1 Mixnet Literature Review

One critical component of a large number of cryptographic voting schemes is the anonymous channel, usually implemented by a robust, universally verifiable mixnet. The literature in this field spans 25 years without a single literature review. This dissertation (Chapter 3) presents just such a review of advances in mixnet research over the years, starting with Chaum's first mixnet in 1981 through the most recent mixnet work of early 2006.

1.5.2 Scratch & Vote

We propose Scratch & Vote, a paper-based cryptographic voting protocol. Inspired by the ballots of Chaum and Ryan [40, 41], S&V presents three significant practical advantages:

1. ballots and the ballot casting process use simple technology easily deployable today,
2. tallying is homomorphic, requiring only a single threshold decryption per race, and
3. ballots are self-certifying, which makes auditing and ballot casting assurance more practical.

Scratch & Vote uses 2D barcode to encode the encrypted votes and a scratch surface that hides the randomization values used to encrypt these ciphertexts. Thus, a ballot can be audited by simply removing the scratch surface, without contacting any central authority. Only ballots that have *not* been scratched off can actually be used in the vote casting: each voter takes two ballots, audits one and votes with the other. With a 50% chance of catching an error with each voter, any moderate attempt at fraud will be caught with just a handful of voter verifications.

1.5.3 Ballot Casting Assurance & Assisted-Human Interactive Proofs

We have seen that cryptography provides universal verifiability of the tabulation process. In addition, cryptography gives Alice two important capabilities: she can *verify directly* that her vote has been correctly cast, and she can *realistically complain* if she discovers an error. One contribution of this dissertation is the exact definition of these properties, and the coining of the term *ballot casting assurance* to describe them. In particular, ballot casting assurance supplants the unfortunately-abused term "cast as intended," which has been used with differing meaning by various voting research communities.

We then present *Assisted-Human Interactive Proofs (AHIP)*, a definitional framework for interactive proofs where the verifier is significantly computationally limited—i.e. human. In the context of voting, this framework is particularly useful in describing how a voting machine proves to Alice that her vote was correctly encrypted. In particular, AHIP is the first definitional framework that captures the notion of a *secret voter receipt* and the security properties required of such a receipt.

We provide two implementations of an AHIP proof that a ciphertext encodes a specific option j out of m possible choices. The first protocol, a tweaked version of Neff's MarkPledge, provides ciphertexts whose length depends not only in the security parameter, but also in the desired soundness. The second protocol provides ciphertexts linear only in the security parameter, as long as soundness is within the range of the security parameter (it almost always is). Both schemes are particularly interesting because they are proven secure in this framework and demand very little from voters: they need only be able to compare very short strings—e.g. 4 characters.

1.5.4 Public Mixing

This dissertation also introduces *Public Mixing*, a new type of ballot-preserving anonymous channel. Typical mixnets await their inputs, then mix in private, then prove their mix. Public mixes prove their mix, then await their inputs, then mix through entirely public computation. In other words, all private operations and proofs in a public mix can be performed *before* the inputs are available. One should notice that this concept is by no means intuitively obvious. Public mixing creates, in a sense, the ability to obfuscate the program that performs the mixing. It is reasonable to think of public mixing as a combination of the advantages of homomorphic voting and mixnet voting: all election-day operations are public, yet all ballots are fully preserved.

We provide constructions of public mixing algorithms using the BGN [23] and Paillier [133] cryptosystems. The former case is simpler to explain but relies on a recently introduced cryptosystem with younger assumptions. The latter case is a significantly more complicated construction that relies on a much better understood set of assumptions and a cryptosystem that is well established in the literature.

Also covered in this work is the efficient distributed generation of a public mixer, because public mixing isn't composable. In the voting setting, it is crucial that no single party know the anonymization permutation. Thus, this efficient distributed generation ensures that multiple authorities collaborate to jointly create a permutation, which no small subset of the authorities can recover on its own.

1.5.5 Collaboration and Authorship

The mixnet review itself is the work of the author, though, of course, the individual schemes reviewed are the works of their respective authors. Scratch & Vote is joint work between the author and Ronald L. Rivest. Assisted-Human Interactive Proofs and Ballot Casting Assurance are joint work between the author and C. Andrew Neff. Public Mixing is joint work between the author and Douglas Wikström.

1.6 Organization

Chapter 2 reviews a number of cryptographic concepts important to voting protocols, including public-key cryptography, homomorphic cryptosystems, and threshold cryptosystems. Chapter 3 reviews the mixnet literature. Chapter 5 defines the concept of ballot casting assurance, and the Assisted-Human Interactive Proof model and implementations. Chapter 6 describes public mixing.

Chapter 2

Preliminaries

Protocols for democratic elections rely on numerous cryptographic building blocks. In this chapter, we review the concepts and notation of these building blocks. We begin with a review of public-key cryptography, its security definitions, and the principal algorithms that we use in practical protocols. We review homomorphic cryptosystems, the interesting properties they yield, and the security consequences of these properties. Then, we consider threshold cryptosystems, where the process of key generation and decryption can be distributed among trustees, a task of great importance to voting systems. We also review zero-knowledge proofs, another critical component of universally verifiable voting, and we briefly review program obfuscation, which is of particular importance to the contributions of this dissertation. We also cover universal composability, a framework for proving protocols secure that has become quite useful in the recent voting literature.

The last section of this chapter reviews prior work in universally verifiable cryptographic voting, including mixnet-based systems and homomorphic aggregation. It defines notation that cuts across the various schemes. It is this notation which we will use in the rest of this work.

2.1 Basics

We denote by κ the main security parameter and say that a function $\epsilon(\cdot)$ is negligible if for every constant c there exists a constant κ_0 such that $\epsilon(\kappa) < \kappa^{-c}$ for $\kappa > \kappa_0$. We denote by PT, *PPT*, and PT*, the set of uniform polynomial time, probabilistic uniform polynomial time, and non-uniform polynomial time Turing machines respectively. We use the notation $\xleftarrow{R}$ to denote either a uniform random sampling from a set or distribution, or the assignment from a randomized process, i.e. a *PPT* algorithm, with uniform choice of randomization values.

2.2 Public-Key Encryption

Public-key encryption was first suggested by Diffie and Helman [56] in 1976, and first implemented by Rivest, Shamir, and Adleman [145] in 1977. At its core, it is a simple, though somewhat counter-intuitive, idea: anyone can encrypt a message destined for Alice, but only Alice can decrypt it. More precisely, Alice can generate a keypair composed of a public key pk and a secret key sk. She then distributes pk widely, but keeps sk to herself. Using pk, Bob can encrypt a plaintext m into a ciphertext c. The ciphertext c is then effectively "destined" for Alice, since only Alice possesses sk, with which she can decrypt c back into m.

More formally, we can define a public-key cryptosystem as follows.

Definition 2-1 (Public-Key Cryptosystem) *A public-key cryptosystem* PKCS *is a set of three PPT algorithms $\mathcal{G}, \mathcal{E}, \mathcal{D}$, such that, given security parameter κ, the following operations are defined:*

- **Keypair Generation:** *matching public and secret keys can be generated by anyone using the public algorithm $\mathcal{G}$.*

$$(pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa)$$

- **Encryption:** *a plaintext m in the message space M_{pk} can be encrypted using the public key pk and the encryption algorithm $\mathcal{E}$. This process is usually randomized, using randomization value $r \in \mathsf{R}_{pk}$*

$$c = \mathcal{E}_{pk}(m; r)$$

We denote C_{pk} the space of ciphertexts c.

- **Decryption:** *a ciphertext c in the ciphertext space C_{pk} can be decrypted using the secret key sk and the decryption algorithm $\mathcal{D}$. This process is always deterministic: a given ciphertext always decrypts to the same plaintext under a given secret key.*

$$m = \mathcal{D}_{sk}(c)$$

Given such a cryptosystem, one can consider different security definitions.

2.2.1 IND-CPA Security

Intuitively, a cryptosystem is said to be *semantically secure* if, given a ciphertext c, an adversary cannot determine any property of the underlying plaintext m. In other words, an adversary cannot extract any semantic information of plaintext m from an encryption of m. Semantic security was first defined in 1982 by Goldwasser and Micali [80], who also showed that semantic security is equivalent to *ciphertext indistinguishability with chosen plaintexts* [81]. This latter definition, known as *GM Security* or *IND-CPA* , is a more natural one, so we state it here.

In this definition, given a public key pk, the adversary chooses two plaintexts m_0 and m_1 and is then presented with c, a ciphertext of one of these plaintexts, chosen at random. If the adversary cannot guess which of the two plaintexts was chosen for encryption with noticeably better than 50% chance (i.e. picking one at random), then the scheme is said to be secure against chosen plaintext attack.

Definition 2-2 (IND-CPA Security) *A public-key cryptosystem* $\mathsf{PKCS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is said to be* IND-CPA*-secure if there exists a negligible function* $\nu(\cdot)$ *such that, for all* $\mathsf{Adv} \in \mathrm{PT}^*$*:*

$$\begin{aligned} \Pr\Big[(pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa); (m_0, m_1, \mathsf{state}) \leftarrow \mathsf{Adv}(\mathsf{choose}, pk) &\ ; \\ b \xleftarrow{R} \{0,1\}; c \xleftarrow{R} \mathcal{E}_{pk}(m_b); b' \leftarrow \mathsf{Adv}(\mathsf{guess}, c, \mathsf{state}) &\ : \\ b = b' &\Big] < \frac{1}{2} + \nu(\kappa) \end{aligned}$$

We know of a number of efficient schemes that are IND-CPA-secure.

RSA with OAEP Padding. In so-called "raw RSA" [145], two safe primes p and q are selected, $pk = (n, e)$ where $n = pq$ and $e \not| \phi(n)$, and $sk = d$ where $ed = 1 \bmod \phi(n)$. Encryption is then performed as $c = m^e \bmod n$, and decryption as $m = c^d \bmod n$. Clearly, since the encryption operation is deterministic given m and pk, raw RSA is *not* IND-CPA-secure: an adversary can just encrypt m_0 and m_1 and compare them against the challenge ciphertext.

RSA can be made IND-CPA-secure using message padding such as OAEP [16]. Instead of encrypting the raw message m, RSA-OAEP encrypts $m||OAEP(m)$, where $OAEP(m)$ includes randomness.

El Gamal. El Gamal [71] is the prime example of an IND-CPA-secure cryptosystem. Consider g the generator of a q-order subgroup of $\mathbf{Z}_p^*$, where p is prime and q is a large prime factor of $p - 1$. Key generation involves selecting a random $x \in \mathbf{Z}_q^*$, at which point $sk = x$ and $pk = y = g^x \bmod p$. Encryption is then given as

$$c = (\alpha, \beta) = (g^r, m \cdot y^r), \quad r \xleftarrow{R} \mathbf{Z}_q^*.$$

Decryption is performed as

$$m = \frac{\beta}{\alpha^x}$$

Paillier. Paillier [133] is another good example of an IND-CPA-secure cryptosystem. Consider $n = pq$ as in the RSA setting. Consider $\lambda = lcm(p-1, q-1)$. Consider the function $L(x) = (x-1)/n$. Consider a generator g of $\mathbf{Z}_{n^2}^*$ specially formed such that $g = 1 \bmod n$. The public key is then simply n, while the private key is λ. Encryption of $m \in \mathbf{Z}_n$ is performed as $c = g^m r^n \bmod n^2$ for a random $r \in \mathbf{Z}_n^*$. Decryption is performed as

$$m = \frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$$

We provide here a brief explanation of the Paillier cryptosystem, given that it is particularly interesting and useful for our work in this dissertation. Recall that:

- $\phi(n) = (p-1)(q-1)$ is Euler's totient function.
- $\lambda = lcm(p-1, q-1)$ is the output of Carmichael's function on n.
- The order of $\mathbf{Z}_{n^2}^*$ is $n\phi(n)$.
- For any $a \in \mathbf{Z}_{n^2}^*$:
  - $a^\lambda = 1 \bmod n$
  - $a^{\lambda n} = 1 \bmod n^2$

Thus, consider the decryption function defined above, in particular the denominator. Recall that $g = 1 \bmod n$, which we can also write $g = n\alpha + 1$ for some integer α.

$$\begin{aligned} L(g^\lambda \bmod n^2) &= \frac{((1+n\alpha)^\lambda \bmod n^2) - 1}{n} \\ &= \frac{(n\alpha\lambda) \bmod n^2}{n} \\ &= \alpha\lambda \bmod n^2 \end{aligned}$$

Note that the exponentiation above reduces to the multiplication because all other monomials in the expansion are multiples of n^2. One can then easily see that, because r^n will cancel out by exponentiation to λ:

$$L(c^\lambda \bmod n^2) = m\alpha\lambda \bmod n^2$$

and thus that the decryption works as specified.

2.2.2 IND-CCA Security

Indistinguishability with respect to adversarially-chosen plaintexts is not enough for all applications. Intuitively, one should also consider the possibility that the adversary can obtain the decryption of a few chosen ciphertexts before receiving the challenge ciphertext. This notion of security is called IND-CCA-security, informally known as "security against lunchtime attacks." The model is that the adversary might have access to a decryption box while the owner is "out to lunch" (possibly metaphorically.) Later, the adversary will try to use the information gained over lunch to decrypt other ciphertexts.

Definition 2-3 (IND-CCA Security) *A public-key cryptosystem* PKCS $= (\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is said to be IND-CCA-secure if there exists a negligible function* $\nu(\cdot)$ *such that, for all* Adv $\in$ PT*, *given a decryption oracle* ODec$(\cdot)$*:*

$$\mathsf{Pr}\Big[(pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa); (m_0, m_1, \mathsf{state}) \leftarrow \mathsf{Adv}^{\mathsf{ODec}_{sk}(\cdot)}(\mathsf{choose}, pk) \quad ; \\ b \xleftarrow{R} \{0,1\}; c \xleftarrow{R} \mathcal{E}_{pk}(m_b); b' \leftarrow \mathsf{Adv}(\mathsf{guess}, c, \mathsf{state}) \quad : \\ b = b' \Big] < \frac{1}{2} + \nu(\kappa)$$

As it turns out, the notion of IND-CCA security is not as interesting as its more powerful variant, IND-CCA2 security.

2.2.3 IND-CCA2 Security

The IND-CCA2 security definition gives the lunchtime attacker even more power: after the challenge ciphertext has been issued, the attacker gets further access to the decryption oracle ODec$(\cdot)$, which the attacker can query for anything, except of course the challenge ciphertext itself. Informally, this implies that an attacker is unable to extract any information about a ciphertext, even if he's able to request the decryption of any other ciphertext, even ones derived from the challenge ciphertext.

Definition 2-4 (IND-CCA2 Security) *A public-key cryptosystem* PKCS $= (\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is said to be IND-CCA2-secure if there exists a negligible function* $\nu(\cdot)$ *such that, for all* Adv $\in$ PT*, *given a decryption oracle* ODec$(\cdot)$*:*

$$\mathsf{Pr}\Big[(pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa); (m_0, m_1) \leftarrow \mathsf{Adv}^{\mathsf{ODec}_{sk}(\cdot)}(\mathsf{choose}, pk) \quad ; \\ b \xleftarrow{R} \{0,1\}; c \xleftarrow{R} \mathcal{E}_{pk}(m_b); b' \leftarrow \mathsf{Adv}^{\mathsf{ODec}_{sk,c}(\cdot)}(\mathsf{guess}, c) \quad : \\ b = b' \Big] < \frac{1}{2} + \nu(k)$$

where ODec$_{sk,c}(\cdot)$ *is a decryption oracle which answers all queries by decrypting the requested ciphertext, except for the challenge ciphertext c where it answers with NULL.*

IND-CCA2 security is considered the "gold standard" of public-key cryptosystems (though, in some cases, the alternate standard of plaintext-awareness [89] is also considered.) Effectively, the only known way to obtain a fresh ciphertext for a given message is to encrypt the plaintext yourself.

2.2.4 IND-RCCA Security

As it turns out, IND-CCA2 security may be overkill for a number of applications. In particular, given a ciphertext c, it might be acceptable to let anyone create a new ciphertext c' such that $\mathcal{D}_{sk}(c) = \mathcal{D}_{sk}(c')$. Notably, someone without the secret sk would still be unable to generate a ciphertext whose plaintext is related to that of c in some way other than equality.

Thus, there is a middle ground between IND-CPA and IND-CCA security: IND-RCCA security [142], which specifically allows an adversary to generate a "fresh" ciphertext c' from an existing ciphertext c, such that $\mathcal{D}_{sk}(c) = \mathcal{D}_{sk}(c')$. More formally:

Definition 2-5 (IND-RCCA Security) *A public-key cryptosystem* PKCS $= (\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is said to be IND-RCCA-secure if there exists a negligible function $\nu(\cdot)$ such that, for all* Adv $\in$ PT*, *given a decryption oracle* $\mathsf{ODec}(\cdot)$*:*

$$\Pr\Big[(pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa); (m_0, m_1) \leftarrow \mathsf{Adv}^{\mathsf{ODec}_{sk}(\cdot)}(\mathsf{choose}, pk) \quad ;$$
$$b \xleftarrow{R} \{0,1\}; c \xleftarrow{R} \mathcal{E}_{pk}(m_b); b' \leftarrow \mathsf{Adv}^{\mathsf{ODec}_{sk,m_0,m_1}(\cdot)}(\mathsf{guess}, c) \quad :$$
$$b = b' \quad \Big] < \frac{1}{2} + \nu(\kappa)$$

where $\mathsf{ODec}_{sk,m_0,m_1}$ *is a decryption oracle which answers all queries by decrypting the requested ciphertext, except for ciphertexts which decrypt to either m_0 or m_1.*

2.3 Homomorphic Public-Key Encryption

Homomorphic public-key cryptosystems exhibit a particularly interesting algebraic property: when two ciphertexts are combined in a specific, publicly-computable fashion, the resulting ciphertext encodes the combination of the underlying plaintexts under a specific group operation, usually multiplication or addition.

Definition 2-6 (Homomorphic Cryptosystem) *A public-key cryptosystem* PKCS $= (\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is said to be homomorphic for binary relations $(\oplus, \otimes)$ if:*

- $\forall (pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa)$, *given message domain* M_{pk}, $(\mathsf{M}_{pk}, \oplus)$ *forms a group.*
- $\forall (pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa)$, *given ciphertext range* C_{pk}, $(\mathsf{C}_{pk}, \otimes)$ *forms a group.*
- $\forall (pk, sk) \xleftarrow{R} \mathcal{G}(1^\kappa), \forall (c_1, c_2) \in \mathsf{C}^2_{\mathsf{PKCS},pk}$,

$$\mathcal{D}_{sk}(c_1 \otimes c_2) = \mathcal{D}_{sk}(c_1) \oplus \mathcal{D}_{sk}(c_2).$$

2.3.1 Re-encryption

An immediate consequence of a cryptosystem's homomorphic property is its ability to perform reencryption: given a ciphertext c, anyone can create a different ciphertext c' that encodes the same plaintext as c. Recall that PKCS is homomorphic for $(\oplus, \otimes)$ if $(\mathsf{M}_{pk}, \oplus)$ forms a group, which means there exists an identity plaintext m_0 such that, $\forall m \in \mathsf{M}_{pk}, m \oplus m_0 = m$. Thus, given a homomorphic cryptosystem PKCS, we can define the reencryption algorithm as follows:

$$\mathcal{RE}_{pk}(c; r) = c \otimes \mathcal{E}_{pk}(m_0; r)$$

If $\mathcal{D}_{sk}(c) = m$, then $\mathcal{D}_{sk}(\mathcal{RE}_{pk}(c)) = m$, too.

2.3.2 Security of Homomorphic Cryptosystems

The malleability of ciphertexts in homomorphic cryptosystems limits the security of such schemes. In particular, the ability to reencrypt immediately indicates that the system is not IND-CCA2-secure, and can be at best IND-RCCA-secure. Even more significant, the ability to create a ciphertext of a *related but different* plaintext breaks even IND-RCCA security. Specifically, an adversary can take the challenge ciphertext c, create $c' = c \otimes \mathcal{E}_{pk}(\tilde{m})$ for some $\tilde{m}$ known to the adversary, query ODec with c' to obtain m', and compute $m = m' \oplus \tilde{m}^{-1}$.

It is not well understood whether homomorphic schemes can be IND-CCA-secure: can ODec help an adversary succeed if it can only be used *prior* to the challenge ciphertext? Thus, we know that homomorphic cryptosystems can be IND-CPA-secure, but we do not know whether they can be IND-CCA-secure.

2.3.3 Homomorphic Schemes in Practice

A number of practical schemes are homomorphic.

RSA. In raw RSA, encryption is performed as $c = m^e \bmod n$. Thus, clearly, $c_0 \times c_1 = (m_0 \times m_1)^e \bmod n$. Raw RSA is thus homomorphic on operations $(\times, \times)$. That said, raw RSA isn't even IND-CPA-secure, which means it isn't very useful in many applications. RSA-OAEP, on the other hand, is quite useful, but loses the homomorphic property due to the non-malleable OAEP padding.

El Gamal. In El Gamal, encryption is performed as $c = (g^r, m \cdot y^r)$. Thus, if we define $\otimes$ as the element-wise product of ciphertext pairs, then El Gamal is homomorphic for $(\times, \otimes)$:

$$(g^{r_1}, m_1 \cdot y^{r_1}) \otimes (g^{r_2}, m_2 \cdot y^{r_2}) = (g^{r_1+r_2}, (m_1 m_2) \cdot y^{r_1+r_2}).$$

El Gamal is particularly interesting: it exhibits a homomorphism and is IND-CPA-secure.

Exponential El Gamal. If El Gamal is homomorphic for multiplication with a plaintext in the base, then one is immediately tempted to adapt El Gamal to use a plaintext in the exponent in order to exhibit a homomorphic addition. In fact, this can be done, but at a large cost: decryption requires performing a discrete log, which inherently limits the plaintext domain M to polynomial size. Exponential El Gamal is defined as :

- **Key Generation:** same as El Gamal, select a prime p such that another large prime q divides $(p-1)$. Select g, a generator of a q-order subgroup of $\mathbf{Z}_p^*$. $\mathsf{M}_{pk} = \mathbf{Z}_q$. $sk = x$, where x is randomly selected in $\mathbf{Z}_q^*$. $pk = y = g^x \bmod p$.
- **Encryption:** similar to El Gamal, except the plaintext is now in the exponent.

$$\mathcal{E}_{pk}(m; r) = (\alpha, \beta) = (g^r, g^m y^r) \bmod p.$$

- **Decryption:** similar to El Gamal, except a discrete logarithm is now required in addition to the usual computation.

$$\mathcal{D}_{sk}(\alpha, \beta) = \log_g \left[\frac{\beta}{\alpha^x}\right] \bmod p.$$

- **Homomorphic Addition:** exactly the same as El Gamal's homomorphic multiplication, using a ciphertext operation which performs element-wise multiplication on the ciphertext pairs:

$$\begin{aligned}
\mathcal{E}_{pk}(m_1; r_1) \otimes \mathcal{E}_{pk}(m_2; r_2) &= (g^{r_1}, g^{m_1} y^{r_1}) \otimes (g^{r_2}, g^{m_2} y^{r_2}) \\
&= (g^{r_1+r_2}, g^{m_1+m_2} y^{r_1+r_2}) \\
&= \mathcal{E}_{pk}(m_1 + m_2; r_1 + r_2).
\end{aligned}$$

In practice, the decryption limits the message domain, e.g. a few trillion possible messages at the most. Decryption is usually performed by various discrete-logarithm algorithms, for example the baby-step giant-step algorithm [160], which requires $O(\sqrt{m})$ time.

Paillier. In Paillier, encryption is performed as $c = g^m r^n \bmod n^2$. Clearly, this scheme is homomorphic for $(+, \times)$ over the plaintext space $\mathbf{Z}_n$:

$$\begin{aligned}
\mathcal{E}_{pk}(m_1; r_1) \times \mathcal{E}_{pk}(m_2; r_2) &= (g^{m_1} r_1^n) \times (g^{m_2} r_2^n) \\
&= g^{m_1+m_2} (r_1 r_2)^n \\
&= \mathcal{E}_{pk}(m_1 + m_2; r_1 r_2).
\end{aligned}$$

Note that Paillier decryption is efficient, which means the plaintext domain can be superpolynomial while retaining the additive homomorphism.

Generalized Paillier. Damgård et al. [48] generalize the Paillier scheme, so that a public key with modulus n can encrypt plaintexts in $\mathbf{Z}_{n^s}$ into ciphertexts in $\mathbf{Z}_{n^{s+1}}$. Computations modulo n^2 are replaced by computations modulo n^{s+1}. For this generalized version, we write

$$\mathcal{E}^{\mathsf{pai}}_{n,s}(m) = g^m r^{n^s} \bmod n^{s+1}$$

and we use M_{n^s} and C_{n^s} to denote the corresponding message space $\mathbf{Z}_{n^s}$ and ciphertext space $\mathbf{Z}^*_{n^{(s+1)}}$. Damgård et al. prove that the security of the generalized scheme follows from the security of the original scheme. The properties of this extended Paillier cryptosystem can be seen as parallel to those of the typical Paillier cryptosystem:

- the order of the group $\mathbf{Z}_{n^{(s+1)}}$ is $\phi(n)n^s$.
- the order of $(n+1)$ is n^s in $\mathbf{Z}_{n^{(s+1)}}$, thus we will use $g = n+1$ as the base for encryption: its exponent is perfectly sized for plaintexts in $\mathbf{Z}_{n^s}$.
- If we denote $s = r^{(n^s)}$, effectively the randomization value, then: $s^{\phi(n)} = 1 \bmod n^{(s+1)}$ and $s^\lambda = 1 \bmod n^{(s+1)}$. Thus, when computing c^λ during decryption, the randomization value cancels out, and we get:

$$c^\lambda = g^{m\lambda} \bmod n^{(s+1)}$$

Generalized Paillier provides two interesting high-level properties:

1. *longer plaintexts with better efficiency*: using the same public key, n, plaintexts longer than $|n|$ can be encrypted, while the ciphertext overhead remains $|n|$.
2. *layered encryption*: plaintexts can be encrypted multiple times under the same public key, each time adding an overhead of $|n|$ bits. This is interesting in that the group orders maintains all homomorphic properties intact at every layer. Note that this property has not been noticed before, and is described for the first time in Chapter 6.

2.4 Threshold Public-Key Cryptosystems

In many applications, including notably voting, it is desirable to allow decryption only when a quorum of "trustees" agree. In other words, the secret key sk isn't available to a single party. Instead, l trustees share sk: trustee i has share $sk^{(i)}$. If at least k of the l trustees participate, then decryption is enabled. If fewer than k trustees participate, then the security properties of the cryptosystem are fully preserved.

There are two conceivable approaches to generating the shares $\{sk^{(i)}\}$. The simpler approach is for a "dealer" to generate (pk, sk) normally, split sk into shares, then distribute these shares to the appropriate trustees. A more secure approach is to have the trustees generate the keypair together, with no single party ever learning the complete sk in the process.

2.4.1 Secret Sharing

Critical to the implementation of threshold cryptography is the concept of secret sharing, first introduced by Shamir in 1979 [159]. In this scheme, a secret s in a finite field is shared as $s^{(1)}, s^{(2)}, \dots, s^{(l)}$, where any k-sized subset of these n shares reveals s $(k \leq l)$, but any subset of size smaller than k reveals nothing about s. Shamir's implementation produces a polynomial $P(x)$ of degree $k-1$ over the finite field in question, such that $P(0) = s$, and each share $s^{(i)}$ is a point (x, y) such that $y = P(x)$ (and $x \neq 0$, of course). Using Lagrange coefficients for polynomial interpolation, k points are sufficient to recover the polynomial P, and thus $s = P(0)$. Fewer than k points, however, will hide s information-theoretically.

Recall that this interpolation process defines Lagrange interpolation polynomials for each point. Given $\{(x_i), (y_i)\}_{i \in [1,k]}$ the k points we wish to interpolate, we denote $\lambda_i(x)$ the interpolation polynomial that corresponds to point i:

$$\lambda_i(x) = \prod_{j=1, j\neq i}^{k} \frac{(x - x_j)}{(x_i - x_j)}$$

The interpolated polynomial is then:

$$P(x) = \sum_{i=1}^{k} \lambda_i(x) y_i$$

Since we seek only $P(0)$, the secret, we can skip the computation of the actual polynomial coefficients, and go straight to:

$$s = P(0) = \sum_{i=1}^{k} y_i \left(\prod_{j=1, j\neq i}^{k} \frac{-x_j}{(x_i - x_j)} \right)$$

Note that each share is the *pair* (x, y). Thus, it is permissible for all x_i to be public, with the corresponding y_i remaining secret. This allows anyone to compute the Lagrange coefficients, ready to be combined with the actual y_i values at the appropriate time.

2.4.2 Secure Multi-Party Computation

In 1986, Yao [182] showed that any multi-party computation can be performed using a garbled representation of the circuit that implements the computation. Yao's technique involves a gate-by-gate, bit-by-bit decomposition of the computation. Thus, though incredibly powerful as a generic method, it it is quite inefficient in practice.

Clearly, threshold cryptosystems can be implemented using a simple Shamir secret sharing scheme and two garbled circuits: one that generates, splits, and distributes the keypair (pk, sk) to all trustees, and one that combines the shares to perform actual decryption. In practice, however, it is best to find a cryptosystem that explicitly supports some efficient mechanism for threshold operations.

2.4.3 Efficient Threshold Schemes

El Gamal. The algebraic structure of the El Gamal cryptosystem makes it particularly useful for threshold operations, as first described in this setting by Desmedt and Frankel [53]. Consider sharing the secret key x into shares $x_1, x_2, \dots, x_l$, using Shamir k-out-of-l secret sharing. Each share x_i is associated with a public-key share, $y_i = g^{x_i}$. Note that x and y here should not be confused with the coordinates of the points in the Shamir secret-sharing scheme: they are the private and public key, respectively.

Decryption in this setting then becomes:

$$m = \frac{\beta}{\prod \alpha^{x_i \lambda_i(0)}}$$

where α^{x_i} can be computed by each trustee independently.

Most importantly, it is then possible to achieve distributed generation of a key, where no single party, not even a setup "dealer", learns the complete secret key, as shown by Pedersen [135]. The protocol functions as follows, at a high level:

1. each trustee generates secret share $x_i \in \mathbf{Z}_q^*$, and publishes $y_i = g^{x_i}$.

2. each trustee secret-shares x_i to all other participants using a k-out-of-l verifiable secret sharing scheme, so that cheating parties can be caught.

3. every k subset of the trustees can then perform the lk operations to decrypt an El Gamal ciphertext with two layers of threshold actions, one layer to reconstitute the actions of each x_i, and another layer to reconstitute the actions of the x_i's into the actions of the overall secret x.

Note that numerous variants of the Pedersen protocol have been published, and that certain subtle weaknesses were revealed and fixed by Gennaro et al. in 1999 [74].

RSA. Obtaining threshold properties from RSA is significantly more challenging, in particular with respect to key generation, which requires that the product of two primes be obtained without any single party knowing these two primes. Efficient schemes are known [54, 155, 73, 161] for threshold decryption. In addition, schemes more efficient than generic multi-party computation are also known for key generation [25, 69, 49], though they remain quite a bit slower than those for discrete-log-based systems like El Gamal. At a high level, these various protocols define the following operations:

- **decryption**: much like the El Gamal setting, the general idea of RSA threshold decryption is to share a secret exponent, in this case the private exponent d, then use Lagrange interpolation to recombine k out of l of these. Earlier work assumed this interpolation was too difficult to perform over $\mathbf{Z}_n^*$, since the order of the group is unknown, and chose to perform polynomial interpolation over a subgroup of $\mathbf{Z}_n^*$. Shoup [161] showed that, if the primes factors of the RSA modulus are "safe primes"—e.g. each is twice another prime plus 1—then polynomial interpolation is, in fact, possible, over $\mathbf{Z}_n^*$.

- **key generation**: Boneh and Franklin [25] first implemented a relatively efficient scheme for the distributed generation of RSA keys, such that no one party learns the factorization: each trustee obtains a share of the secret exponent d, at which point threshold decryption is possible as described above. Frankel et al. [69] improved this result by making the generation process resistant against active attack. Damgård and Koprowski [49] combined the efficiency of the Shoup scheme and the distributed nature of the Frankel et al. scheme.

Paillier. Paillier uses number theory quite similar to RSA, but the decryption process is not quite the same. Fouque et al. [68] show how to apply Shoup's method of RSA threshold decryption to Paillier. Damgård and Jurik [48] show a related method that also applies to their generalized version of Paillier. Damgård and Koprowski [49] show how their method for distributed RSA key generation can also be applied to Paillier.

2.5 Zero-Knowledge Proofs

A major component of verifiable voting protocols is the zero-knowledge proof. In a zero-knowledge proof, a prover $\mathcal{P}$ interacts with a verifier $\mathcal{V}$ to demonstrate the validity of an assertion, e.g. "ciphertext c under public key pk decrypts to 'Mickey Mouse'." If the prover is honest—i.e. the assertion is true—then the verifier should accept this proof. If the prover is dishonest—i.e. the assertion is false—then the verifier should reject this proof with noticeable probability. Finally, the verifier should learn nothing more than the truth of the assertion. In particular, the verifier should be unable to turn around and perform this same (or similar) proof to a third party.

The notion of "zero-knowledge" is tricky to define: how can one capture the concept that *no* knowledge has been transfered? The accepted approach is to look at the verifier and see if its participation in the proof protocol bequeathed it any new capability. The protocol is zero-knowledge if, no matter what the verifier outputs after the protocol, it could have produced the very same output without interacting with the prover. Thus, though the verifier may be personally convinced from its interaction that the prover's assertion is indeed true, the verifier is unable to relay any new information convincingly, in particular he cannot perform the proof on his own.

The prover's assertion is formally defined as "x is in language $\mathcal{L}$," where x is a string, and $\mathcal{L}$ is a language, usually an NP language. Thus, the prover $\mathcal{P}$ is given x and a witness w for x such that $\mathcal{R}_{\mathcal{L}}(x, w) = 1$, where $\mathcal{R}_{\mathcal{L}}$ is the binary relation for language $\mathcal{L}$. The verifier $\mathcal{V}$ only gets x as input, of course. The zero-knowledge property of the protocol ensures that the witness w, and in fact any non-trivial function of the witness, remains hidden from $\mathcal{V}$.

Definition 2-7 (Perfect Zero-Knowledge Proof) *An interactive protocol $\langle \mathcal{P}, \mathcal{V} \rangle$ for language $\mathcal{L}$ is defined as a* perfect zero-knowledge proof *if there exists a negligible function $\nu(\cdot)$ such that the protocol has the following properties:*

- **Completeness:** $\forall x \in \mathcal{L}, \mathsf{Pr}\Big[\mathsf{output}_{\mathcal{V}}\big\langle \mathcal{P}(x, w), \mathcal{V}(x) \big\rangle = 1\Big] > 1 - \nu(k)$.

- **Soundness:** $\forall \mathcal{P}^*, \forall x \notin \mathcal{L}, \mathsf{Pr}\Big[\mathsf{output}_{\mathcal{V}}\big\langle \mathcal{P}^*(x), \mathcal{V}(x) \big\rangle = 1\Big] < \frac{1}{2}$

- **Zero-Knowledge:** $\exists PPT\ \mathcal{S}, \forall \mathcal{V}^*, \forall x \in \mathcal{L}, \mathcal{S}(x) \simeq \mathsf{output}_{\mathcal{V}^*}\langle \mathcal{P}(x, w), \mathcal{V}^*(x) \rangle$

2.5.1 Zero-Knowledge Variants

A few variants of this definition exist:

- **Computational Zero-Knowledge (CZK):** the verifier $\mathcal{V}$, and thus the dishonest version $\mathcal{V}^*$, are probabilistic polynomial-time. In other words, a surprisingly powerful verifier might be able to extract some knowledge from an execution of a CZK protocol.

- **Zero-Knowledge Argument**: the prover $\mathcal{P}$ is assumed to be computationally constrained, i.e. it is a *PPT ITM*. This setting must be considered with care, as the *PPT* limitation is dependent on the security parameter κ, but $\mathcal{P}$ may spend significant time preparing for the protocol execution.

- **Honest-Verifier Zero-Knowledge (HVZK)**: the verifier $\mathcal{V}$ is expected to perform according to the protocol. In particular, as the verifier is usually expected to submit a random challenge to the prover, an honest verifier will always flip coins when picking a challenge and will never base his challenge on the prover's messages. The result of an HVZK assumption is that the simulation proof can focus on simulating a transcript of the interaction, rather than simulating anything $\mathcal{V}$ could output. An HVZK protocol can be turned into a non-interactive zero-knowledge (NIZK) proof using the Fiat-Shamir heuristic [65], where the verifier's random messages are generated using a hash function applied to the prior protocol messages. This hash function must be modeled as random oracle, which has recently caused some concern in the theoretical cryptography community [77].

Zero-knowledge proofs play a big role in verifiable voting protocols, where each player must prove that it performed its designated action correctly while preserving voter privacy. As the integrity of the vote takes precedence over the voter's privacy, it can be immediately said that computational zero-knowledge proofs will be preferable to zero-knowledge arguments. We will explore the details of these issues in Chapter 3.

2.5.2 Proofs of Knowledge

Certain zero-knowledge proofs provide an additional property that is particularly useful in proving overall protocol security: they prove *knowledge* of the witness, not just existence. In particular, this means that, given rewindable, black-box access to the prover program $\mathcal{P}$, one can extract a witness w to the assertion that $x \in \mathcal{L}$. More formally, we define a *zero-knowledge proof of knowledge* as follows.

Definition 2-8 (Zero-Knowledge Proof of Knowledge) *An interactive protocol* $\langle\mathcal{P}, \mathcal{V}\rangle$ *for language* $\mathcal{L}$ *is defined as a* zero-knowledge proof of knowledge *if the protocol is zero-knowledge and it has the following, additional property:*

- **Extraction:** $\exists$*PPT ITM* $\mathcal{E}, \forall(x, w) \in \mathcal{R}_{\mathcal{L}}, \mathcal{E}^{\mathcal{P}(x,w)}() = w$. *By* $\mathcal{E}^{\mathcal{P}(x,w)}$, *we mean that we take the prover program* $\mathcal{P}$, *provide it with inputs* (x, w), *and give the extractor* $\mathcal{E}$ *black-box access to this initialized prover program, allowing the extractor to rewind, reply, and provide continuing inputs to* $\mathcal{P}$.

A proof-of-knowledge protocol can be particularly useful in the context of reduction proofs, since the extraction property allows a simulator to get the witness and use it in the reduction process. A zero-knowledge proof without extractability is much more difficult to integrate into a complete protocol security proof.

2.6 Program Obfuscation

Program obfuscation is a functionality-preserving transformation of a program's source code such that it yields no more information than black-box access to the original program. Uses of obfuscation in practice include tasks such as "digital rights management" programs. From a cryptographic standpoint, obfuscation presents enticing features in the realm of delegation: Alice can delegate to Bob the ability to use her secret key *for certain purposes only*, like that of decrypting emails with a particular subject.

Generalized program obfuscation, though it would be fantastically useful in practice, has been proven impossible in even the weakest of settings [12, 78]. To date, point functions are the only specific class of programs known to be obfuscatable [31, 173].

2.6.1 First Formalization

Program obfuscation was first formalized by Barak et al. [12], who also proved that there exists a special class of functions that cannot be obfuscated under this definition. Informally, an obfuscator transforms a program's source code so that:

1. obfuscation preserves the program's functionality,
2. the program's size expands no more than polynomially,
3. the obfuscated source code yields no more information than black box access to the original program.

We begin by formalizing a family of programs, then we define the obfuscation of such a family.

Definition 2-9 (Program Class) *A program class is a family $\{\mathbb{P}_\kappa\}_{\kappa\in\mathbb{N}}$ of sets of programs such that there exists a polynomial $s(\cdot)$ such that $|\mathcal{P}| \leq s(\kappa)$ for every $\mathcal{P} \in \mathbb{P}_\kappa$. The program class is said to be PPT if, for every $\kappa \in \mathbb{N}$, for every $\mathcal{P} \in \mathbb{P}_\kappa$, $\mathcal{P}$ runs in probabilistic polynomial time in κ.*

Definition 2-10 (Program Obfuscation) *An algorithm $\mathcal{O}$ is an obfuscator for a class of programs $\{\mathbb{P}_\kappa\}_{\kappa\in\mathbb{N}}$ if*

1. *for every $\kappa \in \mathbb{N}$ and for every $\mathcal{P} \in \mathbb{P}_\kappa$, $\mathcal{O}(\mathcal{P})$ computes the same function as $\mathcal{P}$,*

2. *there is a polynomial $l(\cdot)$ such that, for every $\kappa \in \mathbb{N}$ and for every $\mathcal{P} \in \mathbb{P}_\kappa$, $|\mathcal{O}(\mathcal{P})| \leq l(|\mathcal{P}|)$,*

3. *for every adversary* $\mathsf{Adv} \in \mathrm{PT}^*$ that outputs a single bit *(i.e. a predicate), there exists a simulator $\mathcal{S} \in PPT$ and a negligible function $\nu(\cdot)$ such that, for every $\kappa \in \mathbb{N}$, for every circuit $\mathcal{P} \in \mathbb{P}_\kappa$*

$$\left|\mathsf{Pr}\left[\mathsf{Adv}(\mathcal{O}(\mathcal{P})) = 1\right] - \mathsf{Pr}\left[\mathcal{S}^{\mathcal{P}}(1^\kappa) = 1\right]\right| < \nu(\kappa) \ .$$

where $\mathcal{S}^{\mathcal{P}}$ denotes the algorithm $\mathcal{S}$ with black-box access to $\mathcal{P}$.

2.6.2 Auxiliary Inputs

In 2005, Tauman-Kalai and Goldwasser [78] showed that the Barak definition is insufficient to cover many typical use cases of obfuscation. In particular, one must consider that the adversary has access to additional inputs beyond just the program's source code. In particular, what happens if the adversary receives two obfuscations of the same program? Or two obfuscations of different programs? Both cases must be considered: obfuscation in the presence of *independent* auxiliary inputs, and obfuscation in the presence of *dependent* auxiliary inputs.

Independent Auxiliary Inputs. Consider the case where Alice is given two obfuscated programs, one from Bob and one from Carol. She may be able to use the obfuscated program from Carol to extract information about Bob's program. More specifically, Barak's definition doesn't preclude Alice learning some predicate $\pi(\mathcal{P}_{Bob}, \mathcal{P}_{Carol})$. Note that the two programs are generated independently. Tauman-Kalai and Goldwasser capture this situation as follows:

Definition 2-11 *An algorithm $\mathcal{O}$ is an independent-auxiliary-input-secure obfuscator for a class of programs $\{\mathbb{P}_\kappa\}_{\kappa\in\mathbb{N}}$ if*

1. *for every $\kappa \in \mathbb{N}$ and for every $\mathcal{P} \in \mathbb{P}_\kappa$, $\mathcal{O}(\mathcal{P})$ computes the same function as $\mathcal{P}$,*

2. *$\exists$ a polynomial $l(\cdot)$ such that, for every $\kappa \in \mathbb{N}$ and for every $\mathcal{P} \in \mathbb{P}_\kappa$, $|\mathcal{O}(\mathcal{P})| \leq l(|\mathcal{P}|)$,*

3. *for every adversary* $\mathsf{Adv} \in \mathrm{PT}^*$ that outputs a single bit *(i.e. a predicate), there exists a simulator $\mathcal{S} \in PPT$ and a negligible function $\nu(\cdot)$ such that, for every $\kappa \in \mathbb{N}$, for every circuit $\mathcal{P} \in \mathbb{P}_\kappa$ and every auxiliary input z of size polynomial in κ (z may* not *depend on $\mathcal{P}$):*

$$\left|\Pr\left[\mathsf{Adv}(\mathcal{O}(\mathcal{P}), z) = 1\right] - \Pr\left[\mathcal{S}^{\mathcal{P}}(1^\kappa, z) = 1\right]\right| < \nu(\kappa) \ .$$

Dependent Auxiliary Inputs. Consider the case where Alice receives two dependent obfuscated programs from Bob. In this case, a slightly tweaked definition is required. As we do not make use of it in this dissertation, we leave it unstated. One simply needs to notice that the definition is effectively the same as Definition 2-11, with auxiliary input z being quantified such that it can depend on $\mathcal{P}$. The details can be found in Goldwasser and Tauman-Kalai's work [78].

More Capable Adversaries. In the past two definitions, adversaries are defined as minimally capable: they only output a single bit. This is effectively a weak definition of obfuscation, which strengthens the impossibility result. In cases where we wish to *construct* actual obfuscations, however, we must consider adversaries that produce arbitrary (polynomial-size) outputs. In particular, we must consider adversaries that output the obfuscated program itself! The simulator will need to simulate an obfuscated program that looks indistinguishable, even though the simulator doesn't know the parameters of this obfuscated program.

Successful Obfuscations. Based on the Barak et al. definition, Canetti [31] and Wee [173] showed how to obfuscate point functions. Tauman-Kalai and Goldwasser showed how to prove these constructions in their updated model. However, no other successful obfuscation constructions have been proven.

Is Obfuscation Enough? Obfuscation is not enough for security. In particular, a program might be inherently insecure: obtaining a number of input/output pairs for that program might be enough to learn the exact circuit it computes, or at least learn some secret information meant to kept secret. Even if such a program is successfully obfuscated, it still reveals that same information. In other words, for most applications, the program that we wish to obfuscate must be proven secure on its own before it can be obfuscated.

2.6.3 Public-Key Obfuscation

Ostrovsky and Skeith [132] propose a slightly different (and weaker) model of obfuscation, where the outputs of the obfuscated program are encrypted versions of the outputs of the original, unobfuscated program. In other words, their technique allows for outsourcing *most* of a computation, but not all of it: a final decryption is still required after the obfuscated program has been executed. They name this model *public-key obfuscation.*

Interestingly, because the outputs of a public-key-obfuscated program are encrypted, Ostrovsky and Skeith's definition is able to capture the additional notion of security missing from the Barak et al. and Tauman-Kalai and Goldwasser definitions: output indistinguishability. Informally, a public-key obfuscator is secure when an adversary cannot distinguish between the public-key obfuscations of two programs it selected (within a class of programs, of course). We now provide a more formal definition. Given that different versions of Ostrovsky and Skeith's work have provided slightly different definitions, we choose one that best fits the work presented in this dissertation, in particular Chapter 6.

Definition 2-12 (Public-Key Obfuscation) *The algorithm $\mathcal{O}$ is a* public-key obfuscator *for the program class $\{\mathbb{P}_\kappa\}$ and the cryptosystem $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ if:*

1. *(Correctness) there exist a negligible function $\nu(\cdot)$ such that, for every $\kappa \in \mathbb{N}$, for every $\mathcal{P} \in \mathbb{P}_\kappa$, for all inputs x:*

$$\mathsf{Pr}\Big[\mathcal{D}_{sk}\big(\mathcal{O}(\mathcal{P})(x, R)\big) = \mathcal{P}(x)\Big] > 1 - \nu(\kappa)$$

   taken over the choice of R, an extra input which parameterizes the execution of $\mathcal{O}(\mathcal{P})$.

2. *(Conciseness) there is a polynomial $l(\cdot)$ such that, for every $\kappa \in \mathbb{N}$ and for every $\mathcal{P} \in \mathbb{P}_\kappa$,*

$$|\mathcal{O}(\mathcal{P})| \leq l(|\mathcal{P}|)$$

Now, we must define what it means for this public-key obfuscator to be *secure.* Ostrovsky and Skeith give an indistinguishability-based definition. Thus, consider first the indistinguishability experiment. Informally, we first generate a keypair. Based on the public key, the adversary selects two programs from the program class. We obfuscate one of the two, selected at random, and we ask the adversary to guess which one was obfuscated.

We now formalize this intuition, which is much like the semantic security for encryption schemes which we explored earlier in this chapter. We denote $\mathbb{P} = \{\mathbb{P}_\kappa\}_{\kappa \in \mathbb{N}}$.

Experiment 1 (Indistinguishability, $\mathsf{Exp}^{\mathsf{oind}-b}_{\mathbb{P},\mathcal{CS},\mathcal{O},\mathsf{Adv}}(\kappa)$)

$$\begin{array}{rcl} (pk, sk) & \xleftarrow{R} & \mathcal{G}(1^\kappa) \\ (\mathcal{P}_0, \mathcal{P}_1, \mathsf{state}) & \leftarrow & \mathsf{Adv}(\mathsf{choose}, pk), \\ d & \leftarrow & \mathsf{Adv}(\mathcal{O}(1^\kappa, pk, sk, \mathcal{P}_b), \mathsf{state}) \end{array}$$

If $\mathcal{P}_0, \mathcal{P}_1 \in \mathbb{P}_\kappa$ return d, otherwise 0.

We can now define the security property we seek from a public-key obfuscator.

Definition 2-13 (Secure Public-Key Obfuscation) *A public-key obfuscator $\mathcal{O}$ for a program class $\mathbb{P}$ with respect to a cryptosystem $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ is secure, or polynomially indistinguishable, if there exists a negligible function $\nu(\cdot)$ such that:*

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{oind}-0}_{\mathbb{P},\mathcal{CS},\mathcal{O},\mathsf{Adv}}(\kappa) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{oind}-1}_{\mathbb{P},\mathcal{CS},\mathcal{O},\mathsf{Adv}}(\kappa) = 1\right]\right| < \nu(\kappa)$$

Implementations. Ostrovsky and Skeith [132] implement the public-key obfuscation of keyword searching on streams, a protocol with much practical potential, for example in the intelligence community, where one can then outsource to an untrusted platform the searching of streams of data for secret keywords. In Chapter 6, we give an implementation of obfuscated shuffling based on a slight variant of this definition.

2.7 Universal Composability

In 2000, Canetti proposed the Universally Composable framework [32] for proving the security of secure protocols. The goals of this framework are ambitious, and proving security in the UC framework is quite powerful:

- the security properties of any interactive protocol can be described in this single framework,
- once proven UC-secure, a protocol can be composed with other UC-secure protocols to automatically yield a secure hybrid protocol.
- in particular, concurrent composition of different protocols preserves security.

We summarize the key points of the framework here, though the reader should consult Canetti's recently updated description [32] for details. Our take on the framework follows Wikström's simplifications [180], which enable us to focus on the key concepts of the framework without getting lost in technical detail.

2.7.1 Why Another Model?

Goldreich, Micali, and Wigderson [76] introduced the concept of ideal-process simulation in their seminal work of 1987. Goldwasser and Levin [79], Micali and Rogaway [116], and Beaver [14] put forth formalized models of the idea in 1990 and 1991. One might wonder, then, why we need another model? In fact, these earlier models did not capture every cryptographic protocol; they are more adequate for straight-line computation rather than reactive protocols (though they do address some reactive protocols.) Universal Composability follows in the footsteps of these models with some additional properties:

1. complete modeling of reactive processes, such as bit commitments and signatures,
2. enforcement of a realistic ordering and timeline represented by the environment entity, allowing for concurrent composition of protocols, and
3. modeling of adaptive adversaries.

2.7.2 High-Level Intuition

At a high-level, the UC framework declares a protocol secure if it is indistinguishable from an idealized version of the protocol where a trusted third-party (TTP) performs all computation. In this ideal world, a participant communicates directly with the TTP, which is called an "ideal functionality" in the UC framework, and no one but the participant and the ideal functionality sees the content of the messages they exchange. Because networks are inherently imperfect, the ideal-world adversary does have one important ability: it can delay messages arbitrarily long and, in particular, it can reorder messages.

Before any protocol can be proven secure in the UC framework, its specific ideal-world functionality must be well defined. This is particularly important: one cannot define security without thinking about the inherent properties of the protocol. For example, if a protocol computes the average of all participant inputs, one should not be surprised if an adversary discovers the sum of all inputs! No amount of cryptography will prevent this "information leakage."

Indistinguishable to Whom? In the UC model, two "worlds" are run in parallel: the ideal world just described, and the real world that actually implements the protocol. The distinguisher is called the environment, and is denoted $\mathcal{Z}$. It provides the inputs to participants in both worlds, and can interrogate participants at any time for some output. In the end, the distinguisher attempts to determine which was the real and which was the ideal world. If he cannot succeed with probability better than 1/2, then the protocol is considered secure.

Adversaries. In the "real world" of the UC framework, there may be any number of adversaries that do any number of interesting things. In particular, the adversary may try to learn data it should not have access to, e.g. one of the honest participant's secret input, and report it back to the environment. To say that a protocol is secure, we must be sure that *for every possible adversarial strategy* in the real world, there is an adversarial strategy in the ideal world that produces outputs that the environment cannot distinguish from the real world.

This may be somewhat confusing to the reader: why would an adversary *try* to fool the environment? The central point to understand here is that, in the ideal world, we must only prove the existence of such an adversary, because, if *any* indistinguishable adversary exists in the ideal world, it means the strategy used in the real world *didn't yield anything more than it could have in the ideal world.* In other words, the adversary gained no advantage from seeing the real execution of the protocol rather than an idealized version of it.

We now present more details of the UC framework, though we leave the formal definitions to the original paper [32]. Note that the UC model handles static and adaptive adversaries differently. We describe adaptive adversaries at the end.

2.7.3 The Players

In the UC framework, participants $\mathcal{P}_1, \ldots, \mathcal{P}_N$ are Interactive Turing Machines (ITMs), as are the real-world and ideal-world adversaries, respectively $\mathcal{A}$ and $\mathcal{S}$. The ideal functionality is denoted $\mathcal{F}$. Actions in the protocol are serialized: only one ITM is active at a given time, though of course the actions of multiple ITMs may be interleaved. The ITMs are linked via a communication network—denoted $\mathcal{C}$ and $\mathcal{C}_\mathcal{I}$ in the real and ideal worlds respectively (this is the central simplification from Wikström [180])—whose job it is to automatically enforce the constraints we assume about the network. We will return to the nature of these constraints momentarily.

All messages sent either via $\mathcal{C}$ or $\mathcal{C}_\mathcal{I}$ are fully authenticated. Corruption and impersonation are modeled, instead, by having the adversary explicitly corrupt certain players. In particular, as mentioned earlier, adaptive corruption—when an adversary corrupts a participant once the protocol has already begun—is considered separately, as it is particularly challenging.

In the ideal world, honest players $\mathcal{P}_1, \ldots, \mathcal{P}_N$ are nothing more than *dummy participants* whose job it is to forward the inputs provided by $\mathcal{Z}$ straight to the ideal functionality. Similarly, they pass any output received from the ideal functionality straight back to the environment $\mathcal{Z}$.

Initialization and Adversarial Capabilities. At the beginning of the protocol, the real-world adversary must decide which participants $\{\tilde{\mathcal{P}}_j\}$ it wishes to corrupt (recall that we consider adaptive adversaries separately). The ideal adversary $\mathcal{S}$ corrupts the corresponding participants. Then, the environment $\mathcal{Z}$ provides inputs to all of the participants in both the real and ideal worlds.

In both worlds, the adversary learns the inputs of the corrupt parties $\{\tilde{\mathcal{P}}_j\}$. In addition, in the real world, the adversary $\mathcal{A}$ can see all messages exchanged between the honest parties, and can send any message it wants from the corrupt parties. In the ideal world, no messages are exchanged directly between parties, thus $\mathcal{S}$ doesn't see any of the content of these messages. However, it is made aware of the message envelope: who sends how much data to whom. It may choose to delay any message as long as it desires. It also controls the corrupted participants' input sent to the ideal functionality.

The network's properties are embodied in the communication model: $\mathcal{C}$ in the real world and $\mathcal{C}_\mathcal{I}$ in the ideal world. Participants send and receive all messages through this communication model. In the real world, $\mathcal{C}$ simply reports a copy of all messages to $\mathcal{A}$, and waits for $\mathcal{A}$'s signal to deliver the message to its intended recipient. In the ideal world, $\mathcal{C}_\mathcal{I}$ only reports the message envelope to $\mathcal{S}$, and waits for its signal to deliver the content of the message. With $\mathcal{C}$ and $\mathcal{C}_\mathcal{I}$ in the model, the ideal functionality specification can be greatly simplified: all network traffic control issues are generically defined by $\mathcal{C}$ and $\mathcal{C}_\mathcal{I}$.

2.7.4 Simulation and the Environment Distinguisher

To prove security, one must construct an ideal adversary $\mathcal{S}$ which produces outputs indistinguishable from those of $\mathcal{A}$ in the real world, no matter what $\mathcal{A}$ does. Thus, using a classic simulation paradigm, $\mathcal{S}$ will use $\mathcal{A}$ as a black box, simulating for $\mathcal{A}$ the rest of the protocol to elicit from $\mathcal{A}$ an output indistinguishable from that which it gives in the real world.

More specifically, $\mathcal{S}$ must simulate to $\mathcal{A}$ all of the honest participants in the protocol. Where corrupt participants are concerned, $\mathcal{S}$ simply receives their inputs from $\mathcal{Z}$ and forward them to $\mathcal{A}$. $\mathcal{S}$'s central job—and the crux of the difficulty of any UC proof—is to somehow simulate messages to $\mathcal{A}$ on behalf of the honest parties in the protocol when $\mathcal{S}$ doesn't know their real input. At specific points in the protocol, $\mathcal{S}$ receives outputs from the ideal functionality that are destined for corrupt parties, which $\mathcal{S}$ can then use in the protocol simulation to $\mathcal{A}$.

Rewinding. In typical black box simulation proofs, it is common to rewind the black box a number of times to *extract* a witness from a proof of knowledge provided by the black box. Unfortunately, this type of rewinding is quite limited in the UC model, because the environment can never be rewound. Thus, any messages sent to the environment cannot be taken back, and the ideal adversary $\mathcal{S}$ must be able to answer queries from the environment at any time as if it were the real adversary $\mathcal{A}$.

As a result, many protocols proven secure in the UC model use so-called *straight-line extractable* proofs of knowledge. In these proofs, the simulator is able to extract the witness not by rewinding, but by carefully crafting the parameters. One well-known example of this parameter crafting is the double-ciphertext trick of Naor and Yung [122] for building CCA2-secure cryptosystems.

2.7.5 Composability and the Hybrid Model

The particular strength of the UC model is its composability theorem. Consider an ideal functionality $\mathcal{F}_A$, securely implemented by a protocol π_A. Then, consider an ideal functionality $\mathcal{F}_B$ that we wish to implement. If we build a protocol π_B that uses protocol π_A as a subprocedure, once or many times, serially or concurrently, then we can prove that π_B securely implements $\mathcal{F}_B$ without considering the details of π_A.

Specifically, rather than comparing the real world to the ideal world directly, we compare a *hybrid world* to the ideal world. In this hybrid world, the participants $\mathcal{P}_1, \ldots, \mathcal{P}_N$ run the actions of protocol π_B normally. However, when they want to run protocol π_A, they simply use the ideal functionality $\mathcal{F}_A$ directly in this hybrid world. $\mathcal{F}_A$ behaves in the hybrid world exactly like any ideal functionality behaves in the ideal world: the adversary, in this case $\mathcal{A}$, only sees the envelopes of messages between $\mathcal{F}_A$ and the honest participants, though it can reorder and delay messages indefinitely as usual.

2.7.6 An Example of a UC Functionality

As an example of a UC functionality, we present the mixnet functionality first proposed by Wikström [179]. This functionality shows both the simplicity and subtlety of defining a UC functionality. Specifically, the actions of the ideal functionality are quite simple: take the inputs, order them, and eventually output them in lexicographical order.

Functionality 1 (Mix-Net) *The ideal functionality for a mixnet, $\mathcal{F}_{\mathrm{MN}}$, running with mix-servers $\mathcal{M}_1, \ldots, \mathcal{M}_k$, senders $\mathcal{P}_1, \ldots, \mathcal{P}_N$, and ideal adversary $\mathcal{S}$ proceeds as follows*

1. *Initialize the following storage:*

    - *list $L = \emptyset$, the list of messages received from senders,*
    - *set $J_S = \emptyset$, the set of senders who have sent a message, and*
    - *set $J_M = \emptyset$, the set of mix servers who have asked the mix to run.*

2. *Repeatedly wait for messages from $\mathcal{C}_\mathcal{I}$:*

    - *Upon receipt of $(\mathcal{P}_i, \textbf{\textit{Send}}, m_i)$ with $m_i \in \{0,1\}^\kappa$ and $i \notin J_S$:*
        - *set $L \leftarrow L \cup \{m_i\}$*
        - *set $J_S \leftarrow J_S \cup \{i\}$*
    - *Upon receipt of $(\mathcal{M}_j, \textbf{\textit{Run}})$:*
        - *$J_M \leftarrow J_M \cup \{j\}$*
        - *If $|J_M| > k/2$, then sort the list L lexicographically to form a list L', and send:*
            - * *$\{(\mathcal{M}_i, \texttt{Output}, L')\}_{i=1}^{k}$*

and ignore further messages.

If $|J_M| \leq k/2$*, send* $(\mathcal{S}, \mathcal{M}_j, \textbf{\textit{Run}})$*, and keep waiting for messages.*

Recall that, using Wikström's simplified notation, we assume that all sent and received messages go through $\mathcal{C}_{\mathcal{I}}$, which routes envelopes appropriately to the ideal adversary $\mathcal{S}$.

2.8 Voting Protocols

Voting protocols are numerous and diverse. In this section, we review their history. We begin with a high-level description of a voting protocol, generic enough to encompass most of the known proposals. We focus on mixnet-based and homomorphic-aggregation voting systems, as they are the two major types of universally verifiable voting systems. The process and notation are diagrammed in Figure 2-1. We do not consider blind signature voting schemes here, as these have fallen out of favor for lack of universal verifiability. We refer the reader to the work of Gritzalis [86], which provides a review of all secure voting methods.

2.8.1 High-Level Process

Verifiable voting protocols in the literature all present the following sequence of events:

1. **Setup:** Election setup parameters are generated and published.

2. **Ballot Preparation:** Alice, the voter, prepares her ballot with the help of a special ballot or machine. The result is an encrypted vote.

3. **Ballot Recording:** Alice's encrypted ballot is posted on a world-readable bulletin board, paired with Alice's identity in plaintext.

4. **Anonymization & Aggregation:** A publicly-verifiable shuffling (and potentially aggregation) algorithm is run, with intermediate results posted on the bulletin board.

5. **Results:** Election officials cooperate to produce a plaintext tally for each race, again with publicly-verifiable proofs posted to the bulletin board.

In general, two large categories of schemes exist: *aggregate voting schemes*, and *ballot-preserving voting schemes*. In the former system, the output of the protocol indicates only the aggregate number of votes for each candidate in each race. In ballot-preserving voting schemes, all *plaintext* ballots are preserved in their entirety all the way through the tallying process.

2.8.2 The Players & Setup

We designate by l the number election officials, where official i is designated $\mathsf{Official}_i$. We designate by N the number of voters, where voter j is designated Voter_j. The election itself, designated $\mathsf{Election}$, defines s races, where race k is designated R_k. Race R_k has o_k pre-defined options to choose from, though, if the protocol supports it, the voter could select to write in a value. In general, Voter_j's selection for race R_k is denoted $t_j^{(k)}$. Voter_j's ballot is thus

$$m_j = (t_j^{(1)}, t_j^{(2)}, \dots, t_j^{(s)}).$$

The election $\mathsf{Election}$ defines configuration parameters, including a system-wide public key pk and l secret key shares $sk^{(1)}, \dots, sk^{(l)}$, where $sk^{(i)}$ belongs to election official $\mathsf{Official}_i$.

2.8.3 Casting a Ballot

Voter_j casts plaintext ballot m_j encrypted into c_j using randomization value r_j under public key pk. The encryption algorithm depends on the tallying and anonymization technique, though it must clearly be semantically secure (and thus randomized.) Specifically, the encoding mechanism of $m_j = (t_j^{(1)}, t_j^{(2)}, \dots, t_j^{(s)})$ depends on the tallying mechanism, though once the encoding is achieved, most schemes perform a normal public-key encryption under pk.

The key requirement, when Alice casts a ballot, is that she gain assurance that c_j encodes her intended ballot m_j. At the same time, Alice should not be coerced by an adversary. Here, we review these two issues, starting with incoercibility. We then specifically review proposals that prove to Alice that her vote correctly encodes her intent while taking into account the fact that Alice is human and cannot perform much computation on her own.

Incoercibility

An important requirement of elections is that adversaries not be able to coerce voters. Often, this property is called *receipt-freeness*, because the protocol must not provide a true receipt of interaction that the voter could present to a coercer. Recall that we cannot simply build a system where Alice is expected to discard her receipt of her own volition, because a coercer can threaten retribution if Alice doesn't preserve her receipt. We note that the terminology of *receipt-freeness* can be confusing because the word "receipt" is misleading: it is possible to provide the voter with a *secret receipt* that she can use for personal verification, but not for proving to a coercer how she voted. The word *incoercibility* better characterizes this property.

The key challenge of achieving incoercibility is to design a process that convinces Voter_j that c_j is indeed the encryption of m_j, her plaintext vote, without revealing the randomization value r_j. Indeed, as we will see, c_j is generally posted on a public bulletin board, along with the voter's name or identifier, for all to see. Thus, if Voter_j learns r_j, she can trivially prove how she voted.

Benaloh and Tuinstra [20] first introduced and implemented incoercible voting in 1994. In their basic proposal, Alice, the voter, is isolated in a typical voting booth, where she privately receives the decryption of some publicly available ciphertexts. She then publicly casts a ballot using a selection of these encrypted bits, whose plaintext value she just learned. Since Alice does not learn the randomization values for these ciphertexts, and since the cryptosystem is semantically secure, she cannot prove how she voted, and no coercer can figure it out on his own either. In their more advanced protocol with multiple authorities, Alice receives proofs of plaintext value that she can even manipulate to claim the opposite of how she actually voted.

The voting booth is effectively a *private channel* between the voter and the election officials. Using this model, a number of additional schemes were developed. Sako and Kilian [150] and Cramer et al. [45] generalized the schemes of Benaloh and Tuinstra and improved its efficiency, though still for approval voting, where ballots are either 0 or 1. Sako and Kilian [151] also adapted these receipt-free techniques to mixnet voting. Okamoto [129] provided receipt-freeness for blind-signature based voting with full-length plaintexts, based on trapdoor commitments: Alice can claim to have voted differently than she actually did.

In 1996, Canetti and Gennaro [33] presented a more generic definition of incoercible multiparty computation and showed how to accomplish it without the private channel assumption. Instead, they use deniable encryption, where the sender of an encrypted message can "present a different story" regarding the encrypted message's plaintext. They specifically implement threshold deniable encryption, which is critical for the voting setting. Interestingly, they also show that incoercible computation is impossible in the face of unbounded coercers. This will be particularly interesting to our work on Assisted-Human Interactive Proofs, in Chapter 5.

The techniques of incoercibility have been particularly explored in the case of homomorphic voting schemes, where efficiency is a critical factor. Of particular note is the work of Cramer et al. [46], Hirt and Sako [90], and Baudron et al. [13]. We briefly review their work in the upcoming paragraph on homomorphic voting schemes.

Human-Verifiable Voting Systems

All of the incoercible schemes just described assume a computationally capable prover: an unaided human cannot perform the math required to participate in the protocol. This is of singular concern in the case of voting systems: if Alice needs to use a computer to cast her ballot, then she must trust the computer. This is by no means a trivial proposal in the case of a federal election where the stakes are particularly high (see Chapter 1).

Neff [28] first proposed a scheme where voting machines that assist Alice in casting her ballot can be audited during election day. At any time, an election official (or other auditor) can use the machine as if he were a voter, then request an audit. Probabilistically, the voting machine cannot cheat more than a handful of voters before it gets caught.

Chaum [40] introduced the term "secret-ballot receipts." In his proposal, a voting machine prints a special ballot composed of two layered pieces of paper where each half looks random, but the superposition of the two layers yields the plaintext vote, as defined by Naor and Shamir [121] in their work on visual cryptography. Numerous variants have emerged from this work, including Ryan's Prêt-a-Voter [41] and Chaum's own simplified version [66]. In Chapter 4, we present our own variant of this type of scheme, with some interesting new properties.

Neff [29] also proposed a machine-based system with a proof technique that can be split into two parts: an interactive portion verified by a human inside a voting booth, and a non-interactive portion verified by a helper outside the voting booth. The non-interactive portion should not reveal the plaintext of the vote, of course. In Chapter 5, we formalize this type of proof with a human-verifiable version of incoercibility, tweak the original protocol to make it truly incoercible according to this definition, and provide a more efficient version of the protocol.

2.8.4 Anonymizing and Aggregating the Ballots

Once a ballot has been correctly encrypted, it is posted on a bulletin board for all to see. The ballots must then be anonymized and aggregated in some way. The process for anonymizing and aggregating ballots differs significantly between the ballot-preserving and aggregate voting systems. Thus, we consider each one separately. We spend a bit more time reviewing the known techniques for aggregate ballot systems, because Chapter 3 addresses the ballot-preserving schemes in great detail.

Ballot-Preserving Election Systems

In the case of ballot-preserving election systems, election officials cooperate to perform anonymization of the ballots. The encrypted inputs are denoted:

$$\{c_{0,j}\}_{j\in[1,N]}$$

Each election official $\mathsf{Official}_i$, with $i \in [1,l]$, in turn processes the encrypted votes. The inputs to $\mathsf{Official}_i$ are denoted:

$$\{c_{i-1,j}\}_{j\in[1,N]}$$

while its outputs, which are also the inputs to $\mathsf{Official}_{i+1}$, are denoted:

$$\{c_{i,j}\}_{j\in[1,N]}$$

The type of anonymous channel that performs this repeated shuffling is called a mixnet, and each official is called a mix server. Numerous techniques exist for this process. The details and chronology are detailed in Chapter 3, but we give a brief overview here.

Chaum [39] introduced mixnets in 1981. Mixnet-based protocols evolved significantly throughout the 1990s starting with Pfitzmann and Pfitzmann's break and fix of Chaum's original mixnet [138]. Reencryption mixnets were introduced by Park et al. [134], making ciphertexts length-invariant in the number of mixes. The scheme was broken and fixed by

Pfitzmann [136], who introduced improved semantic security for El Gamal using q-order subgroups. Sako and Kilian [151] introduced universally-verifiable mixnets, which Ogata et al. [127] made robust against failure and Abe [1] made verifiable in time independent of the number of mix servers. Then came a number of schemes [2, 70] that provided more efficient universally verifiable proofs using special algebraic properties, rather than generic cut-and-choose, in particular Neff's scheme [124], using proof of equality of exponent vector dot product, which Groth [87] generalized to the abstract homomorphic setting. Most recently, Wikström introduced a security definition of a mixnet in the UC setting [179], and a new proof of shuffle secure against this definition [180], including an adaptively-secure mixnet [181] in collaboration with Groth.

Aggregate Voting Systems

In the case of aggregate voting systems, the encrypted votes $\{c_j\}_{j\in[1,N]}$ are generally combined into a single set of ciphertexts $C^{(k)}_{tally}$, where $C^{(k)}_{tally}$ encodes the tallies for race R_k. Depending on the scheme and the size of the election, $C^{(k)}_{tally}$ may be a *sequence* of ciphertexts, though usually never more than one per candidate (i.e. not one per voter, as that would make it a ballot-preserving scheme.)

Aggregate voting systems typically use homomorphic cryptosystems to maintain and increment aggregate tallies under the covers of encryption. Benaloh [43, 19] presented the first practical additive homomorphic scheme for just this purpose: a yes-vote is an encrypted '1', a no-vote is an encrypted '0'. The tally is achieved by homomorphic addition, which anyone can do using only the public key, and threshold decryption, which the officials perform together.

Cramer et al. [45] provided performance improvements, specifically regarding the effect of multiple authorities, and, most interestingly, provided information theoretic privacy of the votes posted on the bulletin board, assuming a separate private channel between the voter and official (e.g. a voting booth). A further improvement by Cramer et al. [46] achieved optimal performance, though "only" with computational privacy.

The problem with additive homomorphic schemes is that they typically achieve their desirable homomorphism by placing the plaintext in the exponent, so that ciphertext multiplication will yield plaintext addition. Consider, as an example, the Exponential version of El Gamal:

$$(g^r, g^m y^r) \times (g^{r'}, g^{m'} y^{r'}) \quad = \quad (g^{r+r'}, g^{m+m'} y^{r+r'})$$

Thus, until 1999, all of these schemes required the computation of a discrete logarithm for decryption, which significantly limits the plaintext space. For single-race approval elections, this is typically not a problem, as the total counter remains quite small: even a counter that can accommodate one hundred million voters requires no more than 28 bits. However, with multiple candidates and multiple races, the early homomorphic ballots quickly becomes unwieldy, requiring one ciphertext per candidate per race for each ballot.

Major progress in homomorphic aggregation techniques became possible with the introduction of the Paillier cryptosystem [133], which offers an additive homomorphism with efficient decryption. It then became possible to "pack" multiple counters within a single encrypted tally, leading to the efficient protocol of Baudron et al. [13] for multiple candidates and races. In addition, the work of Damgård and Jurik [48] on generalized Paillier enabled even larger plaintext spaces, thus enabling more counters—and thus larger elections—in a single ciphertext tally.

One notes that all homomorphic cryptosystems *require* extensive checks on the inputs they process. Otherwise, a malicious input might simply add '1000' into the count of the adversary's favorite candidate. Benaloh [43, 19] first described a zero-knowledge proof of correct form for his ballots, and subsequent homomorphic schemes have provided similar proofs for their inputs.

2.8.5 Tallying

Whether the anonymization and aggregation process yields a handful of tally ciphertexts or individual encrypted ballots, all voting systems then require election officials to perform some form of threshold decryption on these resulting ciphertexts. In the case of an aggregate voting scheme, the result is $M^{(k)}_{tally}$ for each race R_k. In the case of ballot-preserving voting schemes, the result is a set of plaintext ballots, $\{m'_j\}_{j\in[1,N]}$.

Of course, the specific method of threshold decryption often depends significantly on the method used to anonymize and aggregate the encrypted ballots. In particular, a number of mixnet schemes provide combined decryption and shuffling: each election official shuffles and partially decrypts. Again, the details of these schemes is presented in Chapter 3.

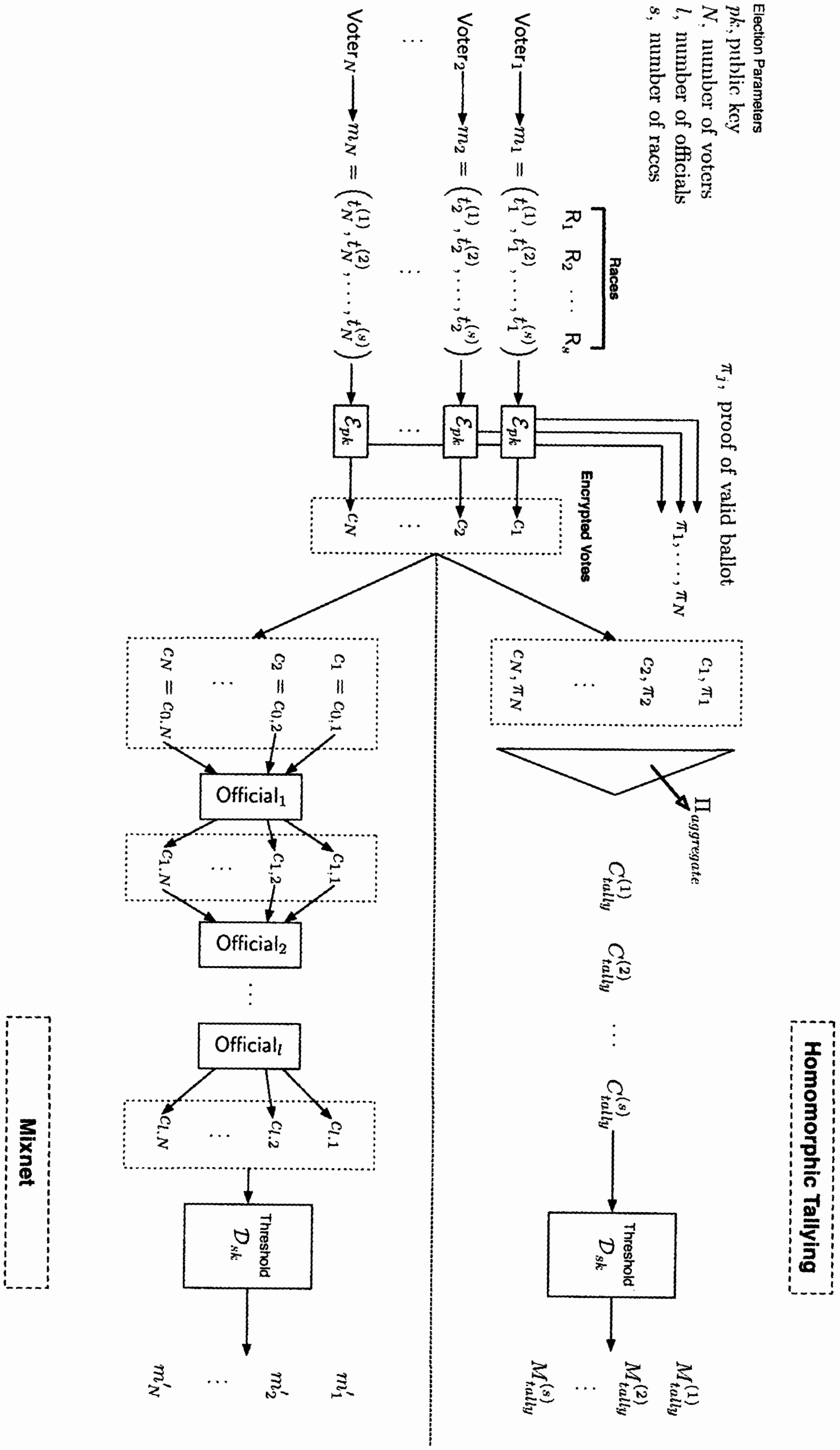


Figure 2-1: Cryptographic Voting Notation: Homomorphic Tallying and Mixnet.

Chapter 3

Verifiable Mixnets: A Review

3.1 Introduction

Consider a set of senders, each with a private message, who wish to generate a shuffled list of these messages, while keeping the permutation secret. Protocols that implement this functionality were first introduced by Chaum [39] in 1981, who called them *mixnets.* There are many different types of mixnets, and many different definitions and constructions. At a high level, mixnets can be categorized into two groups: *heuristics-based mixnets*, and *robust mixnets.*

Heuristics-based mixnets tends to mix inputs more or less synchronously for low-latency applications such as anonymized web browsing [57]. These mixnets generally focus on achieving some level of privacy, without usually worrying about robustness: if a few mix servers drop or otherwise corrupt messages, the impact on the application is generally not horrible: a sender can simply retry using a different set of mix servers.

By contrast, robust mixnets handle applications like voting, which have significantly different requirements. On the one hand, they provide far more flexibility: mixing can take hours or, in some cases, even days, because shuffling is performed in large, well-defined batches, with no need for real-time responses. On the other hand, the correctness requirements are much more stringent: inputs should not be lost or altered, in some cases even when all mix servers are corrupt. The privacy of the shuffle permutation is also important, and should be provably—not just heuristically—protected.

In this chapter, we review the past 25 years of literature on verifiable mixnets. We note that this area of research has been quite productive, with numerous directions explored, interesting attacks discovered, and fascinating techniques developed to improve efficiency. The security definitions have evolved, too, especially in light of recent security frameworks for composability. In short, mixnets have been a fertile area of research. For their critical role in key cryptographic applications, they deserve careful consideration.

3.2 Notation & Preliminaries

We begin with some brief preliminaries regarding notation, categorization of mixnets according to properties of soundness and privacy, and the omnipresent bulletin board in all mixnet proposals.

3.2.1 State of Notation

In the scope of heuristics-based mixnets, where the anonymization process is often called *onion routing*, there has been much debate about notation and a recent attempt at simplification and standardization [42]. However, the notation difficulties of onion routing are not the same as those encountered in describing robust mixnets. Most modern robust mixnets consider the actions of individual mix servers, rarely needing to consider the full composition of symmetric key encryptions. Thus, there is rarely a need to represent the multi-wrapping of an onion.

On the other hand, the widely varying notation used over the years of robust mixnet development remains an impediment to the understanding and comparison of various techniques. Here, we provide a notation generic enough to represent the various robust schemes. The notation aims to be simple and intuitive enough to understand and read quickly. It is optimized for schemes that focus on individual mix servers, eschewing attempts to compactly represent the actions of the mixnet as a whole, except where the algebra inherently simplifies such notation, of course.

3.2.2 Mixnet Notation

Consider a mixnet $\mathcal{M}$ composed of l mix servers, where the i'th mix server is denoted $\mathcal{M}_i$, and i is 1-indexed. For $j \in [1, N]$, $\mathcal{M}_i$ receives ciphertext inputs $c_{i-1,j}$ and produces ciphertext outputs $c_{i,j}$. We denote $\mathcal{MD}^{(i-1)}$ and $\mathcal{MD}^{(i)}$ the decryption algorithms for ciphertexts before and after mix server $\mathcal{M}_i$, respectively, noting that the decryption algorithms may be the same. We emphasize that these decryption algorithms are not be part of the mixnet itself, as the mix servers do not know how to fully decrypt the messages. We denote π_i the permutation applied by mix server $\mathcal{M}_i$. Then, each mix server $\mathcal{M}_i$ computes its outputs such that:

$$\forall i \in [1, l], \forall j \in [1, N], \mathcal{MD}^{(i)}\left(c_{i,\pi_i(j)}\right) = \mathcal{MD}^{(i-1)}\left(c_{i-1,j}\right).$$

This setup is diagrammed in Figure 3-1.

Notation for Randomization Values. When a mix server $\mathcal{M}_i$ provides a proof of mixing, it is usually a proof of knowledge of π_i and randomization factors $r_{i,1}, r_{i,2}, \ldots, r_{i,N}$. These randomization factors could be used for reencryption, or could be random padding extracted after a decryption. In some edge cases, e.g. Wikström's sender-verifiable mixnet [180], the entire witness w_i may, in fact, be a short secret, e.g. a secret key sk_i.

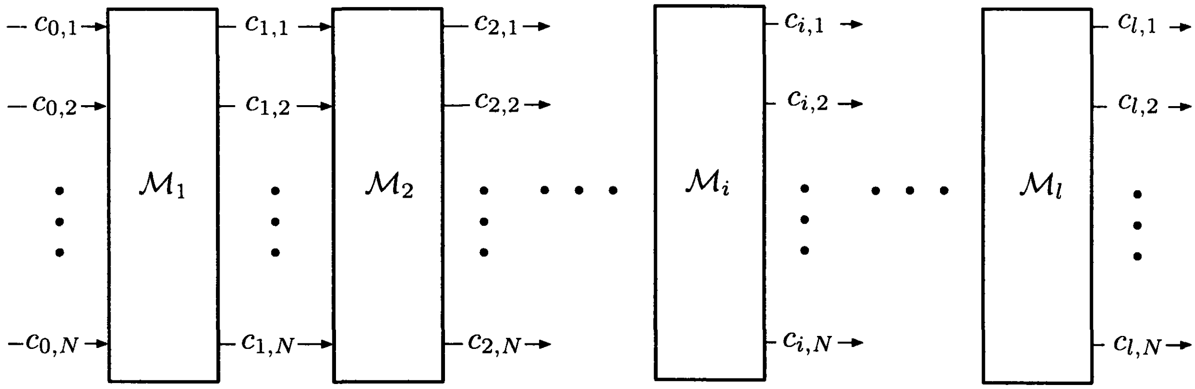


Figure 3-1: **Mixnet Notation**. There are l mix servers, and N inputs.

3.2.3 Categorizing Mixnets

For mixnets that provide interactive proofs of correctness—the ones we're really interested in—there are two major issues to consider: how much information leaks, if any, and under what conditions can the mix servers cheat the proof protocol. These are not boolean questions of course: some information leakage may be acceptable, and some hardness assumptions may be acceptable to guarantee soundness. However, it is important to be aware of these issues when comparing different mixnet constructions.

Privacy. Privacy in a mixnet always depends on computational assumptions: since the inputs and outputs are ciphertexts in a public-key setting, a computationally-unbounded adversary can simply decrypt the inputs and outputs and discover the mixnet permutation. Thus, any privacy guarantee is made under computational assumptions. The more interesting question is the completeness of the privacy. We consider three levels of privacy completeness in a mixnet:

- **complete & independent**: all permutations of inputs-to-outputs are possible, and a computationally-limited adversary gets no information about any correspondence. If, through other channels, some subset of the input-output correspondences are revealed, the proof does not provide any additional correspondences.
- **complete but dependent**: any input can go to any output, but not all permutations are possible. In other words, if a subset of correspondences are revealed through other channels, the proof may leak additional correspondences.
- **incomplete**: the proof itself narrows down the possible correspondences: only some subset of the inputs may correspond to some subset of the outputs.

Clearly, from a theoretical standpoint, the **complete & independent** proofs are the most interesting, and, in fact, the only ones likely to fulfill reasonable security definitions.

However, for some practical settings like voting, the other schemes are still useful: if votes are shuffled across many thousands of voters, revealing some dependencies in the shuffle may be no more harmful than the practical leaks already encountered in the normal voting process, e.g. the precinct's historical voting patterns, the voter's party registration, etc ....

Soundness. The soundness requirement presents significantly more variability. In particular, it is possible to achieve overwhelming soundness without any hardness assumptions. This is particularly interesting in the voting setting, of course, as one must assume that a corrupt mix server has significant computational power at its disposal: it is quite reassuring when no amount of computational power can help the mix server cheat.

We note that, in the case of zero-knowledge protocols, the question at hand is whether the protocol is a zero-knowledge *proof* or a zero-knowledge *argument* [75]. We consider this problem more generally, however, given that not all mixnet proofs are zero-knowledge. Thus, we distinguish three levels of soundness:

- **overwhelming proof**: even a computationally unbounded prover has a negligible chance of successfully proving an incorrect shuffle.
- **overwhelming argument**: a computationally bounded prover has a negligible chance of successfully proving an incorrect shuffle.
- **high but not overwhelming proof or argument**: a prover has a small, but not negligible, chance of successfully proving an incorrect shuffle.

From a practical standpoint, the overwhelming *proofs* are clearly more interesting. The high-but-not-overwhelming techniques are typically much more efficient, and can be viewed as techniques to provide "early returns" in the voting setting.

Interaction of Soundness and Privacy. The two issues of soundness and privacy are intrinsically linked. In particular, if a proof technique is fast but not overwhelmingly sound, it can still be useful, *as long as it doesn't leak privacy.* One can always perform multiple proofs in series: a fast but not overwhelmingly sound proof for early approximate results, and a slower, overwhelmingly sound proof for final assurance. If the former proof fully protects privacy, then there is no disadvantage to using it. (Running these proofs in parallel is risky, unless they've been proven secure in an appropriate model.)

3.2.4 The Bulletin Board

Most robust mixnet protocols make use of a bulletin board, which we denote $\mathcal{BB}$. This bulletin board is effectively a robust, authenticated broadcast channel. In the universally verifiable setting, it is expected that all postings to $\mathcal{BB}$ are recorded for any observer to check. We do not cover the various known algorithms for implementing such a bulletin board, though we note that, without pre-existing cryptographic assumptions like a public-key infrastructure, the problem is one that requires a Byzantine agreement algorithm, as the servers that implement the bulletin board may be corrupt. In this case, fewer than 1/3 of the servers can be corrupt. The reader may wish to consult the work of Lindell, Lysyanskaya and Rabin [110].

3.2.5 Structure of this Work

The various mixnet protocols developed over the years introduced and progressively standardized various design techniques. Many of these techniques are, by now, quite common and well understood, their origin almost forgotten. Wherever possible, we point out the design patterns introduced by each construction, and we reference them when subsequent constructions reuse these patterns.

In addition, various types of attacks have emerged over the years. When a novel type of attack is presented, we also point it out. When a variant of an attack is introduced, we indicate its genealogy.

As a result, the earlier mixnets are described in greater detail, so that the design principles may be fleshed out. The later mixnets are described more briefly, referencing past design principles, past attacks and countermeasures, and detailing only the novelties introduced. Where the complexity of a scheme is particularly high, we only give a high-level intuition, referring the reader to the original work for the details.

Of course, we cannot be complete: there are dozens and dozens of mixnet papers. We attempt to cover the most important ones, specifically the contributions that are most often referenced and that have contributed to the most useful mixnet research directions.

Chronology. We begin, in Section 3.3, with a description of the first mixnets: Chaum's decryption mixnet and Pfitzmann and Pfitzmann's relation attack, then Park et al.'s reencryption mixnet and Pfitzmann's semantic security attack. In Section 3.4, we cover the major wave of universally verifiable mixnets: Sako and Kilian's ground-breaking work and the Michels-Horster attack, Ogata et al.'s alternative verification proposal, Abe's improved proof, and Jakobsson et al.'s generic method – randomized partial checking—for checking any mixnet with slightly relaxed soundness. In Section 3.5, we cover the efficient algebraic proofs: Abe's permutation network scheme, Juels and Jakobsson's millimix, and the linear-time proofs of Furukawa and Sako and, independently, Neff (including Groth's generalization). Then, in Section 3.6 we explore various proof attempts using aggregate properties including work by Jakobsson and Golle et al., and we note the surprising number of attacks discovered against such techniques, including those of Desmedt and Kurosawa, and Wikström. In Section 3.7, we briefly review some particularly interesting variants, including hybrid mixing, universal reencryption, almost entirely correct mixing, and Parallel Mixing. Finally, in Section 3.8, we give an overview of the latest, universally composable security definitions for mixnets, with a focus on Wikström's latest constructions. We conclude with a summary table of some of the mixnet schemes we covered in Section 3.9, and some thoughts on where future mixnet research is likely to go.

3.3 Early Mixnets

3.3.1 Chaumian Mixnet

The first mixnet was introduced by Chaum in 1981 [39], using RSA onions with random padding. Mixnets based on this concept of composed encryption and single-layer decryption at each mix server are sometimes called "Chaumian mixnets." Each mix server $\mathcal{M}_i$ has a public key pk_i and corresponding secret key sk_i. Inputs to the mixnet are prepared as:

$$c_{0,j} = \mathcal{E}_{pk_1}\left(r_{1,j}, \mathcal{E}_{pk_2}\left(r_{2,j}, \ldots \mathcal{E}_{pk_l}(r_{l,j}, m) \ldots\right)\right)$$

Each mix server $\mathcal{M}_i$ then decrypts the outer layer of this onion (if things are done correctly, then the outer layer can be decrypted by the designated mix server), removes the random padding $r_{i,j}$, and outputs the resulting set of diminished onions in lexicographic order.

Design Principle 1 (Encrypted Onion) *A plaintext is repeatedly wrapped using a different public key and random padding at every layer. Each layer is unwrapped by the corresponding mix server.*

Sender Verification in Voting Protocol. Chaum proposes a variant of this channel for voting protocols, in order to help voters check that their vote was properly forwarded along. The protocol requires two runs of the mixnet: in a first run, Alice sends a public key pk_j for which she has the secret key sk_j. She then checks that her public key makes it on the final, decrypted posting to the bulletin board. Then, in the second run of the mixnet, she sends $(pk_j, \mathcal{E}_{sk_j}(m_j))$, where m_j is her vote, padded with a pre-determined number of 0s. Note how encryption is performed with the *secret* key here, flipping the public-key cryptosystem around. Then, everyone can perform the public-key based decryption of the vote in the final round, verifying that only 0-padded results emerge.

This two-step mixing ensures that, in the first phase, a voter can complain if her public key doesn't end up on the bulletin board by revealing all of her randomization values. Then, in the second phase, only messages formed with the first-phase public keys are allowed on the bulletin board. No one but the sender—in possession of sk_j—can prepare a properly 0-padded plaintext.

Breaking the First Chaumian Mixnet. Nine years after the publication of this first Chaumian mixnet, Pfitzmann and Pfitzmann discovered a significant attack using the multiplicative homomorphism of raw RSA and the independent randomness of the padding [138]. The attacker uses two sequential shuffles, providing an adaptively-chosen input to the second shuffle in order to trace an input of the first shuffle.

Specifically, if the attacker wishes to trace input $c_{0,j}$ in the first batch, he provides to the second batch the input $c^* = c_{0,j} \cdot \mathcal{E}_{pk_1}(f)$, using a small f. This algebraic relationship between the inputs yields an algebraic relationship between the outputs, even with the padding. With high probability, no other pair of outputs bears this same relationship. The attacker can thus check every possible output pair until the relationship is found, thereby tracing where the targeted output of the first batch ended up. We call this the *related input attack.*

Attack 1 (Related Input) *Eve, a malicious participant, submits a mixnet input that is related to Alice's honest input. The ciphertext relationship results in a plaintext relationship, which Eve can detect in the mixnet outputs: Eve discovers Alice's private input.*

Pfitzmann and Pfitzmann briefly mention potential countermeasures relating relating to sender-commitments of their inputs and adaptively-secure cryptosystems, but they do not delve into complete details.

3.3.2 First Reencryption Mixnet

In 1993, Park et al. [134] noted that Chaumian Mixnets require a ciphertext size proportional to the number of mix servers, given the concatenation of randomness at each layer of the onion. They proposed the first *reencryption mixnet*, where each mix server rerandomizes the ciphertexts with fresh randomization values that get algebraically combined with existing randomness, rather than concatenated.

The system parameters are p, a prime, the factorization of $p-1$, and g a generator of $\mathbf{Z}_p^*$ (the factorization of $p-1$ is made available to ensure that anyone can verify that g is a generator). Each mix server $\mathcal{M}_i$ generates a secret key

$$sk_i = x_i \xleftarrow{R} \mathbf{Z}_{p-1}^*$$

and the corresponding public key

$$pk_i = y_i = g^{x_i} \bmod p.$$

We denote El Gamal ciphertexts as:

$$c = \mathcal{E}_{pk}(m; r) = (\alpha, \beta) = (g^r, m \cdot y^r)$$

Consider the mixnet's joint public key:

$$\mathsf{PK} = \prod_{i=1}^{l} pk_i = g^{\sum_{i=1}^{l} x_i} = Y$$

Recall the ability to perform reencryption with El Gamal:

$$\mathcal{RE}_{pk}(c; r') = (\alpha \cdot g^{r'}, \beta \cdot y^{r'}) = \mathcal{E}_{pk}(c; r + r')$$

An input to the mixnet is the encryption of the plaintext under the joint public key:

$$c_{0,j} = \mathcal{E}_{\mathsf{PK}}(m_j; r_j)$$

Mix server $\mathcal{M}_i$ then reencrypts each ciphertext with fresh randomness:

$$c_{i,j} = \mathcal{RE}_{\mathsf{PK}}(c_{i-1,j}; r_{i,j})$$

The final output $c_{l,j} = (\alpha_{l,j}, \beta_{l,j})$ is then joint-decrypted by the mix servers, using straightforward El Gamal shared decryption.

Design Principle 2 (Reencryption Mixnet [134]) *Mixnet inputs are encrypted using a cryptosystem with reencryption, usually a homomorphic scheme like El Gamal. Each mix server then shuffles and re-randomizes the ciphertexts by homomorphically multiplying by 1 (or adding 0 if the scheme is additively homomorphic). Decryption occurs after shuffling is finished.*

Variant with Partial Decryption. Park et al. also propose a slightly different mixnet, where the reencryption and decryption phases are performed simultaneously, with each mix server effectively performing partial decryption at each reencryption stage. The last mix server then only needs to perform a normal El Gamal decryption using his single secret key. In the original paper, this variant is actually presented as the primary protocol. However, with the benefit of hindsight, we mention this partial decryption scheme second, since the reencryption-then-decrypt scheme is a bit simpler to understand, and provides the basis for many subsequent schemes. Consider the following notation:

$$\mathsf{PK}_i = \prod_{i'=i}^{l} pk_{i'} = g^{\sum_{i'=i}^{l} x_{i'}} = Y_i$$

In other words, $\mathsf{PK}_i = Y_i$ is the joint public key for the sequence of mix servers starting with $\mathcal{M}_i$. Note that $\mathsf{PK}_1 = \mathsf{PK}$, the joint public key for all mix servers, and that $\mathsf{PK}_l = pk_l$, since $\mathcal{M}_l$ is the last mix server. The inputs to the mixnet are the same, but the actions of the mix server $\mathcal{M}_i$ are slightly different. When a ciphertext reaches mix server $\mathcal{M}_i$, it is expected to be of the form $\mathcal{E}_{\mathsf{PK}_i}(m; R_i)$ and it is denoted $c_{i-1} = (\alpha_{i-1}, \beta_{i-1})$. Using its secret key $sk_i = x_i$, the mix server can transform this ciphertext under PK_i into a ciphertext under PK_{i+1}, which is the joint public key of the remaining mix servers (a combination of one fewer public key). Consider this partial decryption operation as follows:

$$\mathsf{PartialDec}_{sk_i}(c) = (\alpha, \beta \cdot \alpha^{-x_i})$$

Note, in particular, how:

$$\mathsf{PartialDec}_{sk_i}(\mathcal{E}_{\mathsf{PK}_i}(m)) = \mathcal{E}_{\mathsf{PK}_{i+1}}(m)$$

Thus, mix server $\mathcal{M}_i$ performs the following actions on each ciphertext input:

$$c_{i,j} = \mathcal{RE}_{\mathsf{PK_{i+1}}}\left(\mathsf{PartialDec}_{sk_i}(c_{i-1,j}); r_{i,j}\right)$$

Design Principle 3 (Reencryption-and-Decryption Mixnet [134]) *Mixnet inputs are encrypted using a cryptosystem with reencryption, usually a homomorphic scheme like El Gamal. The mixnet public key is jointly generated by the mix servers, each of which possesses a share of the secret key. Each mix server then shuffles, reencrypts, and partially decrypts the ciphertexts. The last mix server's outputs are the mixnet's plaintext outputs.*

Breaking the First Reencryption Mixnet. A year after these first reencryption-based mixnets were published, Pfitzmann [136] showed significant breaks against both.

First, note how El Gamal used over all of $\mathbf{Z}_p^*$ is not semantically secure: by testing the respective subgroup memberships of $\alpha = g^r$ and $\beta = m \cdot y^r$, one can infer information about the subgroup membership of m. Even a passive adversary that doesn't choose any plaintext inputs to the mixnet can observe algebraic relationships and significantly pare down the possible inputs of a given output. We call this attack the *semantic security attack*.

Attack 2 (Semantic Security [136]) *In a reencryption-based mixnet, if the cryptosystem is not semantically secure, an attacker can detect input/output relationships by comparing the input ciphertexts and the last ciphertexts before decryption. Often, the comparison will be algebraic, but any comparison that utilizes the lack of semantic security is a possible attack.*

The countermeasure proposed by Pfitzmann is to generate p as a safe prime, where a large prime q divides $p-1$, g is selected as the generator of a q-order subgroup of $\mathbf{Z}_p^*$, and $m \in \langle g \rangle$. A few years later, this approach was formalized by Tsiounis and Yung [168], who proved the semantic security of El Gamal under this construction.

Countermeasure 1 (Making El Gamal Semantically Secure [136, 168]) *When used in mixnets, El Gamal should be properly parameterized for semantic security: all plaintexts and ciphertexts should be in the q-order subgroup of $\mathbf{Z}_p^*$, where p is selected such that q, a large prime, divides $p-1$.*

Second, even with the semantic security countermeasure in place, an active attacker can submit an input algebraically related to another input, using the homomorphic properties of El Gamal. This is a variant of Pfitzmann's earlier related-input attack (Attack #1). In this variant, Eve uses input $c^* = (\alpha^e, \beta^e)$ for a randomly chosen e where (α, β) is an existing input to the mixnet. Then, one can check which pairs of plaintext outputs (m_0, m_1) fits the relation $m_0^e = m_1$. Similarly, one could use $c^* = (\alpha \cdot g^{r^*}, m^* \cdot \beta \cdot y^{r^*})$, then check for $m_0 = m^* \cdot m_1$. Note that Pfitzmann describes this attack carried out by a small number of dishonest mix servers against the remaining majority of mix servers. However, it seems that, in a realistic setting where not all participants post their message on the bulletin board at the same time, a malicious participant could perform this attack on his own.

As a countermeasure to this active attack, Pfitzmann suggests using techniques developed around the same time for making El Gamal ciphertexts secure against chosen ciphertext attack [47, 140]. Pfitzmann also suggests using redundancy in the plaintexts, though she notes that, if the last mix server is corrupt, it can simply replace the malicious plaintext with a "corrected" one, so that no observer notices the difference.

Countermeasure 2 (Non-Malleability) *Inputs to a mixnet are made non-malleable, so that Eve cannot take Alice's input and convert it into a related input of her own. This countermeasure may include message redundancy or techniques for chosen-ciphertext security.*

3.4 Universally Verifiable Mixnets

The Pfitzmann attacks provided significant motivation to move beyond ad-hoc design for mixnet protocols. In the mid 1990s, new mixnets began to exhibit a property called *universal verifiability*, as first defined by Sako and Kilian [151]. Security definitions were introduced in an attempt to capture the properties desired from mixnets. In the process, the efficiency of mixnets took a significant hit, as the proofs of correctness were particularly onerous.

3.4.1 Introduction of Universal Verifiability

In 1995, Sako and Kilian proposed the first *universally verifiable* mixnet [151] based on the techniques of Park et al. [134]. Sako and Kilian's work was the first mixnet to provide a proof of correct mixing that any observer can verify. As a result, specific attention to individual verifiability—tracing one's own input—was no longer necessary.

This proposal begins with the partial-decryption-and-reencryption mixnet of Park et al. (Design Principle 3), with the countermeasure suggested by Pfitzmann (Countermeasure 1). In addition to its existing tasks, each mix server $\mathcal{M}_i$ also publishes the intermediate results $\mathsf{PartialDec}_{sk_i}(c_{i,j})$, the partial decryption of each input prior to reencryption and shuffling. Then, each mix server provides:

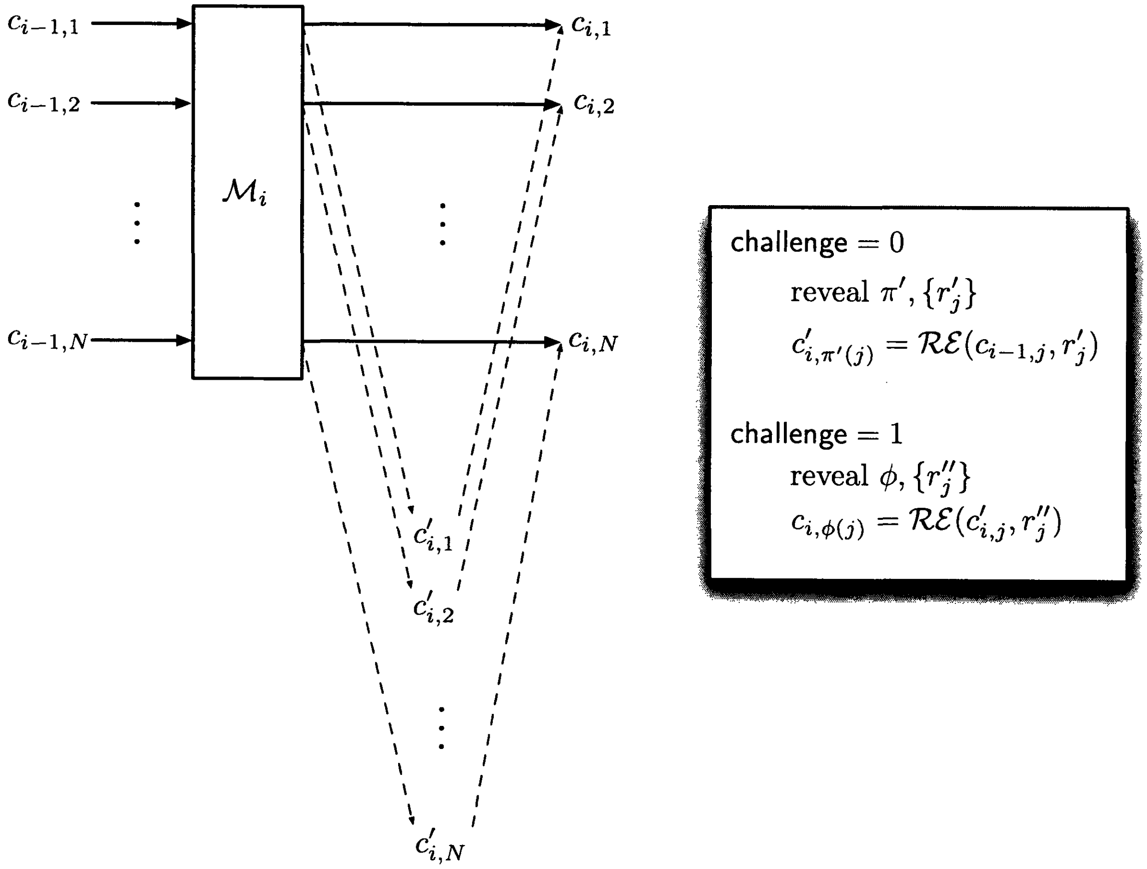
1. a proof of correct partial decryption
2. a proof of correct reencryption and shuffling

The proof of correct partial decryption is straight-forward: given $c = (\alpha, \beta)$, the partial decryption $\mathsf{PartialDec}(c) = (\alpha', \beta')$ effectively yields $\beta/\beta' = \alpha^{x_i}$ where x_i is the mix server's secret key. The mix server must then prove that $(g, y, \alpha, \beta/\beta')$ forms a DDH tuple, meaning that $\log_g(y) = \log_\alpha(\beta/\beta') \bmod p$. Kilian and Sako propose a simple 1-out-of-2, 3-round, cut-and-choose protocol with soundness 50%, though one could also use the Chaum-Pedersen algebraic proof [37] which provides overwhelming soundness. Sako and Kilian point out that the individual proofs can be combined into one, by having the verifier raise all α_j and β_j/β'_j to a random power e_j, multiplying them all together and proving a equality of discrete log on the respective α_j and $\beta_j\beta'_j$ products.

Design Principle 4 (Batch Proof of Knowledge of Randomization Values) *Proving reencryption of a sequence of homomorphic ciphertexts can be batched: the verifier provides a random vector of plaintexts; both prover and verifier compute the homomorphic combination of each ciphertext sequence with this vector challenge, and then proves knowledge of the single randomization value between the first and second dot-product ciphertexts.*

The proof of shuffling is then a fairly typical zero-knowledge proof. Consider π and (r_j) the permutation and randomization values used by a given mix server. The mix server then generates another permutation λ and list of randomization values (t_j), and performs the reencryption and shuffling according to these new parameters, generating what we call the "secondary shuffle outputs." The verifier can then challenge the mix server to reveal either $(\lambda, (t_j))$, which proves that this second mixing was done correctly, or $(\lambda \circ \pi^{-1}, (r_j - t_j))$, which lets the verifier check how the primary shuffle outputs can be obtained by permuting and reencrypting the secondary shuffle outputs. With 50% soundness, this is an honest-verifier zero-knowledge proof of correct shuffling: interaction transcripts are clearly simulatable.

Design Principle 5 (Proof of Shuffle by Secondary Shuffle) *A prover generates a secondary shuffle of the inputs, with independent permutation and randomization values. Based on the verifier's one-bit challenge, the prover reveals either the witness for this secondary shuffle, or the "difference" between the primary and secondary shuffles: the relative permutation and randomization values.*



challenge = 0

reveal π', $\{r'_j\}$

$c'_{i,\pi'(j)} = \mathcal{RE}(c_{i-1,j}, r'_j)$

challenge = 1

reveal ϕ, $\{r''_j\}$

$c_{i,\phi(j)} = \mathcal{RE}(c'_{i,j}, r''_j)$

Figure 3-2: Sako-Kilian Zero-Knowledge Proof of Shuffle. This diagram represents the shuffle phase of the Sako-Kilian proof, after the partial decryption.

Sako and Kilian suggest using Gennaro's techniques for independent broadcast [72] to defeat the related-input attack (Attack 1). This technique requires posters to provide a non-interactive zero-knowledge proof of the plaintext of their posted ciphertext at the beginning of the mixnet.

Countermeasure 3 (Proof of Knowledge of Plaintext) *Each sender provides a non-interactive proof of knowledge of her plaintext message along with her encrypted input.*

Possible Problems. Michels and Horster [118] point out that, if only one mix server is honest, the privacy of all inputs can be compromised. This runs counter to the security model put forth—that if at least one mix server is honest, then privacy is protected.

The attack succeeds because mix servers publish the partially decrypted and unshuffled ciphertexts. Assuming exactly one honest mix server, the $l-1$ remaining *corrupt* mix servers can simply decrypt the partially decrypted ciphertexts. They know the remaining permutation, of course, since they are all colluding, and the single permutation they don't know is unnecessary for the attack, since it hasn't yet been applied.

Attack 3 (Michels-Horster Partial Knowledge Privacy Attack) *Using partial knowledge revealed in the normal mixing process by an honest mix server, corrupt mix servers may be able to cancel out the mixing actions of the lone honest mix server.*

Note that one apparently unpublished countermeasure to this scheme is to simply reverse the mix server's actions: first shuffle and reencrypt, then partially decrypt. Then, if one mix server is honest, the other mix servers can still certainly decrypt the outputs of the honest mix server, but they do not know the permutation contributed by this honest mix server and are thus unable to link the plaintexts to their original senders.

3.4.2 Fault tolerance

In 1997, Ogata et al. suggested two related schemes to provide fault-tolerance: if some of the mix servers abort or misbehave (no more than half), they can be excluded from the anonymization process dynamically, so that the mixing can continue [127]. These protocols use proof techniques similar to those of Sako and Kilian (Design Principle 5), with some added secret-sharing tricks to enable fault tolerance.

Secret-Sharing the Inputs. In their first proposal, Ogata et al. suggest use a a reencrypt-*and*-decrypt mixnet (Design Principle 3), using the additive-homomorphic cryptosystem of Benaloh et al. [43, 19]. Instead of single ciphertext inputs, senders are expected to submit encrypted shares of their input, where each share is encrypted with a different mix server's public key. Secret-sharing is performed using a Shamir polynomial k-out-of-l scheme [159], and Benaloh et al.'s homomorphic secret sharing variant is used for reencryption: a 0-valued polynomial is homomorphically added into each input.

Each mix server thus shuffles and reencrypts the sets of shares. The shuffle proof is based on Design Principle 5: the mix server generates a secondary mixing of the inputs, then reveals, depending on the verifier's challenge bit, either the entire secondary shuffle or the difference between the primary and secondary shuffles. Decryption is then performed by having each mix server publish the decryption of its designated share within each mixnet output. Only k out of the l shares in each output are required to obtain a full decryption.

Design Principle 6 (Fault Tolerance by Secret-Shared Inputs) *Using an appropriate secret sharing scheme and homomorphic cryptosystem, mixnet inputs can be secret-shared such that reencryption is possible by homomorphic combination with a secret-shared identity element. Threshold decryption is thus achieved: a quorum of decryption servers recovers enough shares to produce the fully decrypted outputs.*

Secret-Sharing the Decryption Server Keys. Ogata et al.'s second protocol builds on the reencryption-*then*-decrypt protocol of Park et al. with El Gamal encryption (Design Principle 2). To achieve fault tolerance, each decryption server distributes shares of its secret key to the other decryption servers before mixing begins. At the end of the shuffling stage, if any server refuses to properly decrypt, the others collaborate to decrypt the outputs in its place. It is clear that this principle could apply just as well to the original Sako-Kilian proposal [151].

Design Principle 7 (Fault Tolerance by Secret-Shared Decryption Server Keys) *In a reencryption mixnet, decryption server keys are secret-shared among the decryption servers. If some decryption servers refuse to cooperate in decryption, a quorum of remaining honest decryption servers can recover this server's secret key and perform decryption in its place.*

Foiling relation attacks. Ogata et al. propose having senders give a zero-knowledge proof of knowledge of their plaintext at mixnet input submission time, which we already saw as Countermeasure 3. In addition, in the reencryption-mixnet protocol, each decryption server proves, in zero-knowledge, that it correctly proved its share of the decryption work. Though the precise scheme isn't mentioned, a straight-forward Chaum-Pedersen proof is a good candidate. These measures counter the related-input attacks (Attack 1) by ensuring that no inputs can be based on other inputs unless the other input's plaintext is already known by the adversary.

Countermeasure 4 (ZK PoK of Decryption Key) *Decryption servers provide a zero-knowledge proof of correct decryption, likely via a Chaum-Pedersen proof of correct DDH tuple.*

3.4.3 Verification independent of number of mix servers

In 1998, Abe [1] proposed an extension of Sako and Kilian's mixnet proof system (Design Principle 5) to reduce the amount of work required of the verifier and make it independent of the number of mix servers. To achieve this result, the provers perform some additional work to provide a "joint proof" of their actions, which the verifier can check as if it had been generated by a single mix server. As a result of this construction, this new design is not easily made fault-tolerant.

Each mix server $\mathcal{M}_i$ holds a k-out-of-l share x_i of an El Gamal secret key x, in a proper El Gamal setting as per countermeasure 1. All inputs $c_{0,j}$ are provided with zero-knowledge proofs of knowledge of their plaintext, as per countermeasure 3. Rerandomization and shuffling is performed using El Gamal reencryption, using Design Principle 2. Abe's key contribution is to chain all server proofs so that the verifier need only check the output of the last mix server.

Proving Shuffling. To prove the reencryption and shuffling step, the mix servers perform a secondary mixing (Design Principle 5). The variation in the Abe protocol is that each secondary mix, rather than being based off the mix server's input (which is the prior mix server's primary mix), is based on the previous mix server's secondary mix. If the verifier challenges this secondary mix, then all mix servers reveal their secondary randomization values and permutations simultaneously, using a commitment scheme to enable this simultaneous reveal. If the verifier challenges the difference between the primary and secondary mixes, the mix servers compute this different in sequence, each mix server revealing the difference in turn. In either case, the verifier need only check a single output: the composed secondary shuffle, or the difference between the composed primary and secondary shuffles.

Design Principle 8 (Chained Secondary Shuffle Proof) *Each mix server, in turn, provides a secondary shuffle of the prior mix server's secondary shuffle output. Depending on the verifier's challenge, the mix servers reveal either the mixnet-wide secondary shuffle or the mixnet-wide difference between the primary and secondary shuffle. Either can be computed sequentially by the mix servers, and the verifier only verifies a single secondary shuffle or a single shuffle difference per challenge bit.*

Proving Decryption. Decryption is performed using a slightly modified threshold El Gamal decryption, where the share owners coordinate their actions to reduce the verifier's work. The mix servers within the quorum build up, in turn, the decryption factor for each El Gamal mixnet output $c_{l,j} = (\alpha_j, \beta_j)$. Specifically, $\mathcal{M}_1$ produces $\gamma_{j,1} = \alpha_j^{x_1 L_1}$, where x_1 is the private key share and L_1 its Lagrange interpolation factor. $\mathcal{M}_1$ then produces $\gamma_{j,2} = \gamma_{j,1} \cdot \alpha_j^{x_2 L_2}$, and so on until the quorum of mix servers has effectively produced $\gamma_j = \alpha^x$, which can be used to immediately decrypt $c_{l,j}$.

Like the shuffle proof, the decryption proof can be verified in time independent of the number of mix servers. The mix servers sequentially produce a joint Chaum-Pedersen proof of correct DDH tuple for each mix server output: the servers sequentially and privately contribute to the committed randomness for the first step of the proof and to the third step of the proof, depending on a single verifier challenge. Note that the soundness of this proof is overwhelmingly high without repetition. Figure 3-4 illustrates this process.

Design Principle 9 (Chained Generation of Decryption Factor) *The decryption servers cooperate to produce the joint decryption factor. They also produce a chained Chaum-Pedersen proof of knowledge that this joint decryption factor was produced correctly.*

3.4.4 Verifying Any Mixnet with RPC

In 2002, Jakobsson, Juels, and Rivest [96] introduced Randomized Partial Checking (RPC), a generic mixnet proof system independent of the underlying cryptosystem or shuffling mechanism. This proof system is particularly interesting because of its simplicity: at a high level, each mix server reveals a random half of its input/output correspondences. Thus, in the context of voting, a mix server can corrupt v votes with probability 2^{-v}: corrupting any significant subset of the votes is highly likely to yield detection. Privacy is ensured either probabilistically—the chance that an end-to-end input/output path is revealed can be made quite low with a couple dozen mix servers—or by careful selection of audited correspondences.

More precisely, when mix server $\mathcal{M}_i$ provides outputs $c_{i,j}$, it also provides commitments to values $(j, \pi_i(j), r_{i,j})$ where $r_{i,j}$ is the randomization value that allows anyone to check that, according to the underlying cryptosystem, input $c_{i-1,j}$ was sent to output $c_{i,\pi_i(j)}$ (in short, it is a rerandomization witness). Once this permutation and the mixing outputs are published, the verifier issues the challenge set of inputs (or outputs), and $\mathcal{M}_i$ reveals the appropriate commitments. The verifier can then check that mixing of these inputs (or outputs) was done correctly $c_{i-1,j}$, $c_{i,\pi_i(j)}$, and $r_{i,j}$.

When selecting the proper input (or output) challenges, one must consider the possibility that a complete input-to-output path will be revealed, thus violating some privacy. This issue can be resolved by pairing the mix servers consecutively, $\mathcal{M}_1$ and $\mathcal{M}_2$, $\mathcal{M}_3$ and $\mathcal{M}_4$, etc ... One then randomly selects a half-and-half partition of the values in the middle of each pair, challenging the first mix server of each pair with the first set of the partition, and the second mix server with the second set of the partition. Thus, one forcefully introduces breaks in potential revelation paths. Assuming that fewer than half of the mix servers are malicious, at least one such pair of mix servers is completely honest, thus guaranteeing the privacy of every input.

Design Principle 10 (Randomized Partial Checking) *The mix servers each reveal a random half of their permutation according to a verifier challenge. Privacy is protected either probabilistically—given enough mix servers, no complete path from start to finish is revealed—or by careful selection of the revelations: mix servers are paired, and each pair forces discontinuities in every revealed path. A slightly relaxed definition of soundness (a small number of inputs can be incorrectly mixed) is achieved with overwhelming probability.*

Slightly Weakened Soundness. RPC tolerates a mix server replacing x inputs with probability 2^{-x}. This is a slightly weaker version of soundness than that which we've considered for other mixnets. However, in the voting setting, this is clearly acceptable: any discovered flaw would lead to significant investigation and prosecution, and the probability of having a noticeable impact on the result without getting caught is extremely low.

Improving the Voting Setting. In the voting setting, each mix server is likely to be run by some political party, and each voter is likely to trust only one of those parties (if even that). It has since been noticed, by Chaum [40], that Randomized Partial Checking can be made to accommodate this trust model: each political party runs two consecutive mix servers, thus forming a pair all by itself. Thus, if any single party is honest, the privacy of all votes is ensured.

3.5 Efficient Proofs

In the late 1990s, the literature turned to the goal of achieving *efficient mixnet proofs*, where robustness and universal verifiability could be accomplished within practical running times on tens of thousands of inputs. Rather than use generic zero-knowledge techniques, these new proofs all made use of specific number theoretic assumptions of the underlying cryptosystem, usually El Gamal. In particular, where prior proposals required repeating the entire proof to achieve reasonable soundness, these new proposals provide overwhelming soundness in a single run.

3.5.1 Mixing with permutation networks

In 1999, two independent papers concurrently proposed efficient universally verifiable mixnets using permutation networks: Abe on the one hand [2], and Juels and Jakobsson on the other [94]. Let us name Abe's scheme Permix, and let us use Juels and Jakobsson's name of Millimix. These two papers share a number of techniques, with a handful of unique tweaks on either side. Here, we describe the core ideas of both papers, and point out the interesting unique aspects of each.

Permutation networks. A permutation network (also called a sorting network) is a circuit with N inputs and N outputs which can implement any N-permutation using, as its building block, 2-by-2 sorters. With a recursive design and assuming, for simplicity's sake, that N is a power of 2, such a sorting network can be achieved using $N \log_2 N$ sorters. Such a network is shown in Figure 3-7. If one can build an efficient 2-by-2 cryptographic sorter that hides whether or not it flipped its inputs, then such a construction can lead to a simple and relatively efficient mixer for small input batches.

Design Principle 11 (Mixing with a Sorting Network) *Inputs are passed through a sorting network. Implementing a proof of 2-by-2 mixing is then sufficient to build a complete mixnet.*

Mixnet inputs. Both schemes require mixnet inputs to be semantically secure El Gamal ciphertexts (Countermeasure 1), threshold-secret-shared and encrypted using the mix servers' public keys with threshold k (Design Principle 6). Permix additional suggests that inputs to the whole mixnet include a Schnorr signature [157] to ensure non-malleability (Countermeasure 2). As usual, we denote the inputs to the mixnet as $c_{0,j}$, and, more generally, the inputs to mix server $\mathcal{M}_i$ as $c_{i-1,j}$.

A cryptographic 2-by-2 sorter. Permix and Millimix implement the 2-by-2 sorter in similar ways. Consider inputs $c_0 = \mathcal{E}(m_0) = (\alpha_0, \beta_0)$ and $c_1 = \mathcal{E}(m_1) = (\alpha_1, \beta_1)$, and outputs $c'_0 = \mathcal{E}(m'_0) = (\alpha'_0, \beta'_0)$ and $c'_1 = \mathcal{E}(m'_1) = (\alpha'_1, \beta'_1)$. The mix server responsible for this particular 2-by-2 sorter will compute the outputs as:

$$\begin{aligned} c'_b &= \mathcal{RE}(c_0, r_0) = (\alpha_0 \cdot g^{r_0}, \beta_0 \cdot y^{r_0}) \\ c'_{\bar{b}} &= \mathcal{RE}(c_1, r_1) = (\alpha_1 \cdot g^{r_1}, \beta_1 \cdot y^{r_1}) \end{aligned}$$

Depending on the choice of b, the mix server either passed the inputs straight to the outputs, or flipped them. The reencryption and DDH assumption of El Gamal ensure that no observer can tell b with better than random chance.

Both Permix and Millimix suggest proving correct operation of this 2-by-2 sorter using a CDS disjunctive proof [44] of plaintext equality: either $m_0 = m'_0$ and $m_1 = m'_1$, *or* $m_1 = m'_0$ and $m_0 = m'_1$. The witness-hiding property of CDS ensures privacy. Millimix notes that one can optimize this proof so that there is only one disjunctive proof and one normal proof of plaintext equivalence: $m_0 = m'_0$ *or* $m_0 = m'_1$, *and* $m_0 m_1 = m'_0 m'_1$. Given El Gamal's multiplicative homomorphism, the verifier can easily compute the encryption of the two products $m_0 m_1$ and $m'_0 m'_1$ without interaction.

The exact proof of plaintext equivalence differs between Permix and Millimix. Permix suggests using the Chaum-Pedersen proof [37] of El Gamal plaintext equivalence. Millimix notes the following trick, which transforms knowledge of a rerandomization value into a Schnorr identification protocol:

- given two El Gamal ciphertexts $c = (\alpha, \beta)$ and $c' = \mathcal{RE}(c', \gamma) = (\alpha', \beta')$,
- for a random z, denote $G = gy^z$ and $Y = (\alpha'/\alpha)(\beta'/\beta)^z$,
- note how $Y = G^\gamma$,
- anyone who knows γ can perform a Schnorr signature using public key (G, Y).

This proof protocol for plaintext equivalence is more efficient than Chaum-Pedersen by about 50%.

Robustness. In both Permix and Millimix, robustness is ensured by Design Principle 6: performing threshold decryption *after* mixing. Honest mix servers agree to perform their share of the decryption only if all individual mix proofs are correct. Thus, with a properly parameterized secret-sharing scheme, any level of robustness can be achieved.

Privacy. In Millimix, each mix server performs shuffling on an entire permutation network of its own. Thus, there are l entire permutation networks, and, if at least one mix server is honest, privacy is ensured. In Permix, the setup is slightly more complicated, but also more flexible: assuming at most k adversarial mix servers, Permix uses $k + 1$ permutation networks, assigning 2-by-2 sorter gates such that any given mix server $\mathcal{M}_i$ is assigned gates from a single permutation network. Thus, by a counting argument, at least one permutation network is handled by only honest mix server. When k is maximal, i.e. $k = l - 1$, the Permix architecture is the same as Millimix.

Reencrypt-*and*-Decrypt. Abe also suggests a reencrypt-*and*-decrypt version of his permutation-network-based mixnet. Let us call it Decmix. In this scheme, $k+1$ permutation networks are used, and every column j is assigned a simple additive share x_j of the secret key x. A single mix server controls one or more consecutive columns within a single permutation network, thus having access to the corresponding x_j. Each mix server performs partial decryption and reencryption (Component 3).

To prove correct operation of a 2-by-2 sorter that reencrypts *and* partially decrypts, the mix server responsible for that sorter's column provides a disjunctive proof of knowledge of the appropriate reencryption factors in the case of flip or non-flip.

Design Principle 12 (Privacy-Protecting Mix Server Set Assignments) *Mix servers can be carefully assigned to mixing sets so that, by a counting argument, there is always one honest mix server per set, and always one set composed of entirely honest mix servers. Then, if each such set fully mixes the inputs, correctness is ensured by the existence of the fully honest set.*

3.5.2 Matrix-based proof

Both Millimix and Permix provide better concrete proof speeds for small to medium batches compared to the previous, generic cut-and-choose proofs. In 2001, Furukawa and Sako introduced a proof more efficient than Millimix, Permix, and all prior schemes, both for practical batch sizes and asymptotic measures [70]. This scheme frames the mixing process as a matrix multiplication, then proves in zero knowledge that the matrix in question is a permutation matrix.

As the protocol is fairly involved, we leave the details to the original paper, and provide a general outline. Consider a single mix server $\mathcal{M}_i$, with inputs (α_j, β_j) and outputs (α'_j, β'_j).

Properties of a Permutation Matrix. Consider a m-by-n matrix A_{ij} with elements in $\mathbf{Z}_q$. Consider $P_{jj'} = \sum_{i=1}^{m} A_{ij}A_{ij'}$, effectively the dot product of columns j and j'. Consider $P_{jj'j''} = \sum_{i=1}^{m} A_{ij}A_{ij'}A_{ij''}$, effectively the "3-way" dot-product of columns j, j', and j''. A_{ij} is a permutation matrix if and only if:

- **Property 1:** $P_{jj'} = 1$ if $j = j'$, and $P_{jj'} = 0$ otherwise.
- **Property 2:** $P_{jj'j''} = 1$ if $j = j' = j''$, and $P_{jj'j''} = 0$ otherwise.

Proof Overview. The FS proof decomposes the actions of a reencrypting mix server as follows:

$$(\alpha'_j, \beta'_j) = (g^{r_j} \prod_{i=0}^{N-1} \alpha_i^{A_{ij}}, y^{r_j} \prod_{i=0}^{N-1} \beta_i^{A_{ij}}) \tag{3.1}$$

The Furukawa-Sako proof then contains four subproofs:

1. The matrix A_{ij} that relates (α_j) to (α'_j) has **property 1**, as defined above.
2. The matrix A_{ij} that relates (α_j) to (α'_j) has **property 2**, as defined above.
3. The lists of (r_j) in the equations proven in the last two steps are the same.
4. For each output (α'_j, β'_j), the same r_j and matrix (A_{ij}) were used in the first and second half of the pair. This task is handled by the anonymous e-cash protocol of Brands [27].

Design Principle 13 (Proof of Matrix Multiplication by a Permutation Matrix) *Each mix server demonstrates knowledge of a matrix and randomization values that relate the input ciphertexts to the output ciphertexts, proving in the process that this matrix is a permutation matrix.*

Performance. This scheme provides a proof in linear time on the number of inputs, specifically requiring $18N$ exponentiations.

FS is a ZK Argument. One important note about the Furukawa-Sako protocol is that it is a zero-knowledge *argument*, as soundness depends on the hardness of the discrete logarithm assumption.

3.5.3 Exponent dot-product proof

At around the same time as Furukawa and Sako, Neff [124] introduced what remains to this day the fastest, fully-private, universally verifiable mixnet shuffle proof, requiring $8N$ exponentiations (not counting typical bulk modexp optimizations). Like the Furukawa-Sako proof, the Neff proof is fairly complex. We leave the details to the original paper, and provide a high-level outline here.

Neff decomposes the actions of mix server $\mathcal{M}_i$ in the following way, for input j and some permutation π:

$$(\alpha'_j, \beta'_j) = (g^{r_{\pi(j)}} \alpha_j, y^{s_{\pi(j)}} \beta_j)$$

Note that this equation can be satisfied for *any* set of outputs, including bad outputs. The central idea of the Neff proof is that mixing correctness occurs when $\boldsymbol{r} = \boldsymbol{s}$. Once the inputs and outputs of the mix server are fixed, given a random challenge vector $\boldsymbol{t}$, $\boldsymbol{r} \cdot \boldsymbol{t} = \boldsymbol{s} \cdot \boldsymbol{t}$ implies that $\boldsymbol{r} = \boldsymbol{s}$ with overwhelming probability. The Neff proof requires 3 protocols, each of which is very briefly described here.

Equality of Exponent Products. Consider a vector of k elements (A_i), and another vector (B_i), all elements of our usual q-order subgroup of $\mathbf{Z}_p$ with chosen generator g. Consider $a_i = \log_g A_i$ and $b_i = \log_g B_i$. Neff defines an interactive, zero-knowledge proof, $\mathsf{EqualExponents}$, which:

- given public inputs (A_i) and (B_i),
- given private inputs (a_i) and (b_i),
- proves that $\prod a_i = \prod b_i$.

The details of $\mathsf{EqualExponents}$ can be found in the original paper.

Known-Exponent Shuffle. Consider a vector of k elements (T_i), and another vector (U_i), all elements of our usual q-order subgroup of $\mathbf{Z}_p$ with chosen generator g. Consider $t_i = \log_g T_i$ and $u_i = \log_g U_i$. Assume that $\gamma \in \mathbf{Z}_q, \Gamma = g^\gamma$. Neff defines an interactive, zero-knowledge proof, $\mathsf{KnownExpShuffle}$, which:

- given public inputs $(T_i), (U_i), \Gamma$,
- given private inputs $(t_i), (u_i), \gamma$ and a permutation π,
- proves that $U_i = T^\gamma_{\pi(i)}$.

Given a challenge $w \in \mathbf{Z}_q$ from the verifier, the protocol calls $\mathsf{EqualExponents}$ on:

- $2k$ public inputs
  $(T_1 g^w, T_2 g^w, \ldots, T_k g^w, \Gamma, \ldots, \Gamma)$ and $(U_1 \Gamma^w, U_2 \Gamma^w, \ldots, U_k \Gamma^w, g, \ldots, g)$.
- $2k$ private inputs
  $(t_1 + w, t_2 + w, \ldots, t_k + w, \gamma, \ldots, \gamma)$ and $(u_1 + w\gamma, u_2 + w\gamma, \ldots, u_k + w\gamma, 1, \ldots, 1)$.

Recall that $u_i = \gamma t_{\pi(i)}$. Thus the extra k elements in the proof, $(\Gamma, \ldots, \Gamma)$ and $(g, \ldots, g)$, are added to balance out the γ exponent products in the products. The run of $\mathsf{EqualExponents}$ effectively demonstrates that:

$$\gamma^k \prod_{i=0}^{k-1} (t_i + w) = \prod_{i=0}^{k-1} (u_i + w\gamma)$$

Moving the γ^k inside the product, we're left with:

$$\prod_{i=0}^{k-1} (t_i \gamma + w\gamma) = \prod_{i=0}^{k-1} (u_i + w\gamma)$$

This is effectively the evaluation of two polynomials at a random point w. If the evaluations are equal, then, with overwhelming probability, the polynomials are equal, and there exists a permutation π such that $u_i = \gamma t_{\pi(i)}$.

El-Gamal Shuffle. The proof of shuffle in the Neff Scheme, EGShuffle, proceeds as follows once the mix server has produced outputs (α'_j, β'_j):

- Prover and Verifier engage in a protocol to generate a random vector (T_j) and a random value $\Gamma \in \langle g \rangle$, such that Prover knows $(t_j = \log_g T_j)$, $\gamma = \log_g \Gamma$, and the permutation π used in the actual shuffling. Together, Prover and Verifier compute $(U_j = T^{\gamma}_{\pi(j)})$. Verifier should not learn π, γ, or $(u_j), (t_j)$. This portion of the proof uses KnownExpShuffle as a subprotocol.
- Prover demonstrates knowledge of $R = \sum_{j=1}^{N} r_j t_j$ such that:

$$\log_g \frac{\prod_{j=1}^{N} (\alpha'_j)^{u_j}}{\prod_{j=1}^{N} \alpha_j^{t_j}} = \log_y \frac{\prod_{j=1}^{N} \beta'^{u_j}_j}{\prod_{j=1}^{N} \beta_j^{t_j}} = R\gamma$$

Assuming $a_j = \log_g \alpha_j, a'_j = \log_g \alpha'_j, b_j = \log_y \beta_j, b'_j = \log_y \beta'_j$, the proof shows that:

$$\boldsymbol{a'} \cdot \boldsymbol{u} - \boldsymbol{a} \cdot \boldsymbol{t} = \boldsymbol{b'} \cdot \boldsymbol{u} - \boldsymbol{b} \cdot \boldsymbol{t}$$

If this equation holds, then with overwhelming probability, $a'_i = a_{\pi(i)}$ and $b'_i = b_{\pi(i)}$, and the shuffling is correct.

Design Principle 14 (Proof Equality of Exponent Dot Product) *A mix server commits to a permutation and receives a random challenge vector. It then proves knowledge of the single reencryption exponent between the dot product of the input exponents with this random vector and the dot product of the output exponents with the permuted random vector. The key point is that the single reencryption exponent is the same for both halves of the El Gamal ciphertexts.*

Performance and Soundness. This scheme requires $8N$ exponentiations. It remains, to this day, the fastest proof of a shuffle with overwhelming soundness, even with a computationally-unbounded prover.

Generalization to any homomorphic commitment scheme. Groth [87] generalized Neff's scheme to any homomorphic commitment scheme and homomorphic cryptosystem. He also provided a more complete proof of correctness.

3.6 Proofs by Aggregate Properties

In the development of mixnet schemes, some variants appeared which were eventually shown to be flawed. It is, of course, important to keep track of these attempts, so that we may strive to understand why they were thought to work for a short while. It is also important to consider the techniques proposed, as they may be useful in their own right in other applications, or in slightly weaker security definitions for mixnets. Finally, the attacks against these schemes also yielded interesting new constructions.

In this section, we consider a number of attempts which rely on proving properties of homomorphic aggregations of the inputs and outputs. We provide only a high-level overview and refer the reader to the individual papers for the details, as some of these protocols are quite involved. However, we do provide an intuition for what went wrong and for how the protocols were fixed (if they were). In particular, we point out that, when proving properties on the aggregation of the inputs and outputs, mix servers are not proving knowledge of an actual permutation. It is this "missing piece" which is often at the root of the problem.

3.6.1 A Practical Mix

In 1998, Jakobsson [92] proposed "a practical mix" based on *repetition robustness.* As its name implies, repetition robustness proposes to achieve robust results by repeating the mixing process a number of times. If the results are the same across the various repetitions, then it is overwhelmingly likely that no mix server cheated. If an inconsistency is detected, a tracing process ensues to pinpoint the guilty mix server. It is important to note that this mixnet is not universally verifiable, as a coalition of all mix servers can corrupt the entire election. However, it is an interesting technique in its own right.

Overview. At a high level, the mixing is performed in three major steps: provable distributed homomorphic blinding, threshold decryption, and distributed unblinding-and-shuffling. The mix servers cooperate to jointly blind the plaintexts "through" the encryption layer. Consider specifically the El Gamal setting, where a ciphertext:

$$c = (\alpha, \beta) = (g^r, my^r)$$

is blinded using exponent δ:

$$(\alpha^\delta, \beta^\delta) = (g^{r\delta}, m^\delta y^{r\delta}) = \mathcal{E}(m^\delta; r\delta).$$

Then, the mix servers can safely decrypt the resulting ciphertexts without shuffling, since the outputs will be blinded. Finally, in the most involved part of the process, the mix servers cooperate to jointly unblind the resulting plaintexts while shuffling them. This last phase employs repetition robustness to ensure that all mix servers are acting "correctly." The final portion of this last step requires the mix server to prove correct exponentiation and shuffling by proving, in zero-knowledge, that the product of the outputs is simply the product of the inputs raised to the appropriate unblinding exponent.

A First Flaw. One notable flaw of this mixnet, noted by Jakobsson himself in a subsequent paper [93], is that the mix servers see the un-shuffled blinded plaintext messages. In this setting, two identical messages appear identical when blinded. In an election, if the mix servers know some of the plaintext votes, they are then able to correlate the value of many others. One possible countermeasure here is to pad individual votes to make them unique.

3.6.2 Breaking and Fixing a Practical Mix.

The Attack. In 2000, Desmedt and Kurosawa [55] showed how to break Jakobsson's scheme. Because verification requires only a check on the input and output *products*, a malicious mix server can cheat the system by producing, instead of the honestly blinded and shuffled inputs, a different set of outputs whose product is the same as the product of the would-be honest outputs. Desmedt and Kurosawa show, by way of example, how the last mix server in particular can completely corrupt the set of outputs while passing all verification tests, using as a basis the original, non-anonymized inputs to the first mix server.

Attack 4 (Mixing Cancellation Attack) *When mix servers prove knowledge of some aggregate property instead of the actual permutation, it may be possible for one mix server to "cancel" the effects of the mixing by prior mix servers, by processing as its input an earlier batch and adjusting one of its output to match the desired aggregate. For example, the last mix server can break the privacy of the entire mixing by using, as its input batch, the original mixnet inputs.*

Fixing Repetition Robustness. To patch this attack, Desmedt and Kurosawa introduced a new method of verification. Though this method is not universally verifiable, it is quite interesting in that it is the precursor to Wikström universal composability proof (see Section 3.8). Given l mix servers, the protocol tolerates strictly fewer than $\sqrt{l}$ corrupt mix servers.

The mix servers are partitioned into blocks of size $\sqrt{l}$, of which there are $\sqrt{l}$. Given the assumption that the are fewer than $\sqrt{l}$ corrupt mix servers, each block has at least one honest mix server, and at least one block is composed entirely of honest participants. Each block designates a "shuffler" who performs El Gamal reencryption and shuffling (Design Principle 2). This shuffler privately sends his permutation and randomization values to the other members of the block. Any verifier mix server can pinpoint an error if one occurs. If any errors are declared against a block, then its output is ignored. Since there is at least one block with entirely honest mix servers, at least one block will provide an output without error. This output can then be threshold-decrypted appropriately.

Design Principle 15 (Existential Honesty) *The mix servers are partitioned into blocks to ensure that at least one block is fully honest and no block is fully corrupt. The output of a block is then correct only if it is fully honest. An un-challenged block output can then be threshold-decrypted.*

3.6.3 Flash Mixing

Shortly after "A Practical Mix," Jakobsson proposed Flash Mixing [93], which, though published a year before Desmedt and Kurosawa's mixing cancellation attack (Attack 4), was resistant to it. Like Jakobsson's previous mix, Flash mixing relies on repetition robustness, proving aggregate properties of input/output relationships with repetitions to ensure that deviations will be caught.

The protocol proceeds as follows:

- Two dummies are introduced into the input list, such that no mix server knows the plaintexts of these dummies.
- The resulting list of mixnet inputs, which are El Gamal ciphertexts in the style of Park et al. (Design Principle 2), is copied a number of times, precisely for repetition purposes.
- The mix servers in turn shuffle and reencrypt each of these lists, producing a first shuffle, for the purpose of hiding the position of the dummies.
- The mix servers then produce a second shuffle starting with the output of the first shuffle with independent randomization values and permutations. Note that, at this point, they do not know where the dummies are.
- The mix servers then reveal all of their secret data pertaining to the first shuffle, thereby revealing where the dummies ended up before the second shuffle began.
- The mix servers in turn then proceed to prove the second shuffle:
  - reveal the permutation of the two dummies, including the randomization value for the first dummy, but only a zero-knowledge proof of knowledge of the randomization value for the second dummy.
  - prove in zero-knowledge that the homomorphic product of the inputs equals the homomorphic product of the outputs.
  - reveal the difference between the permutations and randomization values of the mix server's first output list and the other copies of the list. This can be done by a mix server only once it knows the relative differences for the prior mix server.

The purpose of the list repetition is to force a cheating mix server to somehow guess the relative permutations of its inputs in order to successfully modify a subset of them. The dummies are meant to ensure that any mischievous modification to the ciphertexts by a mix server must somehow manage to correctly guess the relative positions of the two dummies in all repeated lists.

Breaking and Fixing Flash Mixing. The Desmedt-Kurosawa attack on A Practical Mix does not work against Flash Mixing as is, because the dummy values prevent the last mix server from simply taking the first mix server's input directly. However, Mitomo and Kurosawa [119] describe a related attack that does succeed against Flash Mixing.

The key idea of this attack is to cancel out the hiding of the dummy values from the first reencryption. Assume the first server is malicious. It performs the first shuffle of all lists correctly. However, in the second shuffle, it produces N corrupt outputs and the two dummies reencrypted from their initial input positions before the first shuffle, $N+1$ and $N+2$. The corrupt N outputs are computed such that their homomorphic product is the homomorphic product of the second shuffle inputs, homomorphically divided by the two dummies from the very beginning. Then, once the first shuffle randomization values and permutations are revealed in the verification of the first shuffle, the first mix server knows how to prove the aggregate randomization value for the input and outputs products, the randomization values for the dummy values, and the relative permutations and randomization values. This attack works because the actions of the first shuffle can be effectively negated, since all of the raw witnesses are revealed before the second shuffle proofs begin.

Attack 5 (Dummy Shuffle Cancelation) *An attacker can cancel the effect of the first reencryption shuffle in Flash Mixing by producing its second reencryption output based on aggregating the mixnet's pre-shuffle inputs, rather than the output of the first reencryption. The effect of the dummy values is negated because their positions before the first shuffle are known to all.*

Mitomo and Kurosawa propose a quick countermeasure to their own attack. They accomplish this by adding a number of zero-knowledge proofs rather than simply revealing raw randomization values. As expected, this change makes the protocol significantly less efficient, which negates the biggest advantage of Jakobsson's original approach.

Countermeasure 5 (Dummy Shuffle Cancellation Countermeasure) *One should trace the dummy values through the first shuffle using zero-knowledge proofs of knowledge, and verifying the products in the second reencryption, all before the first shuffle reencryption values are revealed. This technique forces mix servers to prove knowledge of the reencryption factors for the dummy values without knowing the reencryption values for the first shuffle, there by defeating the cancellation attack.*

3.6.4 Optimistic Mixing

In 2002, Golle et al. [84] proposed a new type of universally verifiable mixnet they called "Optimistic Mixing," with a significantly faster proof when all players behave honestly, and a fallback to existing proofs like Neff's when an error is detected. "Optimistic Mixing" uses El Gamal reencryption, properly parameterized as per Countermeasure 1.

In order to check correctness before the plaintexts are fully revealed, two layers of encryption are used: first the message is encrypted in the usual manner:

$$c = \mathcal{E}(m; r) = (\alpha, \beta) = (g^r, my^r)$$

Then, c is cryptographically hashed, and all components are encrypted in a second layer:

$$d = (d_1, d_2, d_3) = \Big(\mathcal{E}(\alpha; r'), \mathcal{E}(\beta; s'); \mathcal{E}(H(\alpha, \beta); t')\Big)$$

Every message submitted as input to the mixnet comes with a proof of knowledge of $(\alpha, \beta, H(\alpha, \beta))$. The mix servers then reencrypt and shuffle these triples of ciphertexts in the usual manner. Upon completion, a threshold decryption yields $(\alpha', \beta', \gamma')$, and everyone checks that $\gamma' = H(\alpha', \beta')$ for all messages. In addition, using the homomorphic multiplication propery of El Gamal, each mix server proves correct reencryption of corresponding input and output *products*: the α's, β's, and cryptographic hashes. If this proof succeeds, the mix servers agree to decrypt the inner layer, thereby recovering the plaintexts.

The idea of this construction is that the cryptographic checksum should prevent a malicious mix server from successfully altering the outputs without also altering the products.

3.6.5 Wikström Attacks

In 2003, Wikström [178] presented 5 serious attacks, some against Optimistic Mixing, and some against Flash Mixing . These attacks are a compilation of various prior techniques along with some novel ideas. Taken together, these attacks deliver a significant blow to the optimized techniques of repetition robustness and double-enveloping.

Correlation Attacks. In the spirit of the related-input attack (Attack 1), Wikström notes that optimistic mixing requires proofs of knowledge of the $(\alpha, \beta, H(\alpha, \beta))$ one-layer-deep values only, not of the two-layer-deep plaintext m. In addition, the same cryptosystem and public key are used for both layers. As a result, an attacker Eve can roughly take the encrypted mixnet input of an honest user Alice, wrap it in an additional layer of encryption, and provide this as an input to the mixnet with a valid proof of knowledge. The normal mixnet process will decrypt this triple-encrypted message twice, thus revealing the single-encryption of Alice's input, which should never be correlated with Alice's identity. Yet Eve can match this against the single-layer ciphertexts on the bulletin board, and follow the second threshold decryption to discover what Alice's plaintext input was.

Attack 6 (Related Inputs on Double Envelopes with Extra Wrapping) *If inputs are double-enveloped ciphertexts with only a partial proof—i.e. a proof of the inner layer only—related inputs, complete with the requisite proofs, can be constructed by adding an extra wrapping around an existing double-wrapped input.*

If two different keys are used for the inner and outer encryption layers, there remains an attack if the cryptographic keys aren't renewed from one mix to another and if the first mix server is malicious. In a first instance of the mix, the first mix server corrupts two honest inputs by swapping parts of the two ciphertexts. The product proofs remain valid, though the hash check fails, causing a tracing procedure which effectively reveals to everyone what the inner ciphertexts were for Alice and Bob, the two targeted users. The malicious mix server is found and eliminated, but can return as a submitter in a second mixing instance, where he submits outer-encrypted versions of inner encryptions of Alice and Bob found in the previous round. This is now another correlation attack on Alice and Bob: when the messages are fully decrypted, the mix server learns the original plaintexts submitted by Alice and Bob in the first mix.

Attack 7 (Related Inputs on Double Envelopes with Tracing-Based Unwrapping) *When a tracing procedure reveals the inner-ciphertext of a double-enveloped mixnet input, an adversary can force a trace in a first mixnet instance, then reuse this inner ciphertext to create a related input in a second mixnet instance.*

Mixing Cancellation Attack. Wikström offers two additional attacks, where one or more malicious mix servers cancel the effects of mixing. Like in the original, these attacks are possible because each mix server only proves plaintext product equality between its inputs and outputs, rather than knowledge of the specific permutation.

In the first such attack, exactly in the spirit of Desmedt and Kurosawa's attack (Attack 4), the last mix server is corrupt. Instead of rerandomizing and shuffling its input, it shuffles and rerandomizes the *first mix server's inputs*, which are identified by sender on the bulletin board. By tweaking one output to cancel out all of the randomization values accumulated throughout the mixing, the malicious mix server succeeds at its proof. The outputs are then decrypted normally, and the attack only affects the outer layer of reencryption. The last mix server then can correlate any output with its sender, since the only effective shuffling is the one it applied itself.

The second mixing cancellation is a variant of Pfitzmann's original semantic security attack (Attack 2). The first and last mix server are corrupt. Together, they wish to discover Alice's plaintext message. The two mix servers agree on a "tag" value t that is *not* in the El Gamal q-order subgroup. The first mix server multiplies Alice's input by t, and another input by t^{-1} to ensure product equality. The last mix server then tests all of its inputs for the subgroup "tag," removes the tag and its counterpart, and publishes its outputs, knowing exactly where the tagged input ended up. Upon decryption the malicious mix servers learn Alice's plaintext. This attack succeeds because only the first batch of inputs is checked for proper semantic security constraints and only aggregate integrity conditions are checked between inputs and outputs.

Attack 8 (Semantic Security of Intermediate Ciphertexts) *When mix servers only provide proofs of aggregate properties on their ciphertexts, the properties checked on the first batch of inputs may not be automatically preserved throughout the various shuffles. If the starting mix server can "tag" a ciphertext by moving away from the semantic security constraints (and canceling out this deviation for the aggregate check), then the last mix server can recognize this tag and effectively "ignore" the mixing performed in between.*

Delayed Effect Attack. Wikström describes a final attack, where an adversary that controls one mix server and two senders can change his mind on the effective plaintext of the two senders after they have submitted their inputs. This can be used, for example in a voting setting, to decide how one wishes to vote depending on an event that happens after the close of voting but before the mixing begins. In certain settings, this can be problematic.

Wikström's attack again relies on the fact that only an aggregate property of the inputs and outputs is proven by mix servers, and that a trace of certain inputs can be considered benign if the error comes from the sender rather than from a mix server. The adversarial senders provide coordinate invalid inputs:

$$\mathcal{E}(\alpha_0), \mathcal{E}(a\beta_0), \mathcal{E}(H(\alpha_0, \beta_0)) \text{ and } \mathcal{E}(\alpha_1), \mathcal{E}(a^{-1}\beta_1), \mathcal{E}(H(\alpha_1, \beta_1))$$

The first mix server can then choose to "correct" these inputs by homomorphically removing the a and a^{-1}, which doesn't alter the products and thus lets the mix server prove the shuffle even with this substitution. If it chooses not to correct the inputs, they will simply be detected as benign mistakes caused by sender error. Thus, the adversary can delay his decision as to whether the two inputs by these senders should count or not.

Attack 9 (Delayed Decision) *If mix servers only prove aggregate properties, then an attacker can provide incorrect inputs whose aggregate is the same as the aggregate if these inputs were legitimate. The first mix server can then decide, after the close of input submission, whether or not to rehabilitate these inputs.*

3.7 Variants

When working with real-world problems that require mixnets, a number of practical constraints need to be addressed. Beyond the theoretical proof of correctness, a number of techniques have been introduced to address these constraints.

3.7.1 Hybrid Mixnets

When using a public-key cryptosystem like El Gamal, one limitation is that each ciphertext input to the mixnet can only be as big as the plaintext space of the El-Gamal public key allows, usually a few hundred bits or, in the case of elliptic-curve based El-Gamal, no more than 300 bits. This plaintext space can become quickly insufficient, and one immediate idea is to employ a hybrid encryption scheme that combines the asymmetric nature of public-key cryptography and the length flexibility of symmetric encryption. Two interesting schemes exist in this category, though neither is universally verifiable. In fact, there are no known efficient universally verifiable hybrid mixnets, although one could well imagine adapting RPC to these hybrid mixnets.

Length-Invariant Hybrid Mix. In 2000, Ohkubo and Abe [128] introduced the first robust hybrid mixnet. They prove the security of their construction in the Random Oracle Model under the Decisional Diffie Helman (DDH) assumption. They use a novel combination of El Gamal encryption and symmetric encryption, with an ideal hash function mapping El-Gamal group elements to symmetric keys.

The mix servers $\{\mathcal{M}_i\}_{i\in[1,l]}$ select secrets $a_i, x_i \in \mathbf{Z}_q^2$, then compute and publish:

$$(h_i, y_i) = (g^{\prod_{j=1}^{i} a_j}, h_i^{x_i})$$

The generation of these keys is sequential, starting with bootstrap value $a_0 = g$. Mix server $\mathcal{M}_i$ receives h_{i-1} from the prior mix server $\mathcal{M}_{i-1}$, computes $h_i = h_{i-1}^{a_i}$, then $y_i = h_i^{x_i}$. Effectively, each mix server $\mathcal{M}_i$ publishes an El Gamal public key y_i with generator h_i, with h_i dependent on the previous mix server's key.

To send a message into the mixnet, a sender generates $r \in \mathbf{Z}_q$ and computes symmetric keys $K_i = H(y_i^r)$ using each mix server's public key y_i. The plaintext m is then chain-encrypted as $m' = \mathcal{E}_{K_1}(\mathcal{E}_{K_2}(\ldots(m)\ldots))$, and the sender sends $(\alpha_0, \beta_0) = (g^r, m')$ to the bulletin board. The mix servers then sequentially recover y_i^r, extract K_i, and each unwrap one layer of the symmetric encryption:

$$\begin{aligned} \alpha_i &= \alpha_{i-1}^{a_i} = h_i^r \\ K_i &= H(\alpha_i^{x_i}) = H(y_i^r) \\ \beta_i &= \mathcal{D}_{K_i}(\beta_{i-1}) \end{aligned}$$

Mix server $\mathcal{M}_i$ then publish the (α_i, β_i) values in random order.

Design Principle 16 (Chained Symmetric Key Scheduling) *Mix servers publish chained El Gamal public keys. Senders can use these chained public keys to schedule symmetric keys—one for each mix server—using a hash function on a randomized encryption of 1 under that mix server's public key. The mix servers can sequentially recover their specific encryption of 1 through sequential partial decryption, yielding the appropriate secret key for decryption at each step.*

Ohkubo and Abe also suggest extensions to make their mixnet robust against malicious senders and mix servers. First, using Countermeasure 2, they propose a Fiat-Shamir style non-interactive zero-knowledge proof of knowledge of the randomization exponent r based on (α_0, β_0). Then, they suggest using the method of Desmedt and Kurosawa (Design Principle 15) to partition the servers into $\sqrt{l}$ groups of size $\sqrt{l}$. Assuming at most $\sqrt{l} - 1$ mix servers are malicious, each group has at least one honest server, and one group must be entirely honest. By distributing all keying material for a given server to all other members of the server's group, we ensure that any deviation is detected, since there is at least one honest server per group, and we ensure that the shuffle permutation remains secret, because there is at least one group with entirely honest servers who do not leak their shuffle permutation.

Optimally Robust Hybrid Mix. In 2001, Jakobsson and Juels [95] proposed a hybrid mixnet with optimal robustness: up to half of the mix servers can be corrupt, rather than $\sqrt{l}$ in Ohkubo and Abe's proposal. However, the length of the inputs is now linear in the number of mix servers, rather than invariant.

Their proposal uses ideas similar to those of Ohkubo and Abe: each mix server chains an El Gamal public key from the prior mix server, which enables a kind of "key scheduling" of unique keys for each mix server using a single random exponent. In addition, Jakobsson and Juels insert a MAC (message authentication code) within every layer of the symmetric key wrapping. This additional MAC causes the length of the ciphertext to grow linearly with the number of mix servers, while allowing a mix server to determine if the prior mix server cheated. A mix server must also prove a product equality relationship between its inputs and outputs, which is unlikely to be malleable given the MACs.

Design Principle 17 (Chained Symmetric Key Scheduling with MACs) *Mix servers and senders prepare the same chained setup with key scheduling as in Design Principle 16. In addition, senders provide a MAC within each layer of symmetric encryption, using a MAC key generated from a similar but independent chain-and-partial-decrypt operation. Each mix server can thus verify that the prior mix server was honest, and can publish evidence to the contrary if it wasn't.*

3.7.2 Universal Reencryption

In 2004, Golle et al. [82] proposed universal reencryption, a mixnet technique which does not require that mix servers know the public key of the ciphertexts they are mixing. This approach is useful in that it does not require the mix servers to be part of the key generation process. Golle et al. mention that this may be useful in the case of anonymizing RFID tags, where RFID tag values may be opportunistically reencrypted by various mix servers to prevent tracking. Of course, this kind of mixnet only performs reencryption, not decryption.

The basic construction for universal reencryption is simply to have a ciphertext input for plaintext m be a pair of El Gamal ciphertexts $(\mathcal{E}_{pk}(m), \mathcal{E}_{pk}(1))$. Using the homomorphic property of El Gamal, it is then trivial to reencrypt this ciphertext. To do so, pick reencryption exponents $r, s \in \mathbf{Z}_q$, and output $(\mathcal{E}_{pk}(m) \otimes \mathcal{E}_{pk}(1)^r, \mathcal{E}_{pk}(1)^s)$. Golle et al. consider *semantic security under reencryption* to prove the security of this scheme, a property which, in the case of El Gamal, follows from the DDH assumption.

Design Principle 18 (Universal Reencryption) *Given a homomorphic cryptosystem, a sender's input is a pair of ciphertexts: the encryption of the identity, and the encryption of the desired message. The mix server can use the encryption of the identity element to reencrypt both ciphertexts.*

3.7.3 Almost Entirely Correct Mixing

In 2002, Boneh and Golle [26] proposed a technique for rapidly proving almost correct mixing of El Gamal reencryption mixnets. The proof implies correctness with high, but not overwhelming probability, and it may leak some information about which inputs correspond to which outputs. However, the proof is quite fast and the technique fairly interesting.

The basic idea is to challenge the mix server with a subset of the inputs, such that each input is chosen with independent probability 1/2. The mix server must provide an equally-sized subset of the outputs, then prove knowledge of the rerandomization value between the homomorphic product of the chosen inputs and the homomorphic product of the corresponding outputs. This test can be repeated α times. Boneh and Golle prove that, if all challenges succeed, the mix server performed correctly with probability $(1 - (5/8)^{\alpha})$.

Though the protocol is intuitive, the proof is fairly involved. At a high level, Boneh and Golle collect, via extraction, a number of input-set/output-set pairs from the prover, including the randomization value. If the mix server isn't mixing properly, then they eventually find some non-trivial multiplicative identity on the input plaintexts. Boneh and Golle assume randomly distributed plaintexts, and reduce solving the discrete logarithm problem to this extraction by plugging in specially-crafted plaintexts, derived from a discrete-log challenge pair (g, h).

Design Principle 19 (Repeated Verification of Homomorphic Aggregations) *Using the homomorphic property of the cryptosystem, a verifier can request aggregate permutation information on a large subset of the inputs, where the prover only demonstrates knowledge of the reencryption between the respective homomorphic aggregations of the selected inputs and corresponding outputs.*

3.7.4 Parallel Mixing

In 2004, Golle and Juels [83] introduced parallel mixing. Golle and Juels note that practically all mixnet protocols require serialized mixing: one mix server mixes while all others wait. Instead, they propose a scheme where the set of inputs is split into batches, so that all mix servers can mix in parallel.

To provide enough "mixing," each mix server shuffles its batch, then partitions and redistributes the result to *all other mix servers.* To protect against malicious mix servers, a number of rotations are added between redistributions: each mix server shuffles, then the batches are each passed to another mix server in a circular order (without redistribution), and shuffled again. The number of redistributions depends on the number of mix servers, while the number of rotations depends on the number of assumed corrupt mix servers.

The details of the proof of security are provided in the original paper [83]. They argue that this mixnet provides complete but dependent privacy, and, given enough inputs, almost independent privacy.

Design Principle 20 (Parallel Mixing) *Mix servers shuffle in parallel rather than in series. Repeating shuffling ensures privacy: rounds of redistribution ensure a good enough mix, while rounds of rotation ensure privacy against corrupt mix servers.*

3.8 Universally Composable Mixnets

A mixnet is almost always a component of a larger, more complicated protocol, e.g. voting. As attacks over the years have shown, it is common to see a mixnet proven secure in a particular setting, only to find that it breaks down in a slightly different setting, e.g. if the mixnet is run twice with the same public key. The question, like for many other complex protocols, is how to precisely determine what the true security properties of a mixnet should be.

One approach to solving this problem generically is to prove security in the Universally Composable framework of Canetti [32], which we reviewed in Chapter 2. In this framework, a protocol proven secure can be composed with other secure protocols, even in parallel. In addition, the UC framework forces a high-level analysis of the security properties we desire from a mixnet.

3.8.1 A First Definition and Implementation

In 2004, Wikström [179] provided the first universally composable mixnet definition and implementation. The details of this new protocol are not particularly efficient, but the security proof is particularly interesting.

Defining the Ideal Functionality. Wikström gives $\mathcal{F}_{\mathrm{MN}}$, an ideal functionality for a mixnet. This functionality is particularly simple, relying on the UC baseline to guarantee security: the senders submit their individual input to the ideal functionality. When a majority of the mix servers choose to "mix," the ideal functionality sorts the inputs lexicographically and outputs the resulting list. As is typical in the UC setting, the ideal adversary $\mathcal{S}$ can delay messages to the ideal functionality as it sees fit, though of course it cannot see the communication with the ideal functionality.

Definition 3-1 (Ideal Mixnet) *The ideal functionality for a mixnet, $\mathcal{F}_{\mathrm{MN}}$, with senders $\mathcal{P}_1, \ldots, \mathcal{P}_N$, mix-servers $\mathcal{M}_1, \ldots, \mathcal{M}_l$, and ideal adversary $\mathcal{S}$, proceeds as follows:*

1. *Initialize:*
   - $J_{\mathcal{P}} = \emptyset$*, the set of senders who have sent a message to the ideal functionality,*
   - $J_{\mathcal{M}} = \emptyset$*, the set of mix servers who have sent a message asking to start the mixing.*
   - $L = \emptyset$*, a list of messages received from senders.*

2. *Repeatedly wait for messages:*
   - *on message $(\mathcal{P}_j, \textit{\textbf{Send}}, m_j)$, if $\mathcal{P}_j \notin J_{\mathcal{P}}$ and m_j is properly formed (e.g. it is a proper element in some pre-defined group), set $L \leftarrow L \cup \{m_j\}$ and $J_{\mathcal{P}} \leftarrow J_{\mathcal{P}} \cup \{\mathcal{P}_j\}$. Send $(\mathcal{S}, \mathcal{P}_j, \textit{\textbf{Send}})$.*
   - *on message $(\mathcal{M}_i, \textit{\textbf{Run}})$, if $\mathcal{M}_i \notin J_{\mathcal{M}}$, then set $J_{\mathcal{M}} \leftarrow J_{\mathcal{M}} \cup \{\mathcal{M}_i\}$.*
     If $|J_{\mathcal{M}}| \geq \lceil l/2 \rceil$, then prepare list L' as the lexicographically sorted version of L, and send $\{(\mathcal{P}_j, \texttt{\textbf{Output}}, L')\}_{j \in [1,N]}, \{(\mathcal{M}_i, \texttt{\textbf{Output}}, L')\}_{i \in [1,l]}$. Stop responding messages.
     Otherwise, send $(\mathcal{S}, \mathcal{M}_i, \textit{\textbf{Run}})$.

Wikström also gives UC definitions for an ideal bulletin board and an ideal distributed El Gamal key generation, suggesting implementations of the former by Lindell, Lysyanskaya and Rabin [110] and of the latter by Gennaro et al. [74]. We leave the details of these additional components to their respective publications and to Wikström's paper.

Wikström uses Canetti's UC definitions [32] for generic definitions of zero-knowledge proofs of knowledge of a relation, in this case zero-knowledge proofs of knowledge of input plaintexts and a zero-knowledge proof of knowledge of the randomization values and permutation. For both of these modules, Wikström provides implementations.

Implementation Techniques. Wikström's mixnet uses the El Gamal cryptosystem. The zero-knowledge proof of knowledge of plaintext used by senders uses a modified version of Feldman's verifiable secret sharing [64], where senders secret-share their witness to all mix servers by encrypting each share with the target mix server's public key and posting the result on the authenticated bulletin board. Each mix server decrypts and verifies his share and posts the result of this verification to the bulletin board for all to see.

For the proof of proper shuffle, Wikström proposes a variant of the Desmedt and Kurosawa scheme (Design Principle 15). The l mix servers are partitioned into sets, such that any given set has at least one honest mix server and at least one set is entirely honest. Mix servers then perform a number of reencryptions and shuffles, posting the witnesses on the bulletin board encrypted under a subset of the other mix servers' public keys. In contrast with the original Desmedt-Kurosawa proof, Wikström proof requires that this process be repeated a certain number of times, each time with a different partition of the verifiers. This prevents malicious verifiers from conducting a denial-of-service attack, and ensures extractability for the UC ideal adversary.

3.8.2 Sender-Verifiable Mixnet

In 2005, Wikström [180] proposed a slightly different approach to mixnet verification, *sender verifiability*, where a *sender* can verify that her input is correctly making its way through the mix servers. This new protocol significantly reduces the size of the per-mix-server witness and introduces a novel technique for proving a correct shuffle, provable in the UC framework. Here, we review the sender-verifiable technique and give a brief overview of the proof concepts. We leave the details of the proof protocol to the original paper [180], as they require much technical subtlety. As the author notes, sender verifiability may be problematic when dealing with coercion in a voting setting. However, even if one is willing to do away with sender verifiability, Wikström techniques remain quite interesting, given the short mix server witness and the complete determinism of the shuffling operation.

The Need for Reencryption? Wikström notes that El Gamal mixnets typically use reencryption because, when a plaintext is encrypted with El Gamal into ciphertext (α, β) with joint public key $Y = \prod_{i=1}^{l} y_i$, partial decryption yields $(\alpha, \alpha^{-x_i}\beta)$, with the α value unchanged. Clearly, if this were the only alteration between input and output ciphertexts, the permutation would leak by simple comparison of the input and output α values. Thus, in Park et al.'s original paper on reencryption mixes, one proposal is to partially decrypt *and* reencrypt at the same time. This then leads to the realization that the mix servers can focus on the reencryption in order to simplify the proofs, leaving the shared decryption as a subsequent step.

Indistinguishable Partial Decryption. Instead, Wikström proposes a variant of El Gamal where partial decryption yields an indistinguishable ciphertext under the DDH assumption. Each public key share pk_i defines an additional generator h_i, with the secret key sk_i augmented to include the discrete logarithm of h_i base g.

Thus, a secret key in Wiström-El-Gamal is $sk = (w, x) \in \mathbf{Z}_q^2$, and the corresponding public key is $pk = (h = g^w, y = g^x)$. Encryption is then:

$$\mathcal{E}_{pk}(m; r) = (\alpha, \beta) = (h^r, my^r)$$

and decryption is

$$\mathcal{D}_{sk}(\alpha, \beta) = \alpha^{-x/w}\beta$$

Most interestingly, consider a joint public key formed recursively, with each server sequentially injecting its public key into the joint key:

$$\begin{aligned} \mathsf{PK}_{l+1} = (H_{l+1}, Y_{l+1}) &= (g, 1) \\ \mathsf{PK}_i = (H_i, Y_i) &= (H_{i+1}^{w_i}, Y_{i+1}H_{i+1}^{x_i}) \end{aligned}$$

Thus, a message is encrypted under the joint public key produced by the first mix server $\mathcal{M}_1$ after all other mix servers have injected their contribution:

$$c_0 = \mathcal{E}_{\mathsf{PK}_1}(m; r) = (\alpha_0, \beta_0) = (H_1^r, mY_1^r)$$

Server i can then partially decrypt as follows:

$$\begin{aligned} c_i = (\alpha_i, \beta_i) = \mathcal{D}_{sk_i}(\alpha_{i-1}, \beta_{i-1}) &\overset{\text{def}}{=} (\alpha_{i-1}^{1/w_i}, \alpha_{i-1}^{-x_i/w_i}\beta_{i-1}) \\ &= (H_i^{r/w_i}, H_i^{-rx_i/w_i} mY_i^r) \\ &= (H_{i+1}^r, mY_{i+1}^r) \\ &= \mathcal{E}_{\mathsf{PK}_{i+1}}(m; r) \end{aligned}$$

Thus, the output of the last server is $c_l = \mathcal{E}_{\mathsf{PK}_{l+1}}(m; r) = (g^r, m)$, and g^r can be discarded to yield the plaintext m. Note the following equalities for all $i \in [1, l]$

$$\alpha_i = \alpha_{i-1}^{1/w_i} \tag{3.2}$$

$$\frac{\beta_{i-1}}{\beta_i} = \alpha_i^{x_i} \tag{3.3}$$

Note that the DDH assumption ensures that, if one of two ciphertexts is partially decrypted, no adversary can tell which one it was. Thus, in the mixnet setting, each mix server $\mathcal{M}_i$ takes inputs $(c_{i-1,1}, \dots, c_{i-1,N})$ and produces outputs $\big(\mathcal{D}_{sk_i}(c_{i,\pi_i(1)}), \dots, \mathcal{D}_{sk_i}(c_{i,\pi_i(N)})\big)$, without reencryption, with permutation π_i selected so as to produce a lexicographically sorted list.

Design Principle 21 (El Gamal with Indistinguishable Partial Decryption) *The El Gamal cryptosystem can be augmented so that each secret key includes an additional secret w and encryption is performed using generator g^w rather than g. Thus, partial decryption of such a ciphertext requires changes to both elements of a ciphertext pair, and makes the partial decryption operation indistinguishable.*

Proof of Partial-Decryption and Permutation. In this new setting, there is only a short witness that mix server $\mathcal{M}_i$ must prove knowledge of: the secret key (w_i, x_i). Wikström proposes a new proof technique to accomplish this task. At a high level, the prover and verifier interact as follows:

- Prover commits to its permutation π in the typical El Gamal q-order subgroup, by providing element-wise commitments to a permutation of a random vector with elements in the q-order subgroup.

- Verifier provides a challenge of distinct primes $(p_i)_{i \in [1,N]}$, all of the same bit-length.

- Prover commits to a permutation of the primes $(p_{\pi(i)})_{i \in [1,N]}$ in an appropriately sized RSA group, then proves that $\prod_{i=1}^{N} p_i = \prod_{i=1}^{N} p_{\pi(i)}$ and $\sum_{i=1}^{N} p_i = \sum_{i=1}^{N} p_{\pi(i)}$. The sizing of the RSA group ensures that these two proofs in fact demonstrate that $(p_{\pi(i)})_{i \in [1,N]}$ is indeed a permutation of $(p_i)_{i \in [1,N]}$.

- Prover effectively proves the equations 3.2 and 3.3 using equality of exponent dot products of inputs and outputs with the random vector of primes (similar to Neff's technique, Design Principle 14.)

Wikström proves this construction secure in the UC framework, using his prior mixnet ideal functionality definition. He also shows how his dual RSA/El-Gamal technique of permuting a random challenge of primes can be applied to other mixnet constructions that perform reencryption.

Design Principle 22 (Equality of Exponent Dot Product with Vector of Primes) *A mix server performs a proof of a shuffle with exponent dot product, similar to Design Principle 14, using a random vector of primes and a commitment to a shuffle of these primes according to the original permutation. The random vector of primes is particularly useful for proving appropriate shuffling: using an appropriately sized RSA group, demonstrating equality of element-wise sums and products is enough to demonstrate proper permuting of the test vector.*

3.8.3 Adaptively Secure Mixnet

In 2006, Wikström and Groth [181] gave the first adaptively secure mixnet and prove its security in the UC framework. Recall that, in this adaptive setting, parties to the protocol may become corrupt in the middle of the process. This presents significant complications when proving the UC security of a protocol, as the ideal simulator must be able to simulate an honest participant's complete history when this participant becomes corrupt. Some protocols assume that participants can erase critical information as they go, in order to give the adversary less power once it corrupts the participant: the ideal adversary only needs to simulate the data that was *not* erased. Here, Wikström and Groth present a protocol that *does not require erasures.* We describe *very briefly* the two techniques they use to solve these complications. We leave the technical details to the original paper, as they are very involved and require much additional reading.

Mixnet Protocol and Simulation Difficulties. Wikström and Groth use a variant of the Paillier cryptosystem [133], with the usual additive homomorphism and thus reencryption ability. Senders provide ciphertext inputs with a proof of knowledge of the plaintext. Each mix server reencrypts and shuffles its inputs. Then, all mix servers jointly reencrypt the last batch, which the mix servers finally decrypt jointly. This is the adaptation of Design Principle 2 to the Paillier cryptosystem. Two complicating scenarios arise:

1. when a sender is corrupted after having already produced the ciphertext and proof of knowledge of plaintext, and

2. when a mix server is corrupted after having shuffled its batch of inputs.

These scenarios provide a simulation problem in the UC proof of security: the ideal adversary has to fake the actions of honest senders and mix servers without knowing their inputs, but then, at corruption time, produce a history consistent with the now-revealed inputs and the already published actions, like the proof of knowledge of plaintext for senders and the proof of knowledge of randomization values and permutation for the mix servers.

Adaptively Corrupt Senders. The special Paillier variant introduced by Wikström and Groth enables a double-ciphertext construction similar to that of Naor and Yung [122]. This construction provides "straight-line extractability": the ideal adversary can generate *one* of the two keypairs, allowing it to extract the plaintexts without rewinding. This property is crucial to proving security in the UC framework.

More importantly, the special form of this Paillier variant allows an ideal adversary that has generated part of the keying material to produce a special ciphertext that it can open up—along with a valid simulated history tape – to any plaintext at a later date. This property lets the ideal advesary simulate adaptively corrupt senders in the mixnet setting: once corruption occurs and the real input is revealed, the ideal adversary simulates the proper history tape for that input and the bogus ciphertext it initially produced but can no longer "retract."

Design Principle 23 (Double-Ciphertext Construction) *A Naor-Yung double-ciphertext construction for mixnet inputs provides straight-line extractability, which enables the simulation of a proper history tape in the case of adaptively corrupted senders in the UC framework.*

Adaptively Corrupt Mix Servers. Wikström and Groth also introduce a novel, last batch joint reencryption step, which is key to providing simulatability of adaptively corrupt mix servers. Specifically, when a mix server is corrupted after it has shuffled its inputs but before the ideal functionality has revealed the final ordering of the plaintexts, the ideal adversary suddenly learns the real permutation input to the now-corrupt mix server. In the joint reencryption step, the ideal adversary can use the special properties of the Paillier variant ciphertext and its simulation of the keying material to cheat and "swap" the ciphertexts as needed to correct for the prescribed permutation.

Design Principle 24 (Final Joint Reencryption) *The mix servers perform a final joint reencryption step before decryption. This step enables simulation of a proper history tape for adaptively corrupted mix servers in the UC framework.*

3.9 Summary

In this section, we provide a summary table of the mixnets we described, their basic properties, and the known attacks against them. This table should help to quickly determine whether a mixnet protocol has been broken.

3.10 Conclusion

The mixnet research field is rich with fascinating contributions to secure cryptosystems, efficient zero-knowledge proofs, and secure protocol definitions. With the efficient proofs

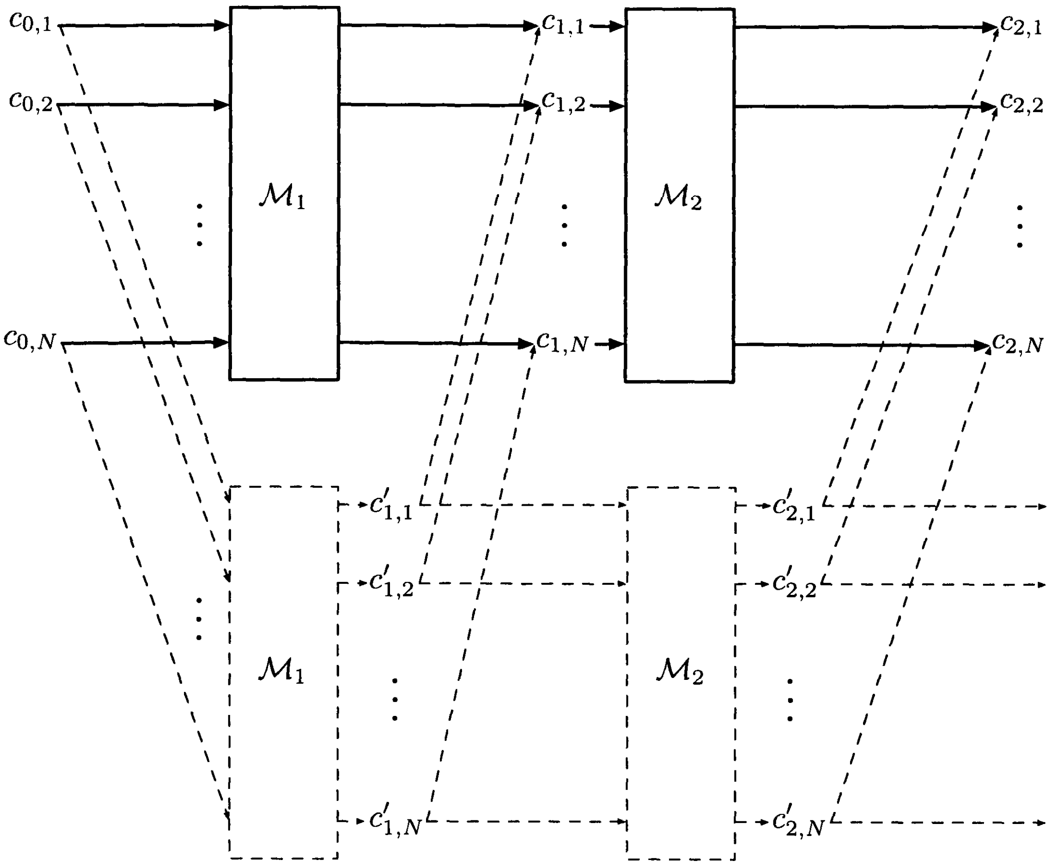
Mixnet Scheme	Based On	Attacks & Fixes	Fault-Tolerant	Verif.	Soundness	Privacy
Chaum Onions [39]	-	[138]	No	Sender	-	C&I
Park et al. partial dec. [134]	-	[136, 151]	No	Sender	-	C&I
Park et al. reenc. [134]	-	[136, 151]	No	Sender	-	C&I
Sako and Kilian [151]	[134]	[118], this review	No	**Univ.**	OW Proof	C&I
Ogata et al. [127]	[134, 151]	-	Yes	**Univ.**	OW Proof	C&I
Abe [1]	[134, 151]	-	Yes	**Univ.**	OW Proof	C&I
Jakobsson, Juels, Rivest [96]	[39]	-	-	**Univ.**	OW Proof †	C&I
Permutation Networks [2, 94]	[134]	-	Yes	**Univ.**	OW Proof	C&I
Furukawa & Sako [70]	[134]	-	Yes	**Univ.**	OW Arg	C&I
Neff [124]	[134]	-	Yes	**Univ.**	OW Proof	C&I
A Practical Mix [92]	[134]	[93, 55]	Yes	Quorum	-	C&I
Flash Mixing [93]	[134, 92]	[119, 178]	Yes	Quorum	-	C&I
Optimistic Mixing [84]	[134, 92]	[178]	Yes	**Univ.**	OW Arg	C&I
Almost Correct Mixing [26]	[134]	-	Yes	**Univ.**	HP Proof (α-dep.)	Incomplete
Parallel Mixing [83]	[134]	-	Yes	-*	-*	C & D
UC Mixnet [179]	[134, 55]	-	Yes	**UC**	OW Proof	C&I
Sender-Verifiable Mixnet [180]	[134, 179]	-	Yes	**UC**	OW Proof	C&I
Adaptively-Secure Mixnet [181]	[179]	-	Yes	**UC**	OW Proof	C&I

* These characteristics depend on the subprotocol chosen, which is independent of Parallel Mixing.

† Randomized Partial Checking has a slightly weaker definition of soundness, though one sufficient for voting.

Table 3.1: Summary of Mixnet Protocols. Each mixnet protocol is listed with the prior protocols from which it inherits, the papers that present flaws, the papers that present fixes, the verifiability and privacy properties. The protocols are ordered chronologically. The indicated privacy and soundness are indicated as per the *originally claimed* values, not the result of any potential breaks. We do not include the hybrid mixnets or universal reencryption in this comparison, as they are qualitatively different. C&I stands for "Complete and Indepenent," while C&D stands for "Complete and Dependent."

of Neff [124] and Furukawa and Sako [70], the efficiency of mixnets has become practical. With the UC modeling work of Wikström, mixnets now have a solid formal model for solid security proofs. It remains to be seen, in the future, whether efficient implementations like Neff's can be adapted to frameworks like Universal Composability.



challenge $= 0$

reveal $\pi'_1, \ldots, \pi'_l$

reveal $\{r'_{1,j}\}, \ldots, \{r'_{l,j}\}$

$c'_{i,\pi'_i(j)} = \mathcal{RE}(c'_{i-1,j}, r'_{i,j})$

challenge $= 1$

reveal $\phi_1, \ldots, \phi_l$

reveal $\{r''_{1,j}\}, \ldots, \{r''_{l,j}\}$

$c_{i,\phi_i(j)} = \mathcal{RE}(c'_{i,j}, r''_{i,j})$

Figure 3-3: Abe Zero-Knowledge Proof of Shuffle. The secondary shuffle is represented in dotted lines. Each mix server's secondary shuffle is dependent on the prior mix server's secondary shuffle, rather than the primary shuffle as in the Sako-Kilian proof.

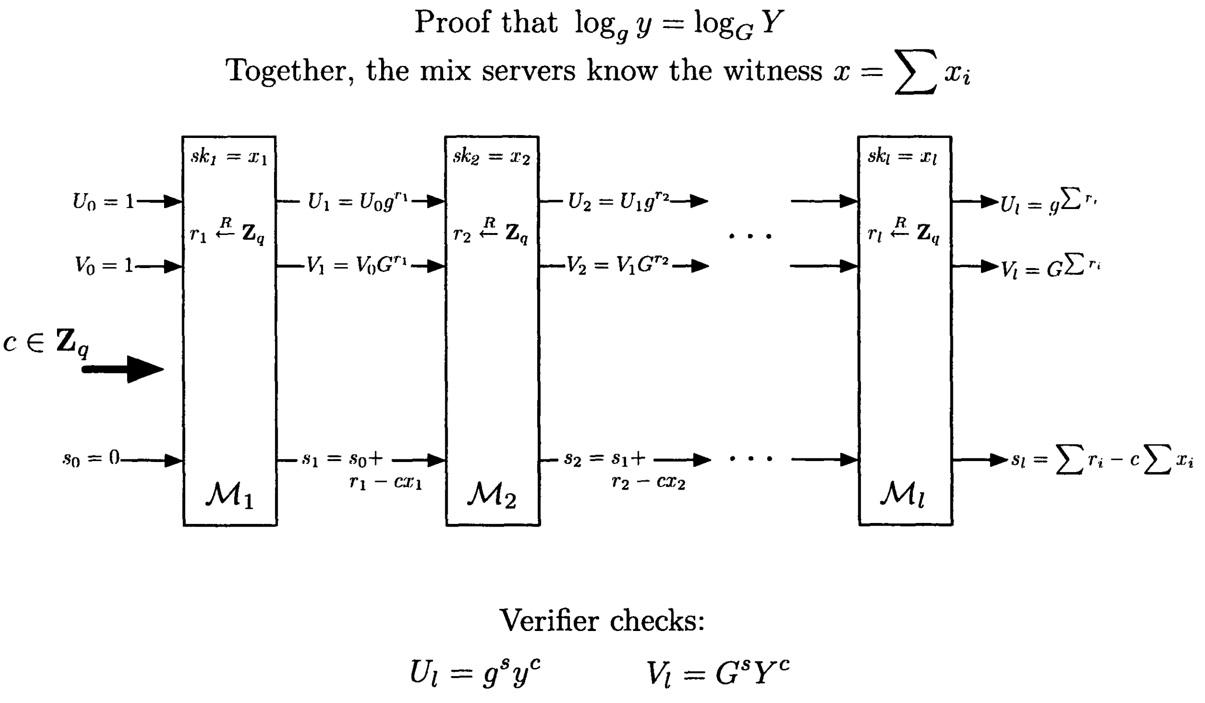


Figure 3-4: Abe's Chained Chaum-Pedersen Proof of Joint Decryption. The typical Chaum-Pedersen exchanges are passed through all the mix servers. For simplicity the diagram leaves out the Lagrange interpolation suggested by Abe, and shows only one (g, y, G, Y) tuple. Abe shows how to process *all* decryptions with one r_i per mix server, effectively performing all proofs in parallel.

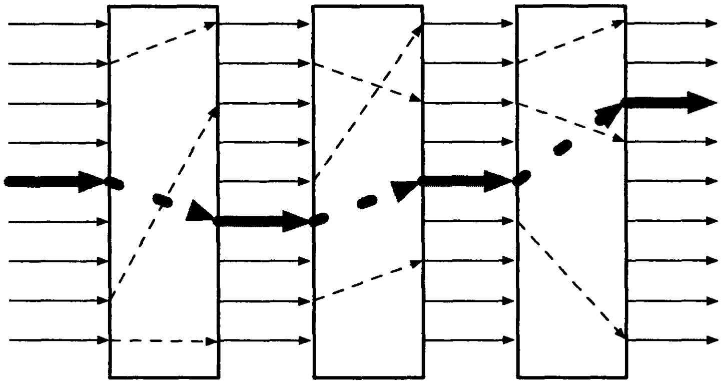


Figure 3-5: Randomized Partial Checking. Each mix server reveals a random half of its correspondences. Shown in red is a fully revealed path, which can happen if the number of mix servers is small and the verification selection is completely random.

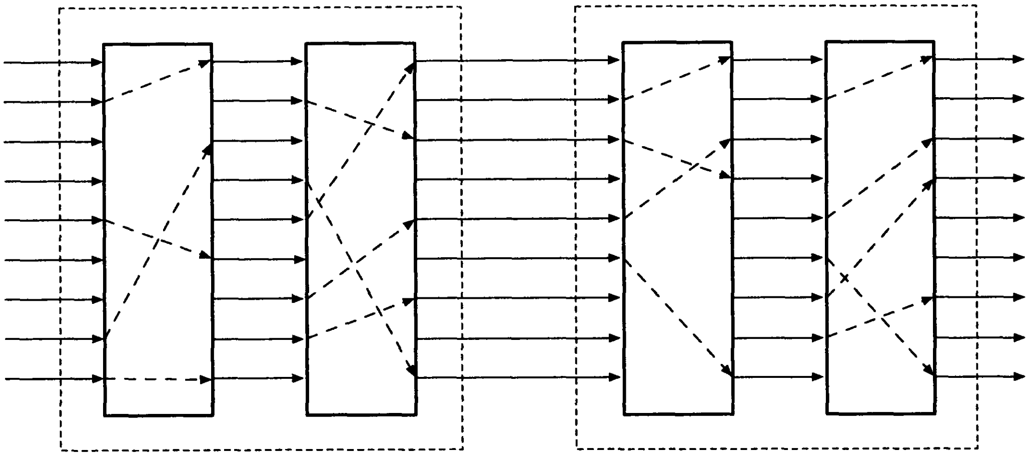


Figure 3-6: Randomized Partial Checking with Pairing. Mix servers are sequentially paired. The ciphertexts "between" both members of each pair are randomly partitioned. The first mix server reveals the corresponding inputs of its block, while the second mix server reveals the corresponding outputs of its block. The partition ensures that no complete path from first input to last output is revealed.

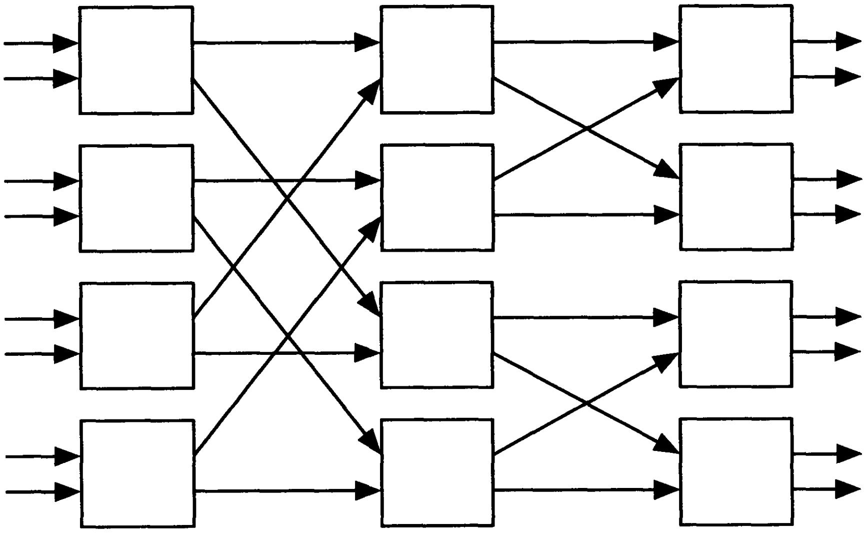


Figure 3-7: A Sorting Network. Composed of 2-by-2 sorters, this sorting network accomplishes any permutation with an optimal number of sorters. This is called a Butterfly Network.

Chapter 4

Scratch & Vote

This chapter covers work to appear at the Workshop on Privacy in the Electronic Society, in October 2006 [18]. This is joint work with Ronald L. Rivest.

4.1 Introduction

Cryptography can reconcile public auditability and ballot secrecy in voting. Votes are encrypted and posted on a public bulletin board, along with the voter's name in plaintext. Everyone can see that Alice has voted, though, of course, not what she voted for. The encrypted votes are then anonymized and tallied using publicly verifiable techniques.

Most cryptographic voting schemes require complex equipment and auditing. A certain degree of complexity is unavoidable, as the functional goal of cryptographic voting is to run an election correctly *while trusting third parties as little as possible.* Unfortunately, this complexity often stands in the way of adoption. If it takes significant expertise to understand how a voting system functions, and if the operation of the system is particularly complex, election officials and the public may be reluctant to adopt it. The question, then, is how much can we simplify the voting process while retaining cryptographic verifiability?

Voting systems & scratch surfaces. In recent months, Arizona has proposed running a cash-prize lottery for all citizens who vote [143]. In response, a well-known online satirical publication jokingly proposed a "Scratch & Win" voting system [158]. Though our proposal, Scratch & Vote, uses scratch surfaces, it should not be confused with a game of chance. That said, we hope that, given the public's familiarity with scratch surfaces, our own use of them will help spark more widespread interest in the topic of cryptographic voting.

4.1.1 Scratch & Vote

We propose *Scratch & Vote* (S&V), a cryptographic voting method that provides public election auditability using simple, immediately deployable technology. S&V offers:

1. **Paper ballots**: ballot casting is entirely paper- and pen-based.

2. **Self-contained ballot auditing**: ballots contain all the necessary information for auditing; there is no need to interact with the election officials.

3. **Simple tallying:** ballots are tallied using homomorphic encrypted counters rather than mixnets. Anyone can easily verify the final tally, and election officials need only cooperate to decrypt a single tally ciphertext per race.

The voter experience is simple and mostly familiar:

- **Sign in**: Alice signs in and obtains a ballot with randomized candidate ordering. Election officials should not see this candidate ordering. The ballot is perforated along its vertical midline, with candidate names on the left half and corresponding optical-scan bubbles on the right. A 2D-barcode is positioned just below the checkboxes on the right. A scratch surface labeled "void if scratched" is positioned just below the barcode, and an additional perforation separates the scratch surface from the rest of the right half. (See Figure 4-1.)

- **Audit [optional]**: Alice may select a second ballot for auditing. She scratches off the scratch surface, hands the now void ballot to a helper organization on the premises, i.e. a political party or activist organization she trusts, and receives confirmation that the ballot was well-formed. This gives Alice confidence that her first ballot is also well-formed: if enough voters perform the audit, even a handful of bad ballots will be quickly detected. (See Figure 4-2.)

- **Make selection**: Alice steps into the isolation booth to make and review her selection.

- **Detach ballot halves**: Alice separates the two halves of the ballot. A receptacle is available for her to discard her left ballot half. Note that this discarded half carries no identifying information, only a randomized candidate ordering. (See Figure 4-3.)

- **Casting**: Alice presents the right half of her ballot to an election official, who inspects the scratch surface to ensure it is intact. (Because the left half has been discarded, the election official cannot tell for whom Alice voted.) The official then detaches the scratch surface and discards it in sight of all observers, including Alice herself. Alice then feeds what remains of her ballot—the checkmark and barcode—into an optical scanner. This is effectively her encrypted ballot. Alice takes it home with her as a receipt. (See Figure 4-4.)

- **Verification**: Alice can log on to the election web site to verify that her ballot, including checkbox and barcode, has been correctly uploaded to the bulletin board. If it hasn't, she can complain with receipt in hand. Alice can also verify the entire tally process, including the aggregation of all ballots into a single encrypted tally, and the verifiable decryption performed by election officials. (See Figure 4-5.)

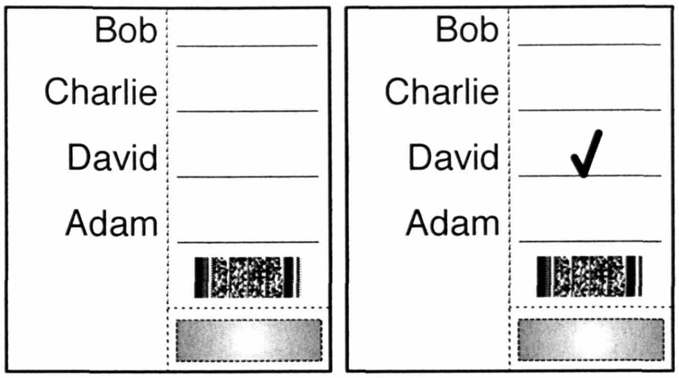


Figure 4-1: A Scratch & Vote ballot, before and after Alice makes her selection. The ballot is perforated along two axes: down the vertical midline, and between the barcode and scratch surface on the right half.

This description uses the Scratch & Vote adaptation of the Ryan Prêt-a-Voter ballot. In section 4.5, we also show how to achieve the same features based on a new ballot inspired by Chaum's Punchscan, whose physical layout accommodates numerous races more easily. We consider the threat model and certain extensions to our proposal that further increase practicality.

4.1.2 Overview of the Ideas

Scratch & Vote combines a number of existing cryptographic voting ideas in a novel way, with some interesting new variations.

Homomorphic tallying. Cryptographic paper ballots do not naturally support write-in votes. Generally, when Alice wants to write in a name, she selects the "write-in" pre-determined pseudo-candidate option, and follows a separate process to specify her candidate. Thus, our first proposal for S&V is to use homomorphic aggregation to simplify the tally for pre-determined candidates, as originally proposed by Benaloh [43, 19] and, more specifically, as extended by Baudron et al. [13] for multi-candidate election systems. This design choice opens the door to further simplifications.

2D-barcode and scratch surface. With homomorphic tallying and pre-determined candidate names, all of the ciphertexts required for one race on one ballot can be fully represented using a single 2D-barcode. The randomization values used to generate these ciphertexts is also printed on the ballot, under a scratch surface. Thus, a ballot is entirely self-contained: by scratching and checking the encryption of the candidate ordering against the ciphertexts in the 2D-barcode, one can immediately verify ballot correctness. This auditing requires only a simple barcode scanner, a basic computer, and the public election parameters.

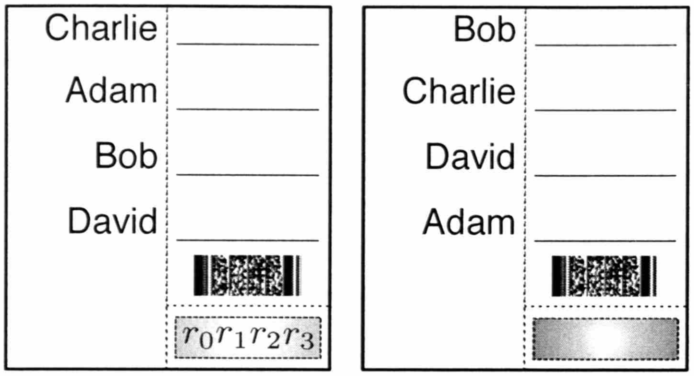


Figure 4-2: Auditing the S&V ballot. Alice receives two ballots and chooses to audit one at random, removing its scratch surface. In this diagram, Alice selects the ballot on the left. Alice's chosen helper organization then scans the barcode, reads the randomization data r_1, r_2, r_3, r_4 (one value per candidate) previously hidden under the scratch surface, and confirms that the ballot is correctly formed. Alice then votes with the second, pristine ballot.

Cut-and-choose at the precinct. Once a ballot is audited, it cannot be used for voting: with its randomization values revealed, the ballot is no longer privacy-protecting. Thus, auditing is used in a cut-and-choose process: each voter may select two ballots, auditing one and voting with the other. The specific advantage of S&V is that this cut-and-choose auditing requires no election official intervention: the ballot and the public election parameters are sufficient. Thus, auditing in S&V is practical enough to be performed live, in front of the voter, *before* she casts her ballot. In addition, local election officials may audit a number of ballots on their own before voting begins: once again, these local election officials only need the public election parameters to successfully audit.

Proofs of correctness and certified ballot list. In a homomorphic tallying system, auditors want assurance that the encrypted ballots contribute no more than one vote per race ; otherwise, a malicious official and voter could collude to artificially inflate a candidate's tally. For this purpose, election officials prepare zero-knowledge proofs of correctness for each official ballot. These proofs are published on the bulletin board for all to see prior to election day, and only ballots whose proofs verify are included in the tally.

As a result of this tallying condition, voters now need assurance that their ballot won't be disqualified at some point after ballot casting. Unfortunately, the sheer size of the proof precludes printing it on the ballot alongside the ciphertexts.

To address this concern, election officials produce a certified ballot list containing ballots that officials are prepared to guarantee as correct. This certified list can be easily downloaded to each physical precinct before the polls open. The voter can then check that his ballot is present on the certified list before voting. In addition, this certification prevents spurious complaints from malicious voters who might inject fraudulent ballots in to the system solely for the purpose of complaining and holding up the proper execution of the election.

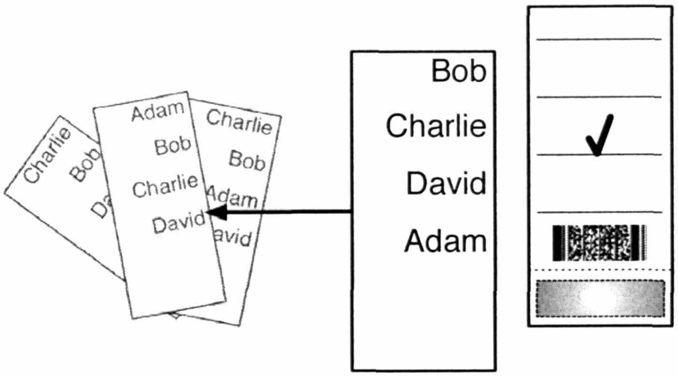


Figure 4-3: Separating the S&V ballot. Alice separates the left half of her ballot and places it into the appropriate receptacle which contains other discarded left halves (Alice could easily take one to claim she voted differently.)

4.1.3 Related Work

Chaum [40] introduced the first paper-based cryptographic scheme in 2004. Ryan et al. [41] proposed a variant, the Prêt-a-Voter scheme, recently extended to support reencryption mixes and just-in-time ballot printing [149]. Another variant by Randell and Ryan [144] suggests the use of scratch surfaces (though for different goals than ours). Chaum's latest variant, called Punchscan [66, 38], proposes a number of interesting variations to further simplify the paper-based voting approach.

All of these methods make use of mixnets to anonymize the ballots, though it should be noted that the mixnet proposed by Punchscan is simplified and uses only hash-based commitments for the shuffle permutation. In this latter case as well as in most decryption mixnets, proofs are performed by randomized partial checking [96]. In the case of reencryption mixnets, Neff's proof [124] can be used. In all cases, however, the officials are involved in the anonymization of each ballot.

4.1.4 Organization

In section 4.2, we cover some preliminaries. We cover the basic S&V method in section 4.3, some potential extensions in section 4.4, and the adaptation of S&V to the Chaum ballot in section 4.5. We consider the system's threat model in section 4.6, before concluding in section 4.7.

4.2 Preliminaries

In this section, we briefly review certain cryptographic concepts. Though some of these concepts are detailed in Chapter 2, we provide simple summaries here for easy reference.

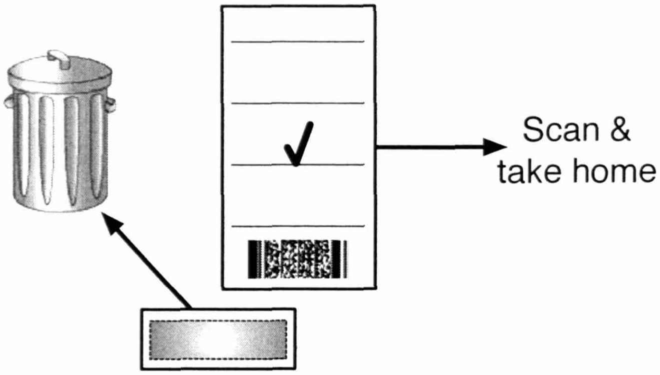


Figure 4-4: Casting the S&V ballot. The election official verifies that the scratch surface is intact, then discards it. The remainder of the ballot is cast using a typical modern scanner (likely more advanced than typical optical scanning voting machines.) Alice then takes it home as her receipt.

4.2.1 Paillier Cryptosystem

The Paillier public-key cryptosystem [133] provides semantically-secure encryption, efficient decryption, and an additive homomorphism by ciphertext multiplication. The details of the Paillier cryptosystem are covered in Chapter 2.

Generalized Paillier. Damgård and Jurik [48] provide a generalization of the Paillier cryptosystem that yields a larger plaintext domain with relatively less ciphertext expansion. Specifically, a plaintext in $\mathbf{Z}_{n^s}$ can be encrypted into a ciphertext in $\mathbf{Z}_{n^{(s+1)}}$, where the security of the scheme still depends on the bit-length of n. The security of this generalized scheme is proven under the same assumption as the original Paillier scheme.

Threshold Decryption. The Paillier cryptosystem supports fairly efficient threshold decryption [68], even in its generalized form [48]. Paillier also supports fairly efficient distributed key generation [49]. In other words, it is possible for l officials to jointly generate a Paillier keypair (pk, sk) such that each has a share $sk^{(i)}$ of the secret key. It is then possible for any k of these l officials to jointly decrypt a ciphertext in a truly distributed protocol.

Practical Considerations. The security of the Paillier cryptosystem relies on the hardness of the factoring problem. Thus, we must assume at least a 1024-bit modulus, and potentially a 2048-bit modulus. Given a κ-bit modulus, plaintext size is κ bits while ciphertext size is 2κ bits. For a larger plaintext domain, we can use Generalized Paillier, as described above.

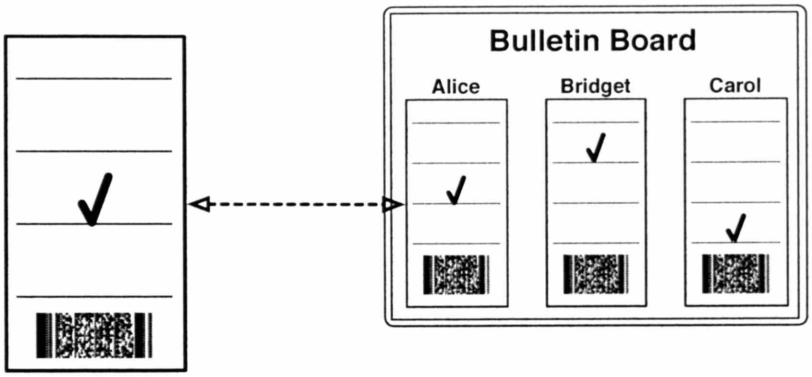
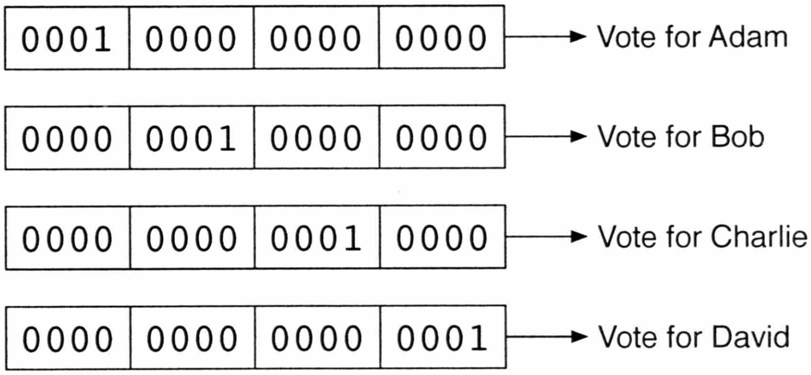


Figure 4-5: Verifying proper S&V casting. Alice can look up her ballot on the web, using her name and confirming that the barcode matches (assuming she or her helper organization has a barcode scanner.)



0001	0000	0000	0000	Vote for Adam
0000	0001	0000	0000	Vote for Bob
0000	0000	0001	0000	Vote for Charlie
0000	0000	0000	0001	Vote for David

Figure 4-6: A homomorphic counter with 4 slots. Assuming decimal encoding in this diagram, a vote for Adam is encoded as 10^{12}, a vote for Bob is encoded as 10^8, a vote for Charlie as 10^4, and a vote for David as $10^0 = 1$.

4.2.2 Homomorphic Counters

The homomorphic voting approach was made practical by Baudron et al. [13], using techniques introduced by Benaloh [43, 19]. The homomorphic multi-counter was specifically formalized by Katz et al. [105].

Baudron et al. describe a multi-counter encrypted under an additive cryptographic system, like Paillier. The bit-space of the plaintext is partitioned into separate counters, ensuring that enough bits are dedicated to each counter so that no overflow occurs from one counter to another (as this would violate the correctness of the multi-counter). See Figure 4-6 for an illustration.

Assuming a message domain of $\mathbf{Z}_n$ where $\kappa = |n|$ is the bit-size of n, we encode a value t_j for counter $j \in [1, z]$ as

$$t_j \cdot 2^{((j-1)M)}$$

and, thus, a set of z counters as:

$$\sum_{j=1}^{z} t_j 2^{(j-1)M}$$

Thus, each counter can run only up to $2^M - 1$, and we must ensure that $\kappa > zM$. To add 1 to counter j contained within the multi-counter T, we use the additive homomorphic property:

$$T' = T \cdot \mathcal{E}_{pk}(2^{(j-1)M})$$

Note that, given the semantic security of the Paillier cryptosystem, an observer cannot tell, from looking at this homomorphic operation, which internal counter was incremented. In other words, given encrypted messages, the homomorphic aggregation into an encrypted counter can be performed by anyone, including election officials *and* observers who have no privileged information.

4.2.3 Proofs of Correctness and NIZKs

If Alice encrypts a message m into a ciphertext c using the Paillier cryptosystem, she can prove, in honest-verifier zero-knowledge, that c is indeed the encryption of m, using a typical, three-round interactive protocol similar to Guillou and Quisquater's proof of RSA pre-image [88].

Using the techniques of Cramer et al. [44], this protocol can be extended to prove that ciphertext c encrypts *one* of possible values $(m_1, m_2, \ldots, m_z)$, without revealing which one. Combining this with the homomorphic proof technique of Juels and Jakobsson [94], one can prove, fairly efficient and in zero-knowledge, that a set of ciphertexts $(c_1, c_1, \ldots, c_z)$ encrypts a permutation of $m_1, m_2, \ldots, m_z$, assuming that no two subsets of $\{m_i\}$ have the same sum:

- for each c_i, prove that c_i encrypts one of $m_1, \ldots, m_z$,
- prove that the homomorphic ciphertext sum $\bigoplus_i c_i$ is the correct encryption of the plaintext sum $\sum_i m_i$.

For more than a handful of plaintexts, more efficient proof techniques are available, including Neff's shuffle proof of known plaintexts [124].

In any case, all of these proofs can be made non-interactive using the Fiat-Shamir heuristic [65], where the interactive verifier challenge is non-interactively generated as the cryptographic hash of the prover's first message in the three-round protocol. These types of proof, first introduced by Blum et al. [22], are abbreviated NIZKs.

4.2.4 Paper Ballots

Existing paper-based cryptographic methods use two types of ballot layout: the Prêt-a-Voter split ballot, and the Punchscan layered ballot. We review these approaches here, as they can be both adapted to use S&V.

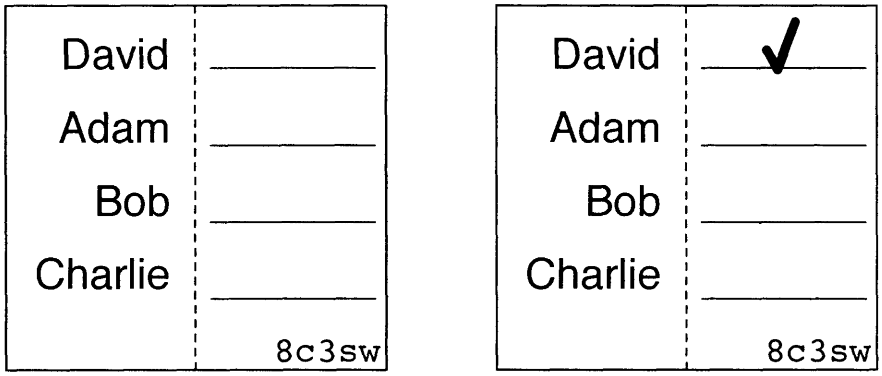


Figure 4-7: The Prêt-a-Voter Ballot: A ballot is a single sheet of paper with a mid-line perforation. The Voter fills in her choice, then tears the left half off and destroys it, casting the right half.

Prêt-a-Voter. In Ryan's Prêt-a-Voter, the ballot is a single sheet of paper with a perforated vertical mid-line. Candidate names appear on the left in a per-ballot-randomized order, with a corresponding space on the right half for a voter to mark his choice. After the voter has marked his choice, the two halves are separated: the left half (the one with the candidate names) is discarded, and the right half is cast. The right half contains a mixnet onion that allows administrators to recreate the left half of the ballot and determine the voter's choice. See Figure 4-7.

Punchscan. In Chaum's Punchscan [66], the ballot is composed of two super-imposed sheets. The top sheet contains the question, an assignment of candidates to codes (randomized by ballot), and physical, circular holes about half-an-inch wide which reveal codes on the bottom sheet. The codes on the bottom sheet match the codes on the top sheet, though their order on the bottom sheet is randomized. There may also be "dummy" holes and "dummy" values showing through.

Alice, the voter, selects a candidate, determines which code corresponds to this candidate, and uses a "bingo dauber" to mark the appropriate code through the physical hole. The use of this thick marker causes both sheets to be marked. Then, Alice separates the two sheets, destroys one, and casts the other. Individually, each sheet displays the voter's choice as either a code or a position, but the correspondence of code to position is only visible when both sheets are together. A hash-committed permutation on both sheets allows the administrators to reconstitute the discarded half and recover the vote. Because Alice chooses which half to destroy and which half to cast, she can eventually get assurance, with 50% soundness, that her ballot was correctly formed: the election officials eventually reveal what the kept halves should look like.

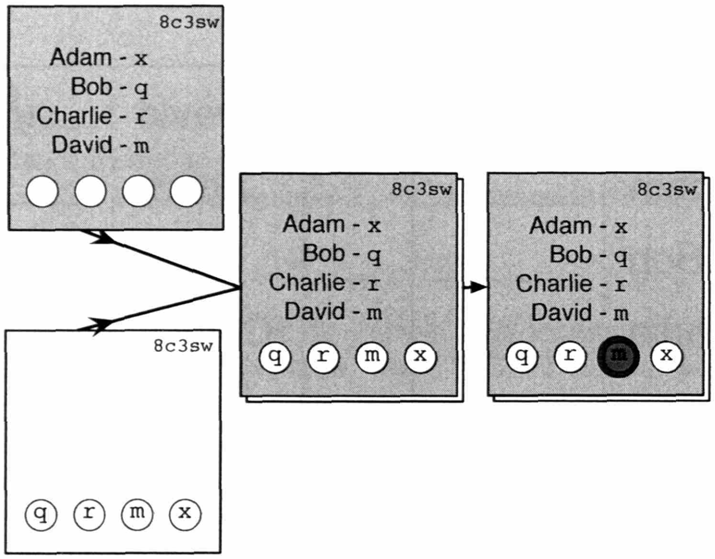


Figure 4-8: The Chaum Ballot: A ballot is composed of two super-imposed sheets. Alice, the voter, marks both sheets simultaneously using a dauber. The two sheets are separated, one is discarded, and the other is scanned and posted on the bulletin board. This same half is also the voter's receipt.

Auditing Prêt-a-Voter and Punchscan. In both Prêt-a-Voter and Punchscan, there are two ballot auditing components: verification of the *correct ballot form*, and verification of *correct tallying*. In both schemes, a system-wide cut-and-choose is performed before the election: for a randomly selected half of the ballots, election officials reveal the randomization values used in creating the ballots. These audited ballots are thus spoiled, as they no longer protect ballot secrecy. The remaining ballots, now proven to be almost all correct with very high probability, are used in the actual election. Once ballots are cast, they are shuffled, and the post-election audit consists of Randomized Partial Checking [96] on the shuffle and the prior permutation commitment. Punchscan adds an additional verification of ballot form after the election, thanks to the voter decision of which half to keep and which half to discard. This guarantees that any cheated ballot that made it through the initial audit will be detected with 50% probability.

Limitations. In the case of Prêt-a-Voter, significant synchronous involvement of election officials is required during all audits. It is particularly challenging to interactively reveal the randomization values for half the ballots while keeping the other half truly secret. Consequently, this audit is performed by election officials in advance. Individual voters must trust that this audit was performed correctly, in particular that election officials didn't collude to produce faulty ballots. This is, in effect, a weakening of the *ballot casting assurance* property (see Chapter 5) we desire from cryptographic voting systems. Ideally, voters should get direct assurance that their vote was recorded as intended, without having to trust election officials.

In the case of Punchscan, there is also some degree of dependence on synchronous involvement of the election officials. It should be noted, however, that the additional check performed on the ballot form alleviates this situation: Alice now gets direct assurance that her ballot was correctly formed. Unfortunately, this assurance comes *after* the close of elections. This may reduce Alice's trust in the system, since the error correction protocol will likely be onerous.

4.3 The Scratch & Vote Method

We now present the details of the Scratch & Vote method. To make the process as concrete as possible, we give a practical example of a single race with four candidates. Our description assumes a single race for now. Section 4.4 naturally extends these techniques to multiple races.

4.3.1 Election Preparation

At some point prior to election day, the list of candidates is certified by election officials and publicized for all to see. The z candidates are ordered in some fashion for purposes of assigning index numbers (alphabetical order is fine). This ordering need not be known by individual voters: election laws on candidate ordering should not apply to here.

Election officials then jointly generate a keypair for the election, where official O_i holds share $sk^{(i)}$ of the decryption key sk, and the combined public key is denoted pk. A parameter M is chosen, such that 2^M is greater than the total number of eligible voters. Election officials ensure that $\kappa = |n|$ is large enough to encode a multi-counter for z candidates, each with M bits, i.e. $\kappa > Mz$. A vote for candidate j is thus encoded as $\mathcal{E}_{pk}(2^{(j-1)M})$. When all is said and done, the election parameters are publicized:

$$params = \Big(pk, M, \{cand_1, cand_2, \dots, cand_z\}\Big)$$

Example. With $z = 4$ candidates, a candidate ordering is assigned: Adam is #1, Bob #2, Charlie #3, and David #4. With 2×10^8 eligible voters (big enough for the entire US), we set $M = 28 > \log_2(2 \times 10^8)$. A Paillier public key with $\kappa = |n| = 1024$ bits is largely sufficient for this single-race election. In fact, we could have up to 35 candidates for this single race, or 7 races with 5 candidates each, without having to alter cryptographic parameters.

4.3.2 Ballot Preparation

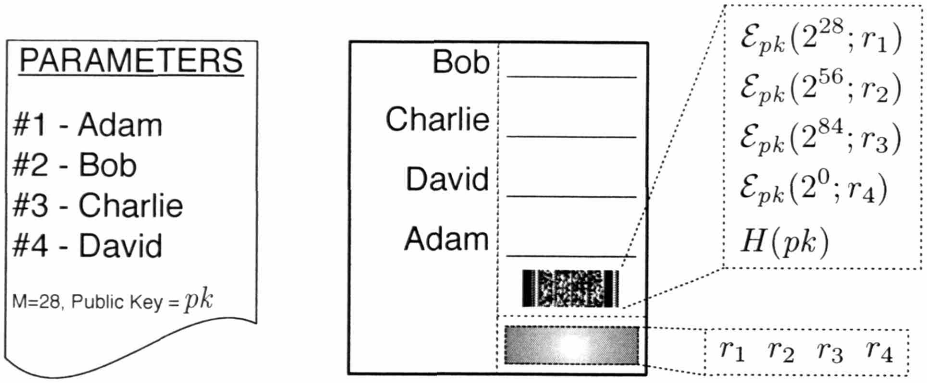

PARAMETERS
#1 - Adam
#2 - Bob
#3 - Charlie
#4 - David
M=28, Public Key = pk
Bob
Charlie
David
Adam
$\mathcal{E}_{pk}(2^{28}; r_1)$
$\mathcal{E}_{pk}(2^{56}; r_2)$
$\mathcal{E}_{pk}(2^{84}; r_3)$
$\mathcal{E}_{pk}(2^{0}; r_4)$
$H(pk)$
r_1 r_2 r_3 r_4

Figure 4-9: Public Election Parameters and the S&V Ballot. Election parameters are on the left. The ballot is midline-perforated. The order of the candidate names is randomized for each ballot. The 2D-barcode on the bottom right corner of the ballot contains ciphertexts encoding the candidates in the corresponding order according to the public election parameters. The scratch surface on the left hides the randomization values r_1, r_2, r_3, r_4 used to create the ciphertexts on the right. The ballot also includes an identifier of the election, which can simply be a hash of the election public key.

The Ballot. Our first S&V ballot is based on the Ryan Prêt-a-Voter ballot[41]. Each ballot is perforated along a vertical mid-line. On the left half of the ballot is the list of candidate names, in a randomized order. The right half of the ballot lines up scannable checkboxes (shown as lines in the diagram), with each candidate on the left half. The voter will mark one of those checkboxes.

Also on the right half, the S&V ballot contains the representation of ciphertexts that encode votes corresponding to each ordered option. The representation of these ciphertexts can be machine-readable only and should be optically-scannable, e.g. a 2D-barcode [174]. Just below this barcode, the ballot also includes a scratch surface, under which are found the randomization values used to produce the ciphertexts in the barcode. (See Figure 4-9.)

The Proofs. Election officials must also generate NIZK proofs of ballot correctness. These proofs will not be printed on the ballot, as they are too long. Instead, they are kept on the public bulletin board, indexed by the sequence of ciphertexts on the corresponding ballot (or a hash thereof), and used at tallying time to ensure that all ballots contribute at most one vote per race.

In addition to these proofs, election officials compile an *official ballot list*, which includes all properly created ballots designated again by the sequence of ciphertexts on each ballot. Officials digitally sign this ballot list, posting the list and signature on the bulletin board.

This official ballot list is particularly useful to help prevent various denial-of-service attacks against both voters and election officials.

Example. Assume the randomized candidate ordering of a given ballot is "Bob, Charlie, David, Alice", or, by index, "2,3,4,1". Recall that $M = 28$. The machine-encoding on the right ballot half should then be: $c_1 = \mathcal{E}_{pk}(2^{28}; r_1)$, $c_2 = \mathcal{E}_{pk}(2^{56}; r_2)$, $c_3 = \mathcal{E}_{pk}(2^{84}; r_3)$, $c_4 = \mathcal{E}_{pk}(2^0; r_4)$.

This encoding requires 4 ciphertexts, or 8192 bits. Under the scratch surface lie, in plaintext, r_1, r_2, r_3, r_4. Election officials also generate $\Pi_{H(c_1,c_2,c_3,c_4)}$, a NIZK of proper ballot form indexed by the ciphertexts, then post it to the bulletin board. The same hash $H(c_1, c_2, c_3, c_4)$ is also included in the compiled list of official ballots, which the election officials eventually sign.

4.3.3 Ballot Auditing

Ballot auditing in S&V uses a cut-and-choose approach to verify candidate ordering (while further ballot correctness is enforced by the bulletin-board NIZK). A random half of the ballots are audited, and almost all remaining ballots are then guaranteed to be correctly constructed: the probability that more than x bad ballots go undetected is 2^{-x}. Once audited, a ballot is spoiled and cannot be used to cast a vote.

Auditing a single ballot. This auditing process is similar to that of Prêt-a-Voter and Punchscan, with one major difference: auditing can be performed using only the *public* election parameters, without election-official interaction:

1. **Scratch**: the scratch surface is removed to reveal the randomization values.
2. **Encrypt**: the candidate ordering is encrypted with the revealed randomization values.
3. **Match**: the resulting ciphertexts are matched against the ciphertexts in the 2D-barcode.

Note that, to automate this process without having to scan or type in the displayed candidate ordering, one might perform the matching the other way around: read the ciphertexts, try all possible plaintext candidates with each revealed randomization value (effectively a decryption), and match the resulting candidate ordering with the ballot's displayed candidate ordering. Matching the ordering can be performed by the voter herself.

Spoiling a ballot. A ballot no longer protects privacy if its randomization values are revealed. Thus, if its scratch surface is removed, a ballot should be considered spoiled, and it cannot be cast. This is consistent with existing uses of scratch surfaces, e.g. those used on lottery tickets, and the usual "void if scratched off" message can be printed on top of the scratch surface. This condition for ballot casting must be enforced at ballot casting time, as will be described Section 4.3.4.

Who should audit? Though we find that individual voter auditing is preferable, some might prefer to audit ballots in a centralized fashion. Scratch & Vote supports such an audit method, of course. One can also imagine officials auditing a few ballots on their own before election day, in addition to per-voter auditing. S&V enables all of these auditing combinations.

Checking for Variability in Ordering. Malicious election officials might attempt to breach Alice's privacy by presenting all voters only ballots with the *same* candidate ordering. To protect against this de-randomization attack, Alice should select her two ballots herself, ensuring that there is enough variability between the ballots offered to her.

Chosen ballot check. Alice must also check the ballot she actually uses: she needs assurance that her ballot will count, specifically that it won't be disqualified for some unforeseen reason, e.g. an invalid NIZK proof at tallying time. For this purpose, Alice checks the presence of her ballot on the certified official ballot list, which she can obtain from the bulletin board ahead of time. If, at a later date, Alice's ballot is disqualified for any reason, she can present the signed official ballot list as a complaint.

4.3.4 Ballot Casting

On election day, after having audited and spoiled a first ballot, Alice enters the isolation booth with a second ballot. She fills it out by applying a checkmark (or filling in a bubble) next to the candidate name of her choice. She then detaches the left half of the ballot and discards it in the appropriate receptacle (inside the booth). She then leaves the voting booth, and casts her ballot as follows:

1. **Confirm**: An election official verifies that the scratch surface on Alice's ballot is intact. This is crucial to ensuring the secret ballot: if a voter sees the randomization values for the ballot she actually casts, then she can prove how she voted to a potential coercer. The official then detaches and discards the scratch surface in view of all observers.

2. **Scan**: Alice feeds the remainder of her ballot through an optical scanning machine, which records the barcode and checkmark position and posts them on a bulletin board *along with Alice's name or voter identification number.*

3. **Receipt**: Alice retains this same remainder as her encrypted receipt. She can later check that her ballot is indeed on the bulletin board.

4.3.5 Tallying

For each ballot on the bulletin board, election officials and observers check its NIZK. If it verifies, the ciphertext corresponding to the checkmark position is extracted from the 2D-barcode and aggregated into the homomorphic counter, just like any other homomorphic voting system. Anyone can verify that only valid ballots have been aggregated, as any observer can verify the NIZK and re-perform the appropriate homomorphic aggregation.

Similarly, all election trustees can independently verify that the homomorphic aggregation has been performed correctly. Then, the single resulting ciphertext counter is decrypted by a quorum of these trustees, along with proofs of correct decryption. The resulting plaintext reveals the vote count for each candidate. The tally and trustee proofs are posted to the bulletin board for all to see.

4.3.6 Performance Estimates

We consider computation and size requirements, specifically

- generating the NIZK proofs for each ballot, given that ZK proofs are typically expensive,
- auditing a ballot at voting time, given the voter's limited time and patience, and
- the physical size of the barcodes.

Consider an election with 5 races, each with 5 candidates.

Proofs. Each race contains 5 ciphertexts, one per candidate. Using the CDS [44] proof-of-partial-knowledge technique, each ciphertext must be proven to encrypt one of the 5 candidates. The CDS technique simulates 4 of these proofs and computes a fifth one for real. This requires the equivalent work of 5 proofs, both in terms of computation time and number of bits needed to represent the proof. In addition, the homomorphic sum of the ciphertexts must be proven to encrypt the sum of the candidate representations, which is one more proof. Thus, each race requires 26 proofs, and 5 races thus require 130 proofs.

Each of these proofs, whether real or simulated, requires two modular exponentiations. The entire proof thus requires a total of 260 modular exponentiations. Conservatively, modern processors can perform a 1024-bit modular exponentiation is approximately 12ms on a 2.8Ghz machine running GMP [85]. Thus, a single ballot proof can be performed in just over 3 seconds on a single CPU.

Each of these proofs is composed of 2 Paillier ciphertext space elements, and one Paillier plaintext space element (the challenge). Assuming a 1024-bit modulus, the ciphertext elements are 2048 bits and the plaintext is 1024 bits. Thus, each proof require 5120 bits, and the entire ballot thus requires 83 kilobytes of proof data on the web site.

Ballot Verification. At the polls, the only verification needed is that of correct encryption on the audited ballot. Given the 5 randomization values, all 5 values of r^n can be computed through modular exponentiation, after which only modular multiplications are needed, which are negligible by comparison. Thus, ballot verification can be performed in 60ms per race, or 300ms for our considered ballot. The scratch-off and the time allotted for each person to vote (1-2 minutes) will likely make the cryptographic cost negligible.

Barcode Size The PDF417 2D-barcode standard [167] can store 686 bytes of binary data per square inch, using a symbol density that is easily printed and scanned. In our example with 25 candidates, 25 Paillier ciphertexts are required, which means 6400 bytes, assuming $\kappa = 1024$ (a 1024-bit modulus for Paillier.) Thus, 10 square inches are sufficient to represent all of the ciphertexts we need for this sample election. Even if we factor in significant error correction, this represents less than 1/8th of the surface area of a a US Letter page.

4.4 Extensions

We now explore a few extensions to make Scratch & Vote even more practical.

4.4.1 Helper Organizations

We do not expect individual voters to show up to the polls with the equipment required to audit ballots and check the official ballot list. Instead, *helper organizations*, e.g. political parties and activist organizations, can provide this service at the polls. Voters can consult one or more of these, at their discretion. Most importantly, these helpers are not trusted with any private election data. Chapter 5 provides more details on these helper organizations and how they fit into the voting process to ensure the property of ballot casting assurance.

4.4.2 Multiple Races & Many Candidates

When the election contains more than one race, it may outgrow the space of the multi-counter, which can only hold $|n|/z$ counters. One solution is to use higher-degree encryption using the Damgård-Jurik generalization [48], so that the counter space can be $s|n|$ rather than $|n|$, with a corresponding ciphertext length of $(s+1)|n|$. Unfortunately, this ciphertext size may outgrow the 2D-barcode encoding, which is expected to hold no more than a few kilobytes.

Another option is to designate, in the public election parameters, separate multi-counters, potentially one per race. In that case, the parameters must also indicate the race/multi-counter assignments. With a separate 2D-barcode per race, most practical cases are accounted for (barring elections such as the California Recall election of 2004, which had more than 100 candidates.)

In extreme cases, such as the California Recall of 2004, there is no choice but to use the Damgård-Jurik generalization, as the individual ciphertexts for a given race should not be distinguishable and thus cannot be assigned to different multi-counters. If a single 2D-barcode cannot hold all the required ciphertexts for that one race, we can, as a last resort, designate a separate 2D-barcode for each candidate. The resulting auditing complexity is an inevitable consequence of these extreme conditions.

4.4.3 Reducing the Scratch Surface

As the printed material behind the scratch surface may become damaged from the scratching, we cannot expect to reliably store significant amounts of data behind this scratch surface. In fact, it is easy to reduce this data by having election parameters designate a pseudo-random function, call it PRF, which generates all the randomization values from a single short seed, which need only be 128 bits. Such a data length can be easily encoded as alphanumeric characters or as a single-dimension barcode, both of which offer enough redundancy to withstand a few scratches.

4.4.4 Chain Voting and Scratch Surfaces

All paper-based voting systems have long been known to be susceptible to chain voting attacks. In these attacks, a coercer gives Alice a pre-marked ballot before she enters the polling location, expecting her to cast this pre-marked ballot and return a blank ballot to him on her way out.

The well-known countermeasure to chain voting attacks [100] suggests having unique ballot identifiers on a tear-off slip attached to the ballot. An official writes down the ballot identifier for Alice before she enters the isolation booth. At ballot casting time, the official checks that the ballot identifier is still present and matches the recorded value. Then, for anonymity, the identifier is torn off and discarded.

This process is, in fact, almost identical to the scratch surface tear-off we suggest. Thus, our election-official verification process can be piggy-backed onto existing practices. In addition to checking the ballot identifier, the election official must simply check the scratch surface integrity. The overhead of our proposal at casting time is thus minimal.

4.4.5 Write-Ins

Like the Prêt-a-Voter and Punchscan ballots, Scratch & Vote does not support write-in votes out of the box. A separate system should be used, where a special option named "write-in" is selected by the voter, and the voter can separately cast the content of the write-in. The details of this process can be worked out for all paper-based schemes, possibly using the vector-ballot method of Kiayias and Yung [107].

4.5 Adapting Punchscan

Recall that Chaum's Punchscan facilitates more races per ballot than Prêt-a-Voter, because the full ballot face can be used without a midline separation. However, Punchscan is also more complicated, because the voter may cast either sheet. This makes the mandatory destruction of the remaining half more delicate, since Alice could easily sell her vote if she successfully preserves the second half.

Thus, we propose a new ballot that combines the qualities of Punchscan and Prêt-a-Voter and adds the S&V method. As this ballot originated from Punchscan, we call it the

Punchscan Scratch & Vote ballot. However, we emphasize that it inherits some properties from Prêt-a-Voter, too.

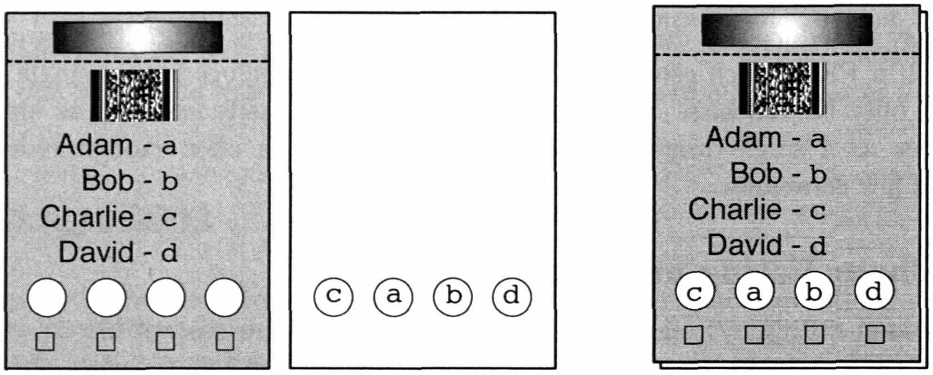


Figure 4-10: The Punchscan Scratch & Vote variant. The left and middle sheets are superimposed to create the ballot on the right. The bottom sheet contains no identifying information. The top layer has circular holes big enough to let the candidate ordering from the bottom sheet show through. The checkmark locations, represented by small squares, are only on the top layer. Note how the codes corresponding to the candidates are intuitive, rather than random.

Punchscan **Scratch & Vote Ballot.** The Punchscan Scratch & Vote ballot, like Punchscan, is composed of two superimposed sheets. Unlike the original Punchscan, the two sheets serve different purposes. Alice, the voter, will be expected to separate both halves and cast the top half in all cases. The bottom half, like Prêt-a-Voter's left half, is generic, and its destruction need not be strongly verified. This change is "safe" because the cut-and-choose is now performed by verifying two ballots, rather than splitting one. Note that, in addition, while the original Punchscan's cut-and-choose can only be verifier after having cast a ballot, this variant allows the voter to verify *before* casting a ballot.

The top sheet lists the races and candidates in standard order, with a standard code assigned to each candidate (e.g. 'D' for democrats, 'R' for republicans.) Again, this differs from the Punchscan ballot, where random codes are assigned to candidates. This top sheet offers checkboxes for each race, and one hole *above* each checkbox which reveals the code letter displayed on the bottom half at that position. Also included on the top sheet are the S&V components: the ciphertexts in a 2D-barcode, and the randomization values hidden under a scratch surface.

The bottom sheet contains only the standard candidate codes in random order. The ciphertexts on the front sheet should match this random order. Note, again, that this bottom sheet is entirely generic: it contains no identifier of any kind, no ciphertexts, and no randomization values. It functions exactly like the Prêt-a-Voter left half.

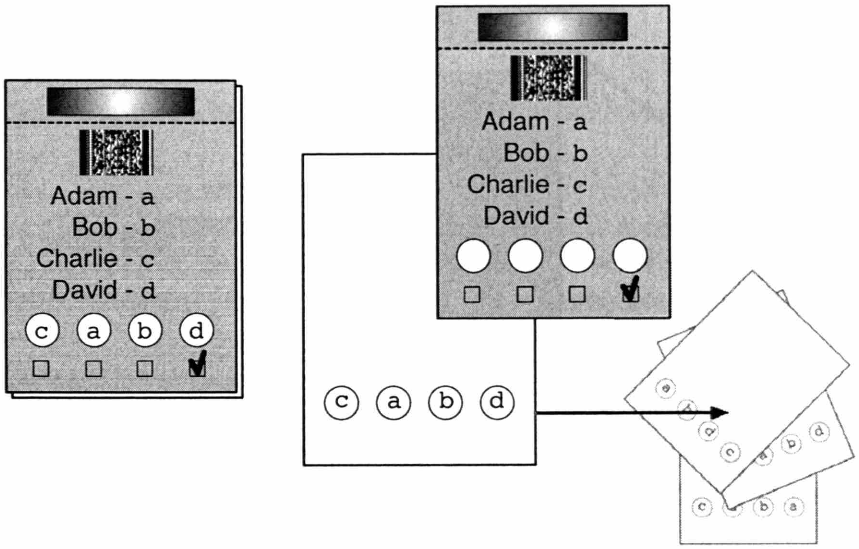


Figure 4-11: The Punchscan Scratch & Vote ballot separation. Alice separates the top and bottom sheets, depositing the bottom sheet in the appropriate receptacle. The top sheet is effectively an encrypted ballot.

Pre-voting Verification. Just like in the original Prêt-a-Voter-based S&V ballot, Alice chooses two ballots. She audits one by scratching off the scratch surface and having a helper organization verify the randomization values for the ballot's candidate ordering. Alice then votes with the second ballot, as the audited ballot with the scratch surface removed is now void.

Casting the Ballot. Alice marks her ballot in isolation. Note that, unlike the original Punchscan method, the markings in the top-sheet bubbles do not bleed through to the bottom half. When she is ready, Alice detaches the bottom half of the ballot and discards it in the proper receptacle (where, again, she can easily grab another bottom half to claim that she voted for someone else.)

Alice then presents the top sheet of her ballot to the election official, who treats it exactly like in the Prêt-a-Voter Scratch & Vote ballot: verify the scratch surface, detach and discard it, and scan the remainder. As previously, this remaining half does not reveal how Alice voted, which means the election official can take all the necessary care to examine the ballot without risking a privacy violation. Again, this remainder serves as Alice's receipt.

4.6 Threat Model

We consider various threats and how Scratch & Vote handles them. We cover the threats thematically rather than chronologically, as some threats pertain to multiple steps of the election lifecycle.

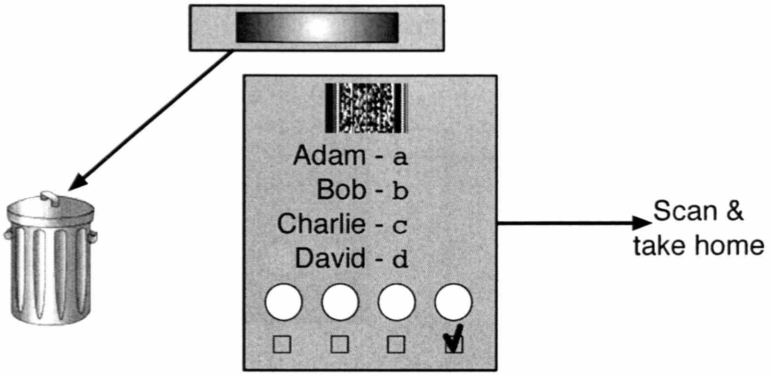


Figure 4-12: Casting a Punchscan Scratch & Vote ballot. An election official verifies that the scratch surface is intact, then tears it off and discards it in view of all observers. The remainder is scanned and cast. Alice then takes it home as her receipt.

4.6.1 Attacking the Ballots

The first obvious target of S&V is the creation of malicious ballot. We consider the various parties that might be involved in performing this attack.

Election officials. An election official might create bad ballots in two ways: a completely invalid ballot or, more perniciously, a valid ballot that does not match the human-readable candidate ordering. In either case, the first line of defense is the voter cut-and-choose: only a small number of such ballots can go undetected, since half of them will get audited randomly. In the case of completely invalid ballots, the verification is much more stringent: election officials would have to answer for certified ballots that do not have a proper NIZK proof, and only valid ballots can have proper NIZK proofs.

External parties. External parties may wish to introduce fraudulent ballots, most likely as a denial-of-service attack against voters at certain precincts, or, by vastly increasing the number of complaints, as a denial-of-service attack against election officials. These problems are thwarted by the certified ballot list. The moment an election official discovers an uncertified ballot, he can begin an investigation. If officials fail to catch the problem, the voters' helper organizations will. Consequently, fraudulent ballots should be caught before the voter enters the isolation booth.

Collusion between officials and voters. A single maliciously crafted ballot could alter the count significantly in the homomorphic aggregation. This is particularly problematic if the officials collude with a voter who won't perform the proper cut-and-choose audit because he is very much interested in using the fraudulent ballot. Once again, the NIZK proof on the bulletin board prevents this from *ever* happening, providing a universally verifiable guarantee that each cast ballot only contributes a single vote to a single candidate for each race.

4.6.2 Attacking Ballot Secrecy

Another obvious target of attack is the secrecy guaranteed by the ballots, especially as the entire protection of a given ballot is contained within the ballot itself.

Leaky Ballots. Election administrators could leak information about the ballot candidate ordering using the ciphertext randomness. This threat is somewhat lessened with the use of seed-based randomness, as long as a portion of the seed is public and selected *after* the individuals ballot seeds are picked. However, this topic requires further exploration.

Tampering with the scratch surface. Eve, a malicious voter, may attempt to remove the scratch surface from her ballot, read the randomization values, and replace the scratch surface undetected. This would allow Eve to sell her vote, given that her encrypted vote will be posted on the bulletin board, along with Eve's full name, for all to see and audit. We must assume, as a defense against this threat, that the scratch surface is sufficiently tamper-proof to prevent such an easy replacement that would fool an election administrator; this assumption appears quite reasonable. Real-world experiments will be necessary to determine what level of tamper-proofing is feasible.

Ballot face recording. One significant weakness of all pre-printed paper-based cryptographic voting, including Scratch & Vote, is that election officials who produce the ballots may record the ballot ciphertexts and candidate orderings, thus violating ballot secrecy.

Even in Prêt-a-Voter and Punchscan, which use multiple authorities in a mixnet setting, the last mix server knows the final candidate ordering. It may be worthwhile to explore clever ways of distributing the ballot generation mechanism. The recent proposal of Ryan and Schneider [149] addresses this threat with on-demand ballot printing, though this requires significantly more deployed technology at the precinct level. The best solution may be process-based, where machines produce ballots and immediately forget the randomness used.

The ballot face recording problem exists for more than just election officials: those who transport ballots may have a chance to record the correspondence of candidate ordering to barcode. We note, again, that Prêt-a-Voter and Punchscan suffer from the same problem. One promising defense in all cases is simply to hide some portion of the ballot such that it can no longer be uniquely identified during transport. For example, individual ballots can be sealed individually in opaque wrappers. Alternatively, the 2D-barcode can be hidden under opaque plastic that can be peeled off prior to voting. If the printed barcode is particularly resilient, one can even use an additional scratch surface.

Casting. At ballot casting time, election administrators must ensure that the cast ballot has not revealed its randomization values, for this would enable vote selling. Of course, this verification must be performed without violating ballot secrecy in the process. Our proposal specifically addresses this threat: an election official only sees the encrypted half of the ballot. He can take all the necessary care to verify that the scratch surface is intact, while ballot secrecy remains protected.

A threat remains: official-voter collusion. If an election official and voter collude to preserve, rather than discard, the scratch surface, the voter may be able to reveal his selection to the official (and later to other parties.) Sufficient observation of the voting process by competing political parties should address this issue. S&V further mitigates this risk with ballots created such that the "missing half" is generic. Thus, voters can easily pick up alternative "missing halves" to claim they voted differently, and a coercer may not be certain whether the claimed half is, indeed the proper decryptor half for Alice's ballot.

Coerced Randomization. Recently, a new threat to paper-based voting systems has been pointed out: coerced randomization [30]. In this attack, a coercer wishes to "reduce to random" Alice's vote. Consider, for example, the situation where Alice votes in a precinct that historically favors one political party by a wide margin. Such precincts are quite common in the United States. A coercer can ask Alice to vote for the first candidate in the list, no matter what that candidate is. The coercer can check this by viewing the bulletin board under Alice's name or voter identification number. Of course, the coercer won't know for sure who Alice voted for—in fact she may, by sheer luck, have obtained a ballot where this position coincides with her preferred choice—but he will have effectively reduced her vote to random. With enough voters, the coercer can statistically reduce the number of votes for the party typically favored by this precinct.

By way of countermeasure, one can offer the voter enough ballot ordering selections that she can pick a ballot where the prescribed behavior fits her personal choice. Unfortunately, the attack can become much more complex: for example, the prescribed behavior may involve voting for a position on the ballot that depends on the ballot identifier. This issue merits additional research. However, we note that Scratch & Vote does not make this problem any worse: Punchscan and Prêt-a-Voter are equally vulnerable.

4.6.3 Attacking the Bulletin Board and the Tally

Much of the security of the tallying process depends on the bulletin board. An attacker may want to insert fraudulent data, for example to change the election parameters or replace an honest citizen's cast vote. Digital signatures on all posted data can prevent such attacks, assuming that a minimal PKI is deployed to certify the public keys of election officials. Public observers of the bulletin board content, including helper organizations, can then detect bad data.

Alternatively, an attacker may want to suppress information from the bulletin board. In particular, the bulletin board server itself may suppress information. To protect against this attack, the bulletin board should be implemented by multiple servers. These servers may run a distributed Byzantine agreement protocol to ensure consistency of content [110], or observers may simply rely on cryptographic signatures of the content and the redundancy of the servers to catch any server the suppresses content.

Given a secure bulletin board implemented as above, attacks on the tallying process can be prevented, because every step is verified with proofs. Ballots are proven well-formed by the NIZK proof on the bulletin board, any observer can re-perform the homomorphic aggregation, and the final tally decryption is also proven correct.

4.7 Conclusion

We have proposed Scratch & Vote, a simple cryptographic voting system that can be implemented with today's technology, at very low cost and minimized complexity. Most importantly, ballots are self-contained: any helper organization, or the voter herself, can audit the ballot *before* casting her vote and without interacting with election officials. Given its intuitive use of scratch surfaces, Scratch & Vote may prove particularly useful in providing an accessible explanation of the power of cryptographic verification for voting systems.

Chapter 5

Casting a Verifiable Secret Ballot

This chapter combines two separate works: Ballot Casting Assurance [6], presented in sections 5.2 and 5.3, and Assisted-Human Interactive Proofs [17], presented in sections 5.4–5.7. Section 5.1 provides an introduction to the concepts and some background. This work was done in collaboration with C. Andrew Neff.

5.1 Introduction

How can a voter be certain that her intent was correctly captured by the "voting system?" She may be certain that she checked the right checkmark, touched the right portion of the screen, punched the right hole, or otherwise filled out her ballot correctly. However, her ballot may get lost, misread, or, in the worst of cases, maliciously removed from the tally, all without her knowledge.

In Chapter 1, we saw, at a high level, how cryptography presents solutions to this challenge. Votes are encrypted and placed on a bulletin board, and Alice, the voter, receives proof that her vote was correctly encrypted. Then, officials cooperate to anonymize and tally the encrypted ballots. Again, proofs of correctness are provided so that anyone may verify this tally.

However, typical cryptographic proofs, e.g. Chaum-Pedersen [37], assume that the verifier, in this case the voter, is able to perform a reasonable amount of computation. When the verifier is human, cryptographic protocols have often assumed that she can use a trusted computing device to perform the verification on her behalf. In the case of voting, however, this solution is unacceptable: the entire correctness of the election would then rely on the trustworthiness of this device and its manufacturer.

This chapter first defines the concept of *ballot casting assurance.* At a high level, a voting system with ballot casting assurance is one that gives Alice direct assurance that her selection became, as intended, an input into the tallying process. We show that the notion of "cast as intended," while it may have been intended to represent this same concept, does not, in its present common usage, imply the end-to-end verifiability we truly seek. We consider how, with ballot casting assurance, we can begin to work on not just the *detection* of errors, but their *correction*.

Then, we delve into technical detail. We provide a definition of interactive proofs when the verifier is human. We call these proofs *Assisted-Human Interactive Proofs* (AHIPs) and emphasize that, in the context of voting, an assistant helps the voter but *does not* learn her vote. We consider MarkPledge, Neff's [29] voter-intent verification protocol, and show that, with a small adjustment, it successfully fulfills the requirements of our definition. Then, we present a significantly more efficient scheme based on the same principles, and once again show that it fits the definition.

Human Limitations. There is an online card trick which fools most people [146]. In this trick, 5 playing cards are displayed on screen, and the player is asked to privately choose one, never indicating this choice to the computer. The computer then asks a number of unrelated questions. Eventually it displays 4 cards and claims "I have read your mind and removed the card you picked from the set." The trick succeeds because the 4 cards displayed on the last screen were never present on the first screen, but most people don't notice, as they only remember the one card they chose, not the cards they set aside. If we are to design a protocol that humans can verify, we must thus ensure that the computation requirements are truly minimal!

This Introduction. In this extended introduction, we present an overview of ballot casting assurance in the context of cryptographic voting system verification. We also briefly review the notion of an interactive proof. Then, we present an intuition for our definition of Assisted-Human Interactive Proofs, and an overview of our two implementations, first a small change on Neff's existing scheme, then a new scheme. The rest of this chapter provides complete technical details, including the exact setup for ballot casting assurance, the precise definition of AHIPs, the detailed protocols descriptions of the two schemes, and the complete proofs of their security according to the AHIP definition.

5.1.1 Verifiability

The key complication of verifiable voting is due to the secret ballot, as described in Chapter 1. With secrecy, auditing becomes significantly more difficult: how can we verify that the votes were properly captured and tallied while ensuring that their secrecy is maintained? However, there are techniques for verification. In particular, the concept of *universal verifiability* has been discussed in the literature for more than 10 years.

Universal Verifiability. Since the early 1990s, many cryptographic voting protocols have been proposed to provide *universal verifiability* [151, 46, 1, 124]. In these schemes, any observer can verify that only registered voters cast ballots and that cast ballots are tallied correctly. Universal verifiability uses cryptography to restore the bulletin board of yore. Ballots are encrypted and posted along with the voter's plaintext identity. Universal verifiability thus provides a public tally, patching a large portion of the auditability hole caused by the secret ballot.

Ballot Casting Assurance. Universal verifiability does not provide complete auditability. In addition to knowing that all votes were correctly tallied, Alice would like *direct verification* that *her vote* was properly cast and recorded into this tally. We define this principle as *ballot casting assurance* and argue that it, too, fills an auditability gap left by the secret ballot. Interestingly, to the average voter, ballot casting assurance may be more intuitive than universal verifiability, for it is a simple, individually relevant question: independently of all other issues, is Alice certain that her vote "made it?"

Ballot casting assurance is effectively the combination of two existing voting protocol concepts: that ballots should be *cast as intended*, and *recorded as cast.* The added benefit of ballot casting assurance is two-fold:

- **End-to-end verification:** Alice only cares that the recorded ballot matches her intention. Verifying the in-between "ballot casting" is an artifact of how certain voting schemes are verified, and need not be built into a definition of voting system security.
- **Direct verification:** Alice wants direct, not mediated, verification of the correct recording of her vote into the tally. In particular, if Alice must trust election officials to eventually record her vote, the scheme does *not* provide ballot casting assurance.

Incoercibility. The difficulty of accomplishing ballot casting assurance comes from ballot secrecy, in particular the incoercibility requirement. Even if Alice wants to sell her vote, the protocol should ensure that she cannot succeed with any reasonable probability. Achieving a realistic combination of incoercibility and verification is the ultimate goal of schemes that support ballot casting assurance.

The Possibility of Failure Recovery. Once ballot casting assurance enables this direct and immediate verification, a new possibility arises: if Alice determines that her vote has not been correctly recorded, she can immediately complain and rectify the situation. Thus, ballot casting assurance is fully achieved when Alice can vote, verify, and revote until verification succeeds.

5.1.2 Interactive Proofs & Receipt Freeness

In an *interactive proof*, a prover $\mathcal{P}$ interacts with a verifier $\mathcal{V}$ to demonstrate that a string x has a certain property, e.g. given $x = (c, m, pk)$, c is an encryption of m under public key pk. More formally, the assertion is that the string x is in some NP language $\mathcal{L}$. Certain proofs are said to be *zero-knowledge*, in which case $\mathcal{V}$ learns nothing more than the truth of the assertion that $x \in \mathcal{L}$. A slightly weaker type of proof is the *witness-hiding proof* [62], where $\mathcal{V}$ may learn some information, as long as she doesn't learn a new witness w to x's membership in $\mathcal{L}$. Thus, if c is the encryption of m with randomization value (and thus witness) r, $\mathcal{V}$ should *not* learn r in either a witness-indistinguishable or a zero-knowledge proof.

In the voting setting, m is Alice's plaintext vote, and c is the encryption of m, destined for the bulletin board. The proof protocol should convince Alice that c encrypts m, yet Alice should be unable to convince a third party of this fact, even if she tries. In other words, Alice cannot learn r. This property is often called *receipt-freeness*, and a number of definitions and implementations exist in the literature (as reviewed in Chapter 2).

In most known interactive proof protocols, $\mathcal{P}$ and $\mathcal{V}$ are assumed to be computers, modeled as probabilistic polynomial-time (*PPT*) Interactive Turing Machines (*ITM*). When one of the parties, Alice, is "only human," a number of protocols suggest providing her with a trusted computing device that performs the protocol on her behalf. This solution is inadequate for voting, as it simply shifts trust from the election officials to the device manufacturer. We need to provide Alice with an *unmediated proof of correct ballot casting*, not one that depends on the trustworthiness of a device. Because of the ballot secrecy requirement, this unmediated proof must be also non-transferable.

To date, this practical setting, with Alice the human verifier, has not been thoroughly modeled. Thus, the handful of proposed protocols in this vein [40, 41] have not been proven secure.

5.1.3 Assisted-Human Interactive Proofs

We introduce *Assisted-Human Interactive Proofs (AHIPs)*, where a human verifier, Alice, *directly* interacts with a machine prover which convinces her of some assertion. As the end of the interaction, Alice receives a secret receipt from the prover, which she provides to a machine assistant. Using both the information she gained in her private interaction with the machine prover and this third-party assistant, Alice can decide whether the protocol succeeded or not. AHIP protocols are a crucial component of voting systems that support ballot casting assurance.

The interesting security property we propose concerns this newly-introduced assistant. Intuitively, the verifier learns (or already knows) some information that the assistant should not learn. In particular, the receipt obtained by the verifier should not reveal this crucial information. Even if Alice, the verifier, is willing to reveal it, she should not be able to convince the assistant with any reasonable probability. This security property should hold across the entire range of possible verifiers, from highly simplistic to fully *PPT*-capable. In the context of voting, this secret receipt hides Alice's vote, even if she is willing to be coerced, and even if she surreptitiously brings computing equipment with her in the voting booth. Thus, the protocol provides a *secret receipt*, but, in the original terminology, it remains *receipt-free.*

Model. We model the prover as a *PPT ITM* $\mathcal{P}$. For notation clarity, we model the limited (human) verifier as two *ITM*s: $\mathcal{V}_{\mathsf{int}}$ and $\mathcal{V}_{\mathsf{check}}$. The $\mathcal{V}_{\mathsf{int}}$ component interacts with $\mathcal{P}$, while $\mathcal{V}_{\mathsf{check}}$ is a non-interactive "checker" component. The initial setup is identical to typical interactive proofs: $\mathcal{P}$ interacts with $\mathcal{V}_{\mathsf{int}}$ to prove that string x is in language $\mathcal{L}$.

We introduce $\mathcal{A}$, the assistant, another *PPT ITM*, that the verifier consults *after* its private interaction with the prover. The Prover $\mathcal{P}$ outputs a secret receipt of some kind, denoted $\mathsf{receipt}\langle \mathcal{P}, \mathcal{V}_{\mathsf{int}} \rangle$. Together, $\mathcal{V}_{\mathsf{check}}$ and $\mathcal{A}$ determine the outcome of the protocol: $\mathcal{V}_{\mathsf{check}}$ checks $\mathsf{receipt}\langle \mathcal{P}, \mathcal{V}_{\mathsf{int}} \rangle$ against the output of $\mathcal{V}_{\mathsf{int}}$, while $\mathcal{A}$ checks its internal consistency. The protocol succeeds if *both* $\mathcal{V}_{\mathsf{check}}$ and $\mathcal{A}$ declare success. Thus, the protocol is complete if both $\mathcal{V}_{\mathsf{check}}$ and $\mathcal{A}$ accept when everyone is honest and $x \in \mathcal{L}$. The protocol is sound if, with some probability, either $\mathcal{V}_{\mathsf{check}}$ or $\mathcal{A}$ declares failure when $x \notin \mathcal{L}$, against any prover $\mathcal{P}^*$. Figure 5-1 represents this setup.

Private-Input Hiding. Interesting AHIPs produce receipts that hide *some portion of the prover's assertion* while still enabling $\mathcal{A}$ to determine the validity of the *full assertion.* We model this with an input string x composed of a public component x_{pub} and a private component x_{priv}. The receipt, denoted $\mathsf{receipt}_{\mathcal{V}}\langle \mathcal{P}, \mathcal{V} \rangle$, may yield x_{pub}, but should hide x_{priv}, at least computationally. More specifically, for every verifier willing to reveal x_{priv}, there is another verifier that, for any other x'_{priv}, falsely claims the same x_{priv}, such that both verifiers are indistinguishable. It is worth noting that a distinguisher has access to the prover's receipt and any verifier output.

We frame this private-input-hiding property as an indistinguishability property from the point of view of the coercer. In the context of voting, this indistinguishability provides incoercibility. More specifically, consider Sally, a willing vote seller, and David, a double-crossing vote seller. Sally acts according to her coercer's requests. David pretends to act according to his coercer's requests, but eventually double-crosses him and votes against the coercer's preferred candidate. Private-input hiding guarantees that, no matter what strategy Sally adopts to convince her coercer that she has complied with the vote-selling "deal," David can adapt his strategy so that he can make a similarly convincing argument, even though, in fact, he did not comply.

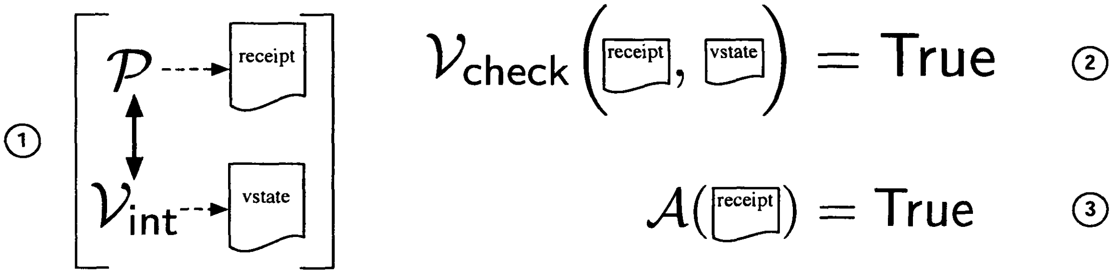


Figure 5-1: (1) Prover $\mathcal{P}$ and the Verifier's interactive component $\mathcal{V}_{\mathsf{int}}$ interact in private, with $\mathcal{P}$ generating a secret receipt and $\mathcal{V}_{\mathsf{int}}$ producing some internal state. (2) $\mathcal{V}_{\mathsf{check}}$, the non-interactive component of the verifier, checks the receipt against the internal state produced by $\mathcal{V}_{\mathsf{int}}$. (3) $\mathcal{A}$ checks the receipt for internal consistency.

AHIP Languages. For the purpose of this work, we consider a class of languages we call, for lack of a better name, AHIP Languages. These languages provide a number of properties which make them particularly interesting for the development of AHIP proofs:

- strings x in the language can be parsed as pairs (x_{pub}, x_{priv}),
- for each x_{pub}, there is at most one x_{priv} such that (x_{pub}, x_{priv}) is in the language, and
- x_{pub} can be sampled in polynomial time given x_{priv}, such that $(x_{pub}, x_{priv}) \in \mathcal{L}$.

In particular, languages of proper ciphertext/plaintext pairs make excellent AHIP languages. In voting, these pairs are the encrypted-ballot/plaintext-ballot pair, where membership of the pair indicates a properly encrypted ballot.

As a result of this definition, the public input hides the private input only computationally: given x_{pub}, a computationally-unbounded algorithm can simply try all possible x_{priv} values of the proper length (given the security parameter) to discover the single matching value. Thus, our AHIP proofs are inherently constrained: they, too, can only hide the private input computationally.

We believe this is, in fact, a natural consequence of the purpose of these protocols. If the protocol were to statistically hide the private input, the verifier would need an additional mechanism to ensure that her receipt indeed corresponds to the specific x_{priv} claimed in her private interaction with the Prover. It is unclear how useful such protocols would be, especially for voting: Alice would not be certain that her receipt indeed encodes her vote. This is consistent with the observations and proofs of Canetti and Gennaro [33] regarding incoercibility.

Witness Hiding. We show, in Theorem 5-1, that if a protocol is private-input hiding for an AHIP language, then it is also witness hiding. At a high level, if a corrupt verifier can learn a witness, then it can simply provide this witness to the Assistant, which is irrefutable evidence of the value of the private input. We note that the inverse implication is not true: a typical witness-hiding proof generally reveals the complete input string or requires it to be public to begin with, thus revealing the private input as well.

5.1.4 Implementing AHIP Protocols for Ballot Casting

In sections 5.5 and 5.6, we provide two AHIP protocols for ballot casting; here, we present an overview and introduction to this material.

In both protocols, the verifier is only expected to compare very short strings. The first protocol, BitProof, is based on an existing protocol due to Neff [29], slightly modified to ensure security under our definition. The second, BetterBitProof, is a new, more efficient protocol that accomplishes the same goal with short receipt representation and minimal round complexity. Both are proven secure against our new definition. First, however, we describe a high-level, black-box-component implementation of an AHIP proof to illustrate the need for specific, optimized protocols.

The Voting Protocol. For voting purposes, we wish to build an AHIP protocol that lets the voting machine demonstrate to Alice, the human voter, that a given ciphertext correctly encodes her candidate choice, represented as index j out of z possible options. Note that this ciphertext may be generated in any number of ways, as long as the prover knows a witness that proves the ciphertext's decryption, for example the randomization value of the ciphertext. The protocol must hide this witness from the voter, and, in addition, it must hide the index j from the assistant. More formally and specifically, given a public key setting (pk, sk), an index $j \in [1, z]$, and ciphertexts $c_1, c_2, \ldots, c_z$, $\mathcal{P}$ proves that:

- $c_j = \mathcal{E}_{pk}(1)$, and
- $c_i = \mathcal{E}_{pk}(0), \forall i \neq j$.

In the AHIP language, the public input x_{pub} is $(pk, c_1, c_2 \ldots, c_z)$, the private input is j, and the witness is $(r_1, r_2, \ldots, r_z)$, the set of randomization values for ciphertexts $(c_1, c_2, \ldots, c_z)$ respectively. We call this a *proof of integer-in-range plaintext.*

In this work, we assume that the list of ciphertexts is always properly formed: exactly one ciphertext will decrypt to 1, while all others will decrypt to 0. The key to the proof is demonstrating that the j'th one decrypts to 1. This assumption is reasonable, as there are numerous zero-knowledge techniques for proving the ciphertext's valid form that can be checked entirely by the assistant without revealing j. This proof can be performed in parallel with the normal AHIP proof, with details printed on the receipt for the Assistant to check: Alice doesn't need to participate. We review this idea in section 5.7.

Construction from Black Box Components. The proof of integer-in-range plaintext can be implemented using black-box cryptographic components, specifically using any public-key cryptosystem with a special honest-verifier zero-knowledge, 3-round proof of plaintext equivalence. Recall that special honest-verifier zero-knowledge proofs, introduced by Cramer et al. [44], provide overwhelming soundness with transcripts that can be simulated entirely from the challenge chal. We assume here that the values 0 and 1 are in the plaintext space of the cryptosystem. If they are not, we can simply use two values m_0 and m_1 in the plaintext space that "play the role" of 0 and 1, respectively. Then, using these components:

1. The prover demonstrates to the verifier, using the SHVZK protocol, that $\mathcal{E}_{pk}(b_j)$ is the encryption of 1 under $\mathcal{E}_{pk}$. The verifier provides her challenge chal in the normal execution of the protocol, after the prover's first message.

2. The prover uses chal to simulate proofs that the other bit encryptions $(c_i)_{i \neq j}$ are also encryptions of 1. These simulations are possible because the proof protocol is special honest-verifier zero-knowledge. These simulated proofs are "lies," of course, since the ciphertexts in question are actually encryptions of 0.

3. The receipt is then composed of all the proof transcripts.

The assistant verifies all transcripts, certifying that all of them are correct. Only the verifier—who saw that only the proof for index j was correctly, sequentially performed—knows which transcript indicates a true proof.

Real-Life Constraints. The black-box-based construction is useful to understand the general structure of our AHIP protocols for proving an integer-in-range plaintext. However, it is critical to note that a typical, human verifier *isn't powerful enough to successfully execute this generic protocol.*

Consider, specifically, the bit length of the SHVZK protocol messages: they are simply too long for a human verifier to remember given any reasonable security parameter. The bit lengths of the prover's first message and verifier's challenge must be small enough for the verifier to remember and compare: 3 or 4 ASCII characters at the most. It isn't possible to achieve such short messages without a specialized proof construction. Note in particular that one cannot arbitrarily restrict the domain of prover messages, since this restriction cannot be respected at simulation time, when the prover's first message is computed as a function of the extracted challenge.

High-Level Construction of our Specialized Protocols. The central idea in both of our proposed proof protocols is to have the prover generate a specially-crafted ciphertext, $\mathbb{c}_i = \mathsf{BitEnc}_{pk}(b_i)$, for each $c_i = \mathcal{E}_{pk}(b_i)$ in the original public input. The special form of $\mathsf{BitEnc}_{pk}(b_i)$ enables a proof protocol that lends itself to particularly short prover and verifier messages in all cases. Soundness is optimally high given the size of the verifier challenge, and the security of the ciphertexts remains overwhelmingly high. The Prover $\mathcal{P}$ also proves that $\mathbb{c}_i$ encodes the same value as c_i, a proof which can be entirely checked by the assistant $\mathcal{A}$ and which, surprisingly, requires no additional verifier challenge.

Communication Model We consider two mechanisms for the Prover to communicate with the Verifier. First, the receipt provides a permanent, append-only channel: once the prover prints some data, the verifier can see it, and the prover cannot modify it. This receipt will eventually be handed to the Assistant, of course, so it cannot be the only means of communication for the Prover, since the Verifier is expecting to obtain assurance regarding the private input which the Assistant shouldn't learn. Thus, in addition, the prover has a private channel to the verifier whose content is not printed on the receipt. This is likely to be the screen of a computer in some isolation booth. Anything sent over this private channel must take into account the severe computational limitations of the verifier.

In the other direction, the verifier is expected to have some basic data entry mechanism for short strings. This may even be a simple barcode reader, as the verifier does not need to communicate much other than a challenge string, which can be easily selected out of a set of pre-printed challenge tickets.

The $\mathsf{BitProof}$ Protocol. We now give a brief overview of our first proposed protocol, based heavily on Neff's construction. Consider α the bit-length of a string that a human verifier can easily compare, e.g. 24 bits (4 human-readable ASCII characters). Consider the original cryptosystem $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ used to encrypt the public input $(c_1 = \mathcal{E}_{pk}(b_1), \dots, c_z = \mathcal{E}_{pk}(b_z))$, and assume this cryptosystem is additively homomorphic, e.g. Exponential El Gamal, where, if $pk = y = g^x \bmod p$:

$$\mathcal{E}_{pk}(m;r) = (g^r, g^m y^r).$$

Note that it is no problem to use Exponential El Gamal here, because we only perform decryption on a limited plaintext space—usually only 1 and 0: no discrete logarithm computation will be required. We simply care about the additive homomorphism. Thus:

1. The prover prepares special bit-encryptions ($\mathbb{c}_1 = \mathsf{BitEnc}_{pk}(b_1), \mathbb{c}_2 = \mathsf{BitEnc}_{pk}(b_2), \ldots, \mathbb{c}_z = \mathsf{BitEnc}_{pk}(b_z)$) with the BitEnc algorithm and the *same public key pk* , as follows:

   $$\mathsf{BitEnc}_{pk}(b) = [u_1, v_1], [u_2, v_2], \ldots, [u_\alpha, v_\alpha]$$

   where all the u_i and v_i values are encryptions of either 0 or 1, using $\mathcal{E}_{pk}$. If $b = 1$, both elements of each pair encrypt *the same bit*: $(\mathcal{D}(u_i), \mathcal{D}(v_i)) = (1,1)$ or $(0,0)$. If $b = 0$, both elements of each pair encrypt *opposite bits*: $(\mathcal{D}(u_i), \mathcal{D}(v_i)) = (1,0)$ or $(0,1)$. Note that, given a bit b, there are two plaintext choices for each pair within $\mathsf{BitEnc}(b)$. All pairs are not the same: the exact selection of bits is decided by the randomization value provided as input to BitEnc by the Prover. See Figure 5-2.

2. The prover prints the sequence $(\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z)$ on the receipt (which the verifier can see) and privately sends a string ChosenString to the verifier, which is the concatenation of the plaintext bit choices for the bit encryption $\mathbb{c}_j$. Recall that $b_j = 1$, and thus that both elements of each $[u_k^{(j)}, v_k^{(j)}]$ pair for all $k \in [1, \alpha]$ encrypt the same bit. It is these bits, of which there are α since there are α pairs of u's and v's, which are concatenated into ChosenString. Note how ChosenString is of length α bits, which is short (4 ASCII characters.)

3. Then, the verifier provides an α-bit challenge string chal. The bits of the challenge indicate for which half of each $[u, v]$ pair the Prover should reveal the corresponding plaintext. For example, if chal = 011000, the Prover opens up [1] for all bit encryptions, all u_1, all v_2, all v_3, all u_4, all u_5, and all u_6. This reveals an α-bit string for each index of the z bit encryptions, $\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z$. For $\mathbb{c}_j$, the challenge should not affect the revealed string, since both elements of any given pair encrypt the same bit: the revealed string should be exactly ChosenString, the value sent by the Prover in the first message. In addition, the homomorphic property of $\mathcal{E}$ allows the Prover to quickly prove that BitEnc_i and c_i have the same plaintext without involving the verifier (details in Section 5.5).

4. For all other bit encryptions $(\mathbb{c}_i)_{i \neq j}$, the result is effectively a random α-bit string which depends on the initial random choices of the prover and the Verifier's random challenge. The Prover can thus complete the simulation for these other bit encryptions by setting

[1] By opening up, we mean here that the Prover simply reveals the randomization value with which he encrypted the ciphertext in questions, thereby revealing the plaintext and the witness of encryption. These randomization values are printed on the receipt.

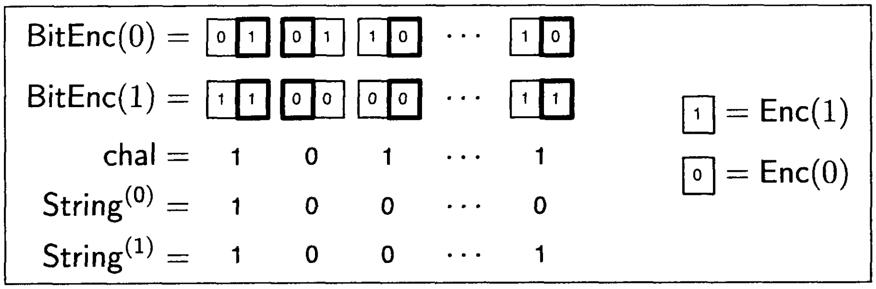


Figure 5-2: The BitEnc operation, where each boxed bit b indicates $\mathcal{E}_{pk}(b)$. Note how chal indicates which encrypted bits to reveal: a 0-bit indicates that the left half of each pair should be opened, while a 1-bit indicates that the right half of each pair should be opened. Note also how $\mathsf{String}^{(0)}$ and $\mathsf{String}^{(1)}$ are thus constructed from the selected, thickened boxed bits.

$\mathsf{String}^{(i)}$, his first message in the proof of $\mathbb{c}_i$ and the equivalent of ChosenString for that proof's transcript, to this now-revealed string.

5. The bit-encryptions are all printed on the receipt, as is the challenge, the strings $\mathsf{String}^{(i)}$, and the revealed randomization values. Only ChosenString = $\mathsf{String}^{(j)}$ was truly committed to ahead of time to the Verifier, and the Verifier need only remember this one α-bit string and check that the same string appears next to index j on the receipt. The soundness of this protocol is $(1 - \frac{1}{2^\alpha})$, and the size of the bit encryptions (and thus the receipt) is linear in α (and κ, the security parameter).

Efficient Implementation. We also provide BetterBitProof, an alternative implementation of the same proof with the same user interface, using an optimized specially-crafted cryptosystem BetterBitEnc. The size of each bit-ciphertext is constant instead of linear, assuming $\alpha < \kappa$ where κ is the security parameter. The bit encryption is performed in $SO(2, q)$, where each bit is represented as a 2-vector element, and each vector is then element-wise encrypted using exponential El-Gamal in a q-order subgroup. The most important aspect of this implementation is its significant efficiency improvement in terms of receipt representation and required computation per ballot. Two El Gamal ciphertexts per candidate in the prover's first message and two El Gamal randomization values per candidate in the prover's third message suffice to achieve overwhelming soundness. The details of this protocol are given in section 5.6.

5.1.5 Previous & Related Work

Humans and Basic Cryptography. In 1994, Naor and Shamir introduced visual cryptography [121], a secret-sharing method that uses visual superposition as a kind of human XOR. A number of improvements were proposed to improve contrast and usability [59, 60]. In 1997, Naor and Pinkas introduced visual authentication [120], a technique to help humans authenticate computers using visual cryptography methods.

Chaum used visual cryptography to build one of the first encrypted voter receipt techniques in 2004 [40]. He and others are continually improving this technique [41], which provides the only known alternative to the scheme we propose here.

Humans and Interactive Proofs. The literature is rich with protocols that allow humans to securely authenticate to computer systems. For example, Hopper and Blum [91] were the first to use the Learning Parity with Noise (LPN) problem as a basis for human authentication, based on Blum et al.'s earlier LPN-related work [21]. Interestingly, the low-computational-power constraint now applies to certain computing environments, like RFIDs, which led Juels and Weis to define an LPN-based authentication system for RFIDs [102].

On another front, a plethora of works deal with a human proving to a computer that he is, indeed, human. A large number of such protocols, often called CAPTCHAs (Completely Automated Public Turing Test to Tell Computers and Humans Apart), use hard Artificial Intelligence problems to exploit tasks that only humans can do efficiently, e.g. reading garbled text. These were first formalized by Ahn et al. [170, 171].

AHIPs are, to our knowledge, the first formalization of an interactive proof where a human being plays the role of *the verifier* as opposed to that of the prover. As both human-based authentication and CAPTCHAs have been classified as HIPs, Human Interactive Protocols, it makes sense to classify Assisted-Human Interactive Proofs as a third kind of HIP.

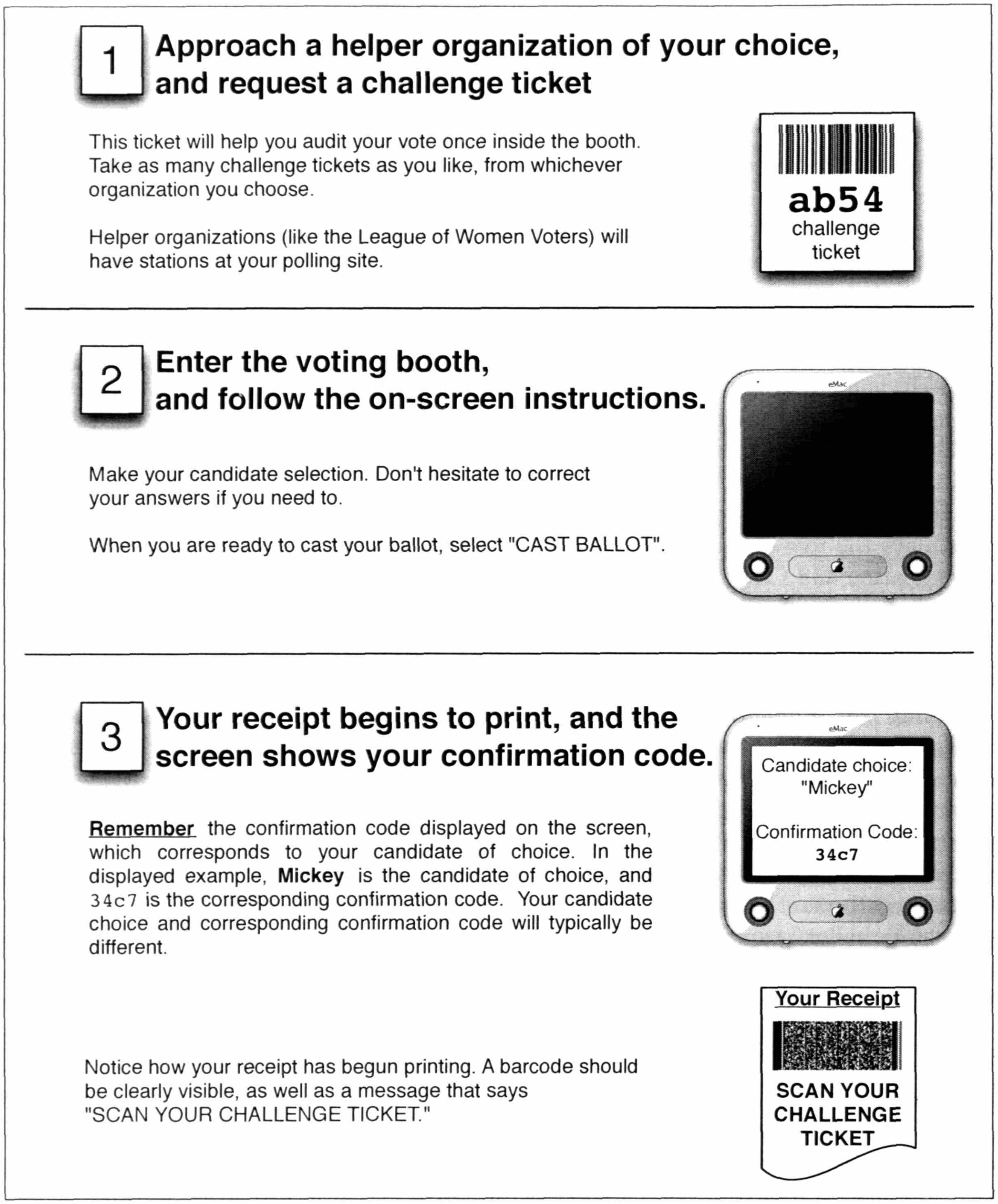
Receipt-Free Elections. Benaloh and Tuinstra first introduced and implemented receipt-free voting [20]. Enhancements and variations were quickly proposed [151, 125, 129], under the assumption of a private channel between the administrator and voter. Canetti and Gennaro [33] showed how to accomplish incoercible multiparty computation without this assumption. The voting literature is plentiful: a large number of receipt-free schemes have been proposed [46, 90, 13], all of which assume a computationally capable voter. Some schemes even achieve optimal ballot secrecy [106].

Assisted-Human Interactive Proofs are, to our knowledge, the first formal treatment of receipt-free ballot preparation with a human voter without fully trusted hardware. Our proposal requires a private channel between the voting machine and the voter.

5.1.6 Organization

In the first half of this chapter, we explore the concept of ballot casting assurance. Section 5.2 frames ballot casting assurance in the context of election auditing, including the well-known concept of universal verifiability. Section 5.3 provides the logistical details of implementing ballot casting assurance, assuming an AHIP protocol exists for the proof portion.

In the second half of this chapter, we explore the details of AHIP protocols. Section 5.4 presents the AHIP definition in detail, proves that an AHIP protocol is also witness-hiding, and gives an overview of the generic structure of an AHIP protocol for voting. Sections 5.5 and 5.6 provide the details of our two implementations, including proofs that they fit the definition provide in section 5.4.



1 Approach a helper organization of your choice, and request a challenge ticket

This ticket will help you audit your vote once inside the booth. Take as many challenge tickets as you like, from whichever organization you choose.

Helper organizations (like the League of Women Voters) will have stations at your polling site.

ab54
challenge
ticket

2 Enter the voting booth, and follow the on-screen instructions.

Make your candidate selection. Don't hesitate to correct your answers if you need to.

When you are ready to cast your ballot, select "CAST BALLOT".

3 Your receipt begins to print, and the screen shows your confirmation code.

Candidate choice: "Mickey"

Confirmation Code: 34c7

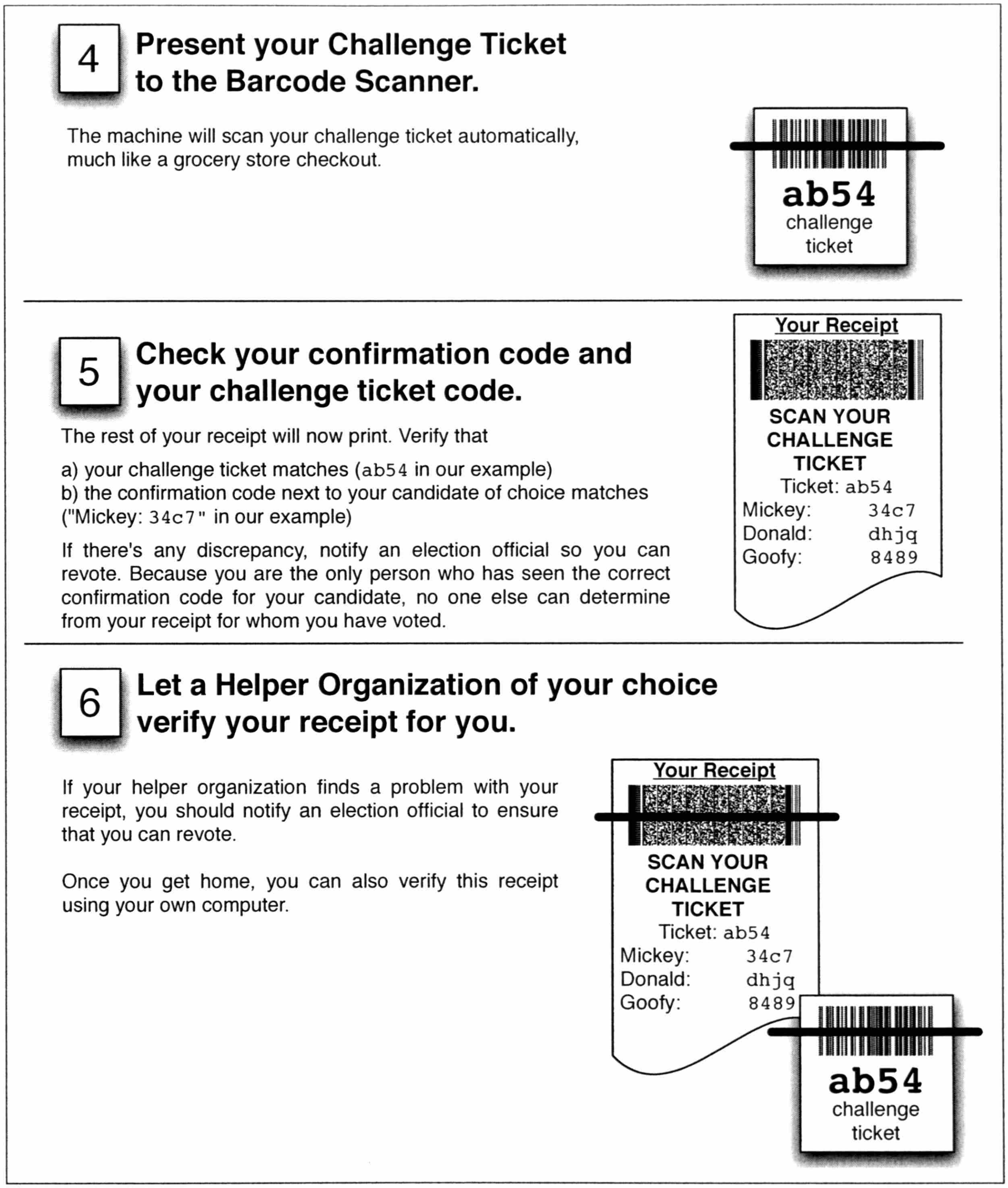
Remember the confirmation code displayed on the screen, which corresponds to your candidate of choice. In the displayed example, **Mickey** is the candidate of choice, and 34c7 is the corresponding confirmation code. Your candidate choice and corresponding confirmation code will typically be different.

Your Receipt

SCAN YOUR CHALLENGE TICKET

Notice how your receipt has begun printing. A barcode should be clearly visible, as well as a message that says "SCAN YOUR CHALLENGE TICKET."

Figure 5-3: Page #1 of Voter Instructions for the Ballot Casting Assurance / Assisted-Human Interactive Proof protocols presented here.



4 **Present your Challenge Ticket to the Barcode Scanner.**

The machine will scan your challenge ticket automatically, much like a grocery store checkout.

ab54
challenge ticket

5 **Check your confirmation code and your challenge ticket code.**

The rest of your receipt will now print. Verify that

a) your challenge ticket matches (`ab54` in our example)
b) the confirmation code next to your candidate of choice matches ("Mickey: `34c7`" in our example)

If there's any discrepancy, notify an election official so you can revote. Because you are the only person who has seen the correct confirmation code for your candidate, no one else can determine from your receipt for whom you have voted.

6 **Let a Helper Organization of your choice verify your receipt for you.**

If your helper organization finds a problem with your receipt, you should notify an election official to ensure that you can revote.

Once you get home, you can also verify this receipt using your own computer.

Figure 5-4: Page #2 of Voter Instructions for the Ballot Casting Assurance / Assisted-Human Interactive Proof protocols presented here.

5.2 Auditing an Election

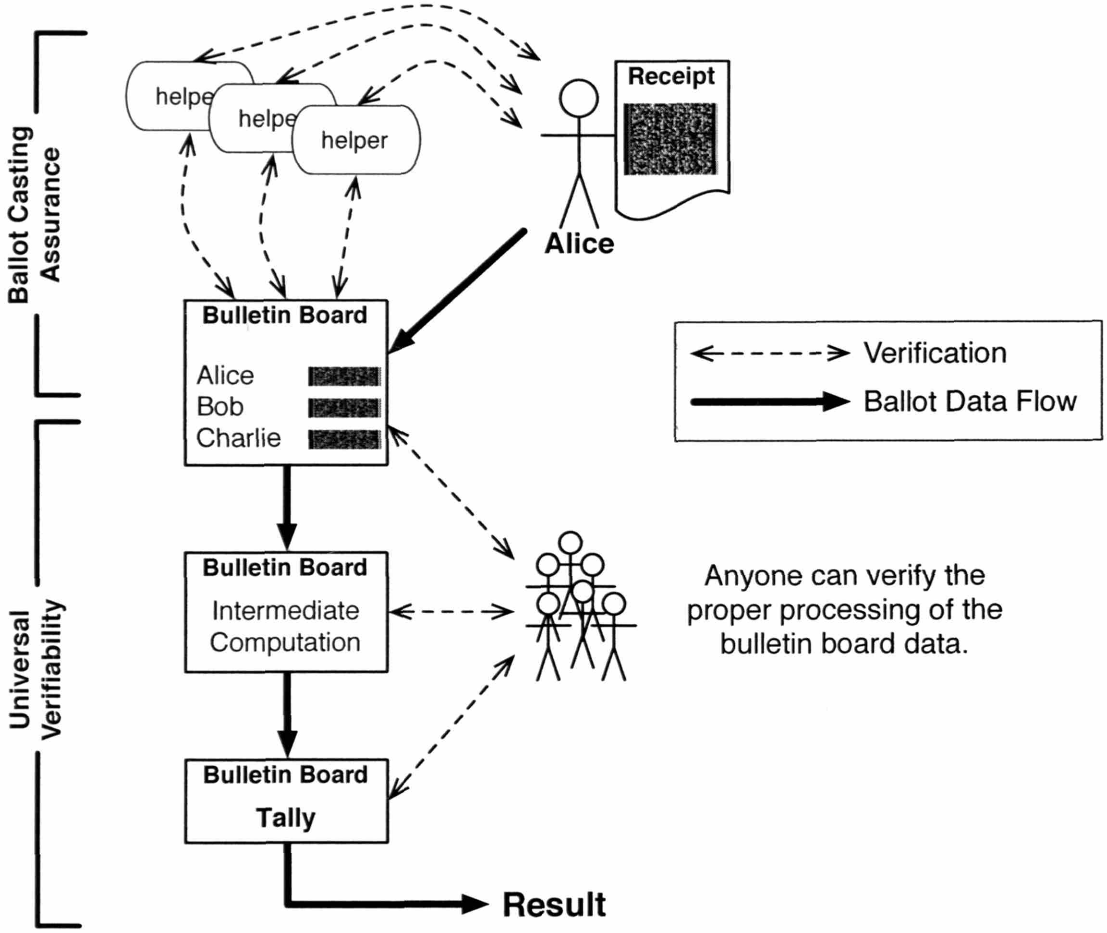


Figure 5-5: Auditing an Election—Ballot Casting Assurance describes how Alice can be certain that her vote was recorded and posted on the bulletin board as she intended, while Universal Verifiability pertains to the tallying process.

With the introduction of the secret ballot, the public lost the ability to directly audit an election. Once Alice casts her ballot, she also hands off all verification ability to the election administrators. Given the complete dissociation of voter identity and ballot content, even election administrators are left with only crude mechanisms for verification: the dominant measure of election reliability is the residual vote, which indicates only the difference between the total number of votes cast and tallied.

High-level voting system verification goals have been explored in prior literature:

1. **Cast as intended:** the ballot is cast at the polling station as the voter intended.

2. **Recorded as cast:** cast ballots are preserved with integrity through the ballot collection process.

3. **Counted as recorded:** recorded ballots are counted correctly.

4. **Eligible voter verification:** only eligible voters can cast a ballot in the first place.

With classic security processes and methods, it is unclear how to achieve all of these verifiability tasks directly. Instead, current voting practices rely on delegating these tasks to election officials.

Universal Verifiability. Over the last twenty years, many cryptographic schemes have been developed to address the *counted-as-recorded* and *eligible voter verification* verification tasks. These schemes are generally said to implement *universal verifiability*, as any observer can verify that all collected ballots have been correctly anonymized, preserved, and tallied.

Generally, these schemes use an authenticated bulletin board, where ballots are posted as ciphertexts together with the voter's plaintext identity. Individual voters can verify that their encrypted ballot is correctly posted on the bulletin board. All observers can check that only eligible voters cast a ballot and watch the tallying process on encrypted votes, which is usually implemented as an anonymizing mixnet or a homomorphic aggregation.

Ballot Casting Assurance. Ballot casting assurance fills the remaining auditability gap: *cast as intended* and *recorded as cast.* How can Alice obtain verification that her intent has been recorded appropriately? If she determines that her vote has been recorded incorrectly, how can she rectify the situation? Were it not for the secret ballot, these questions could be answered trivially.

Two important properties of ballot casting assurance should be emphasized:

- **End-to-end verification**: typical voting security analyses distinguish the properties "cast as intended" and "recorded as cast." This distinction is an artifact of a chain-of-custody approach to verification, where each step must be independently verified. Ballot casting assurance need not concern itself with this specific mechanism for verification. The requirement is end-to-end, from the voter's brain to the bulletin board.

- **Direct verification**: Alice, the voter, should get *direct* and *immediate* verification that her vote was correctly recorded. Mediating the verification via election officials, or delaying the verification until it is too late for Alice to rectify the situation, is insufficient.

Of course, the ballot secrecy requirement remains. Whatever verification is enabled by ballot casting assurance, it must also be incoercible. Even if Alice wishes to sell her vote, nothing she does should give her a significant edge in successfully convincing a third party of how she voted.

A number of recent proposals, including Chaum's visual cryptography ballot [40], its variants [41] including PunchScan [66], and Neff's encrypted receipt scheme MarkPledge, offer solutions that address these requirements: it is possible to give Alice direct verification of her vote without providing so much information that she can prove her vote to a third party, all given conservative assumptions of the voter's computational ability—i.e. we cannot expect a voter to perform complex math. These schemes provide the cryptographic basis for ballot casting assurance in a secret-ballot setting.

5.3 Ballot Casting Assurance

We now consider the voting environment required to achieve ballot casting assurance. We leave the detailed AHIP protocol description for Sections 5.5 and 5.6, and assume, for now, that such a protocol exists: Alice, the voter, privately interacts with a voting machine, which provides a receipt that the Assistant can verify for Alice without learning her vote. The goal is to provide Alice with immediate and direct verification that her vote was correctly recorded, before she leaves the polling location. We note that, when this is achieved, failure recovery by individual revoting becomes a possibility.

5.3.1 Setup

We consider that the election officials have properly generated (pk, sk), the election keypair, using a typical threshold scheme so that no single party holds the secret key. All voting machines have access to pk, as do all voters and observers.

Helper Organizations. We assume the presence of helper organizations at the polling location. These organizations are political or advocacy groups, such that various opinions and voter preferences are represented at a given polling location. These helpers do not have access to any secret election information, only the public election parameters including pk. They have some computing equipment at the location, though there is no need for network access. Clearly, this setup is not realistic for all polling locations: in some cases where polling locations are too small to support helper organizations, voters have to rely on helpers located outside the polling location, where, if helpers detect an error, the complaint process becomes more difficult.

These helper organizations play the role of the AHIP Assistant $\mathcal{A}$. The worst a dishonest assistant can do is claim a valid ballot to be incorrect; in this case, Alice can ask multiple assistants to check her encrypted ballot. She can even check it herself once she gets home and uses her home computer. If a disagreement occurs, the assistants must provide detailed proofs of their judgment, so that election officials can determine who is providing incorrect help.

The Voting Machine. The voting machine in our model presents a typical computer screen interface. In addition, the machine has a scrolling receipt printer: as data is printed on the receipt, the receipt is pushed out so that the voter can see the data as soon as it is printed, even if the receipt is not yet complete. We assume that the printer cannot retract the paper, erase or modify data in any way.

The voting machine is effectively the Prover in our model. It has two mechanisms for "sending messages": on its screen, or on the printer. In both cases, the human verifier can see the messages. The receipt printing mechanism ensures that "sending a message to the receipt" is effectively a commitment, since it cannot be undone.

We also assume that each voting machine generates a keypair for applying digital signatures to each receipt it prints. The public keys of the voting machines should be collected and certified by the appropriate election officials prior to the beginning of election day. Careful planning of this process is required, though we note that, even for today's optical scanning and DRE voting machines, this operation should become standard practice.

Severely Limited Verifier. Alice, the voter, is severely computationally limited. We assume that she can reliably compare or copy short strings, but nothing more.

Private Input and the Secret Receipt. Given m, Alice's plaintext vote, and c, the encrypted vote created by the voting machine, Alice expects a proof that (c, m) is a valid ciphertext/plaintext pair under public key pk. In our model, m is thus the private portion of the input string, while c is the public portion: since it is encrypted, there is no risk in revealing it, assuming the cryptosystem is semantically secure. The scrolling receipt produced by the voting machine is the receipt in the AHIP model.

The Honest Verifier's Challenge. As we cannot always assume that a human will generate a "random enough" string for our model, we expect Alice to obtain a *challenge ticket* from one of her assistant organizations before she enters the isolation booth. This challenge ticket has a number pre-printed on it, along with a corresponding optical bar code, which represent the challenge string chal to be used in the proof inside the isolation booth. To ensure the technical constraint that this protocol be simulatable (and thus AHIP), the challenge ticket also includes a commitment to the challenge, $commit(\mathsf{chal})$, with its own corresponding optical bar code.

Note, however, that this process is only meant to help the voter pick random enough challenges. It is in no way necessary for the model, and $commit(\mathsf{chal})$ does not need to be input to the machine until the appropriate moment in the protocol, after the encrypted vote has been printed on the ballot. In addition, assuming an honest voting machine, the helper organization cannot determine how Alice voted merely by crafting a special chal. Recall that we assume, in this work, that the voting machine is not trying to leak Alice's vote, though, of course, we do want to ensure that the voting machine cannot encrypt Alice's vote incorrectly.

The Encrypted Ballot. Once Alice has selected her candidate, option j, the voting machine produces c. In our setting, c is composed of values $c_1, c_2, \ldots, c_z$, one for each candidate, where $c_j = \mathcal{E}_{pk}(1)$ and $\forall i \neq j, c_i = \mathcal{E}_{pk}(0)$. We assume that the voting machine produces a well-formed ballot, meaning that exactly one of the ciphertexts is $\mathcal{E}(1)$, and all of the others are $\mathcal{E}(0)$ (Section 5.7 explains this assumption.) The ballot can then be used in a mixnet setting (see Chapter 3.)

5.3.2 Ballot Casting

1. Alice consults a helper organization of her choice to obtain her "challenge ticket," which includes the committed challenge and the challenge itself.

2. In the isolation booth with the voting machine, Alice proceeds with her ballot selection and verification. She uses her challenge ticket to provide $\mathit{commit}(\mathsf{chal})$ and chal, using an input device such as a bar code scanner. By the end of the process, Alice has a physical receipt in hand, and she has verified that:

   (a) the verification code next to her candidate of choice matches what she saw on screen

   (b) the challenge printed on the receipt matches the challenge on her challenge ticket.

   In addition, the voting machine is now expected to digitally sign the receipt.

3. The voting machine immediately posts the encrypted vote along with Alice's name to the digital bulletin board.

4. Alice hands her receipt to a helper organization of her choice. The helper verifies that the ballot has been correctly posted, that it is internally consistent, and that it is correctly signed by the voting machine.

   If the verification is in any way unsatisfactory to Alice, she can simply return to step 1 and vote again. When she does, her new vote replaces her old vote on the bulletin board. The bulletin board maintains a history of all ballots cast by Alice, noting which is the last one to be used for tallying.

   Note that this revoting process is similar to current procedures for optically-scanned ballots when the voter makes a mistake and requests a new ballot. As an added bonus here, the history of revoting is kept, in case it ever becomes a source of fraud. In particular, all votes produced by all voting machines are recorded to prevent one voter from later attempting to replace her vote with that produced for another individual.

5. Once Alice is satisfied with her receipt and the verifications performed by various helpers, she leaves the polling location with her receipt in hand.

6. At her discretion, Alice can leave a copy of her voting receipt with any number of helper organization.

Note that, if it is too onerous to post immediately to the bulletin board and verify, live, that the bulletin board has been updated, the process can be modified so that helper organizations only check the voting machine's digital signature. In this case, it might also be useful to witness election officials verifying the signature, as they can be made liable in case the ballot goes missing at a later point.

5.3.3 Complaint & Correction Process

After having left the polling location, Alice can check, using software she trusts—even software she may have written herself – that her vote has been correctly posted and tallied. We consider what happens if Alice finds that her ballot has gone missing from the public tally. We consider, in particular, the possibility of *late revoting.* This process provides significantly increased ballot casting assurance, though we note that it requires significant care to ensure that no new weaknesses are introduced (See Section 5.3.5).

Before a prescribed complaint deadline (maybe 24 hours after polls close), Alice can follow these steps:

1. Present her receipt and identification to the complaint center.

2. If the voting machine signature on the receipt is *incorrect*, the complaint fails and no further action is taken.

3. If the receipt's signature is *correct*, the encrypted ballot is compared to the claimed encrypted ballot on the bulletin board. If it is the same ballot, then there was no mistake, and no further action is taken.

4. If the signature is correct and the bulletin board shows a different ballot than the receipt's, and *no other voter's cast encrypted ballot matches this submitted claim*, election officials should update the bulletin board to reflect Alice's corrected vote.

5. If election officials refuse to update a vote, the voter may turn to a political helper organization that can perform exactly the same checks as the election official and submit a formal complaint (possibly using the press.)

One should assume that most voters will never care to verify and complain on their own. Thus, as described earlier, Alice has the ability to simply hand her ballot to a verification agent who can do all checks on her behalf. If this agent finds that Alice's vote has been mishandled, it may complain on her behalf.

5.3.4 Trust Assumptions

The tweaks we propose do not significantly alter the trust assumptions of typical cryptographic voting schemes. We review these assumptions here, and explain what we expect of the newly-introduced *helper organizations.* It is important to distinguish the two major classes of malicious behavior: those which affect *tally correctness*, and those which affect *ballot secrecy.*

Within ballot secrecy, it is also important to distinguish *subliminal channel attacks*, where the election equipment and data violate secrecy, and *side-channel attacks*, where some external equipment, e.g. a cell phone camera, is used to violate the privacy of the voting booth. This latter class of attacks is extremely problematic in any election scenario, and we do not attempt to address it here.

Voting Machines. The voting machine is *not trusted* for correctness. The proof protocol ensures that the voting machine cannot cheat a single voter with more than very small probability (10^{-6}), which rules out a cheating voting machine having any noticeable impact on the election. In the current model, the voting machine is trusted not to perform subliminal channel attacks on ballot secrecy. The voting machine is expected to keep its secret key private, as it, and its manufacturer, would be blamed for malicious uses of this key.

Bulletin Board. The bulletin board is *not trusted*: it is only a common conduit for authenticated content from the voting machines. It is expected that the various helper organizations will keep an eye on the bulletin board, including ensuring, via regular auditing, that everyone sees the same view of the bulletin board. This can be performed using well-understood auditing techniques, including hash trees [141] or the more advanced authenticated broadcast protocols [110].

Helper Organizations. We assume that at least one helper organization is honest and running correct software. Some helper organizations may be dishonest for certain voters, though it is expected that these organizations will be mutually distrusting, such that it is highly unlikely that all organizations will be dishonest for a given voter. It is important to note that, if a helper organization incorrectly verifies a ballot, the voter may easily consult another organization, likely a competing political party. Thus, a helper organization takes a big risk if it lies.

For ballot secrecy, the helper organizations are completely *untrusted*, as they may have strong incentive to coerce voters. Note that, even if a helper organization provides the "challenge ticket," it cannot violate ballot secrecy.

Voter's Home Software. For correctness, we assume that a small, but significantly non-zero, fraction of voters will run the correct verification software. We assume that it is very difficult for an adversary to target a particular voter at the polling location, corrupt her helper organizations, and corrupt her home computer simultaneously. We assume that, even if an adversary accomplishes this complex attack against one voter, it isn't easily scalable to more voters.

We assume that all voters are potential adversaries when it comes to coercion. We note that, in any scheme, a voter may take a photo of her ballot using a camera-phone: in Hong Kong in 2004, mainland Chinese residents were asked by authorities to have their relatives mail them camera-phone pictures of their ballot [8]. This is the *side-channel attack* previously mentioned, which no current voting system addresses, and which our proposal does not exacerbate.

5.3.5 Threats

The verification, complaint, and correction processes significantly increase voter confidence. At the same time, they open up new avenues for attack. With the trust assumptions defined in the previous section, we now outline some possible attacks and approaches to countering them. We do not attempt to address every issue: ballot casting assurance requires much continued research, and we cannot expect to quickly solve all open problems. We also note that, while these issues are important, they are generally less worrisome than the potential for abuse in today's unverified elections.

Incorrect Verification. A helper organization might act maliciously in verifying ballots at the polling location, claiming that good ballots are bad. This may be a particularly useful tactic for one political party to use in a precinct that is known to be dominated by another political party. The system presented here attempts to mitigate such issues by ensuring that *anyone* can perform all ballot verification tasks, thereby ensuring *public oversight* of the system. A helper organization can be required to present objective, verifiable proof that a ballot is malformed.

Refusal to Replace. During the complaint period, malicious election officials may delay or even prevent the replacement of a ballot on the bulletin board, even when presented with a valid voter complaint. To address this, the system once again ensures that *anyone* can perform the ballot verification, which should enable a voter to escalate a complaint based purely on publicly available data. One also notes that, with all ballots now encrypted, the complaint process can be performed in the presence of mutually distrusting observers.

Abusive Replacement. An attacker may attempt to replace an honest voter's ballot during the complaint period. One should note that this is an entirely new threat given the complaint-and-replace process. The system provides a first technical line of defense: the bulletin board records all ballots produced by eligible voting machines. Bob cannot simply use one of his overwritten ballots to replace Alice's ballot, as any bulletin board observer would detect this manipulation. The system further mitigates this risk by adding a strong identification requirement to the ballot replacement process.

One might also consider legal and procedural disincentives, such as requesting signed affidavits of ballot replacement by the complainant, as well as notifying Alice by mail that her ballot was changed in order to enable fraud detection and investigation. This last measure is particularly important to prevent a new kind of ballot box stuffing, whereby a malicious election official might replace a number of ballots just before the close of the complaint period.

Subliminal Channels. As detailed by Karlof et al. [104], the secret receipt's randomness provides a subliminal channel to a malicious voting machine that wish to leak the ballot plaintext. It will be important to develop methods for removing these channels in future work.

5.3.6 Other Implementations

Ballot casting assurance is implementable via other techniques than the one described here.

Chaum Receipts. Chaum's visual cryptography receipts and related schemes—e.g. PunchScan—offer similar functionality to MarkPledge. In particular, PunchScan [66] provides direct verification for individual voters with soundness 1/2: each voter splits the ballot in two, and selects which layer to take cast. Information later revealed by election officials confirms the correctness of the ballot form.

There is, of course, a significant soundness difference, as MarkPledge easily provides $1 - 10^{-6}$ soundness per voter, while PunchScan provides soundness of 1/2. However, in the voting scenario, soundness of 1/2 is likely sufficient: only attacks that affect a tiny number of ballots might realistically go undetected.

Thus, PunchScan, like MarkPledge, is a voting protocol that provides ballot casting assurance. One should note, however, that certain variants of this scheme, in particular the early versions of Prêt-a-Voter [41], are slightly weaker: the cut-and-choose is performed by election officials because it is logistically too complicated to be performed by voters. A recent variation of the scheme [149] addresses this problem with on-demand printing of ballots in the voting booth, though this change requires more complex equipment at the polling location.

Current Systems. Current election systems, be they DRE, optical scan, or paper based, do *not* provide ballot casting assurance, as they completely separate the voter's identity at the moment of ballot casting, preventing all future direct verification. Even election administrators are limited in the verification tasks they can accomplish. Many types of fraud—a relatively small number of stuffed or lost ballots—may go undetected. When an inconsistency is detected, e.g. a high residual vote rate, the only possible recovery may be to rerun the entire election.

The Impact of Cryptography. An interesting conjecture is that direct verification of secret-ballot elections is only possible using cryptographic techniques. This conjecture should be further explored, and, if found to be true, should provide strong motivation for significant continued research in usable cryptographic voting techniques.

5.4 Model and Definitions

We now turn to the formal model of Assisted-Human Interactive Proofs. We begin with some notation and math preliminaries. Then, we propose a definition of AHIP protocols and describe the basic technique that underlies both implementations of the voting integer-in-range plaintext proof. We describe these two implementations in the next two sections.

5.4.1 Notation

We model cryptographic protocol participants using the established convention of Probabilistic Polynomial Time (*PPT*) Interactive Turing Machines (*ITM*). Each participant is designated by a single calligraphic uppercase letter, e.g. $\mathcal{B}$, whose initial inputs are specified in functional notation, e.g. $\mathcal{B}(x, y)$.

The interactive protocol between two parties $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ takes as inputs x_A and y_A and $\mathcal{B}$ takes as inputs x_B, is denoted $\langle \mathcal{A}(x_A, y_A), \mathcal{B}(x_B) \rangle$. The output of party $\mathcal{A}$ is denoted $\mathsf{output}_{\mathcal{A}} \langle \mathcal{A}(x_A, y_A), \mathcal{B}(x_B) \rangle$. The view of party $\mathcal{B}$ during the protocol execution, defined as the ordered sequence of messages sent and received by $\mathcal{B}$ (not including its initial input), is denoted $\mathsf{view}_{\mathcal{B}} \langle \mathcal{A}(x_A, y_A), \mathcal{B}(x_B) \rangle$. For a zero-knowledge definitional basis, we refer to [75], and for witness hiding protocols, we refer to [62]. Note that we will denote $\mathcal{A}$ the assistant, and Adv the adversary.

5.4.2 Setup

Consider a computationally-limited verifier, modeled as two *ITM*s: the interactive verifier $\mathcal{V}_{\mathsf{int}}$ and the non-interactive checker $\mathcal{V}_{\mathsf{check}}$. We stress that this separation of $\mathcal{V}_{\mathsf{int}}$ and $\mathcal{V}_{\mathsf{check}}$ is for definitional clarity only: both *ITM*s taken together represent the same human Alice. The usual setting initially applies: a prover $\mathcal{P}$ interacts with $\mathcal{V}_{\mathsf{int}}$ to prove that a string x has a certain property—e.g. it is a correct tuple (c, m, pk) such that c is the encryption of m under the public key pk.

More formally, the assertion being proven is that the string x is in language $\mathcal{L}$. Once the interactive portion of the proof is complete, $\mathcal{P}$ outputs a *secret receipt*, denoted $\mathsf{receipt}$, while $\mathcal{V}_{\mathsf{int}}$ outputs some internal state vstate, which simply represents the "communication" between $\mathcal{V}_{\mathsf{int}}$ and $\mathcal{V}_{\mathsf{check}}$ (literally inside Alice's brain). $\mathcal{V}_{\mathsf{check}}$ performs basic consistency checks between $\mathsf{receipt}$ and vstate. Then, a *PPT*-capable assistant $\mathcal{A}$ examines $\mathsf{receipt}$ and performs further consistency checks, in particular those which $\mathcal{V}_{\mathsf{int}}$ and $\mathcal{V}_{\mathsf{check}}$ are computationally unable to perform. The interactive proof succeeds if both $\mathcal{V}_{\mathsf{check}}$ and $\mathcal{A}$ agree that it succeeds.

5.4.3 An Intuition

The Assistant should not learn as much as the Verifier. In particular, recall that, in a typical zero-knowledge proof, the Verifier learns (or already knows) the common input string x and receives a proof that $x \in \mathcal{L}$. In our model, we split the common input string x into two parts: $x = (x_{pub}, x_{priv})$, with x_{pub} the public input and x_{priv} the private input. The Assistant will learn x_{pub}, but should not learn x_{priv}, even though it is assisting the Verifier in checking that the *complete common input* $x = (x_{pub}, x_{priv})$ is in the language $\mathcal{L}$. In particular, the receipt obtained by the Verifier should not convincingly reveal x_{priv}, even if the Verifier is willing to reveal it.

In the voting setting, which is our driving example, x_{pub} is Alice's encrypted vote, while x_{priv} is its plaintext. Alice wants a proof that (x_{pub}, x_{priv}) is a correct ciphertext/plaintext pair: when her encrypted vote appears on the election bulletin board, she can be certain that it correctly captures her intent. However, no assistant should receive solid proof of her actual plaintext vote, x_{priv}, even if she is willing to sell her vote.

Private-Input Hiding. Intuitively, what does it mean for the receipt to hide the private input, even if the Verifier is willing to reveal it? In the voting setting, this is the notion of incoercibility, which has not been modeled in the human-verifier setting to date.

We propose that private-input hiding is a form of indistinguishability from the point of view of a potential coercer, an approach which was noted in various forms by Benaloh and Tuinstra [20], and Okamoto [129], and was particularly developed by Canetti and Gennaro [33]. Specifically, a coercer should not be able to distinguish between two voters, Sally and David. Sally (the "Seller") receives a proof of private input x_{priv}, and tells the coercer this value x_{priv} honestly. David (the "Double Crosser") receives a proof of private input $x'_{priv} \neq x_{priv}$, but tells the coercer that his private input was x_{priv}, just like Sally. If the protocol is private-input hiding, the coercer cannot distinguish between these two verifiers: no matter how much evidence Sally presents, David can fake evidence for his receipt that is indistinguishable.

Computational Hiding. In this work, we only consider computationally private-input hiding AHIPs. In other words, we assume that the Sally/David distinguisher is a *PPT ITM*. This is, in fact, required for the voting setting: the public input is the voter's encrypted ballot, and a decryption yields the voter's plaintext ballot, which is the private input. If the receipt were statistically private-input hiding, the receipt might not actually prove that $(x_{pub}, x_{priv}) \in \mathcal{L}$: it might prove that $(x_{pub}, x'_{priv}) \in \mathcal{L}$, which doesn't match the requirements of the voting scenario. This reasoning matches the proofs provided by Canetti and Gennaro regarding the inherent limitations of the concept of incoercibility [33].

Thus, we assume that x_{priv} is uniquely defined by a given x_{pub}, though it should not be computable in *PPT* without additional secret information. Of course, given the indistinguishability requirement, there are many possible public inputs x_{pub} for any given private input x_{priv}. In the formalization of this concept, we consider the specific constraints on the language for which we build proofs.

5.4.4 Formalization

We now consider a formal definition of Assisted-Human Interactive Proofs. As it is clear that AHIP protocols are applicable to languages of a certain specific form, we first define

Assisted-Human Interactive Proof Languages (AHIP Languages), a class of NP languages.

Definition 5-1 (AHIP Language) *An NP language $\mathcal{L}$ is an AHIP language if, given strings $x \in \mathcal{L}$ of length polynomial in the security parameter κ, and witness w such that $\mathcal{R}_{\mathcal{L}}(x, w) = 1$:*

- **Elements are pairs:** *each element x is a pair: $x = (x_{pub}, x_{priv})$. By x_{pub}, we denote the public input, and by x_{priv}, the private input.*
- **Pairs are sampleable:** $\exists \mathcal{S}_{\mathcal{L}} \in PPT$ *such that,* $\forall x_{priv}$ *where* $\exists x_{pub}$ *such that* $(x_{pub}, x_{priv}) \in \mathcal{L}$, *if* $(x_{pub}, w) \xleftarrow{R} \mathcal{S}_{\mathcal{L}}(x_{priv})$, $\mathcal{R}_{\mathcal{L}}((x_{pub}, x_{priv}), w) = 1$. *The output of $\mathcal{S}_{\mathcal{L}}$ on input x_{priv} is uniform over all values of x_{pub} for which $(x_{pub}, x_{priv}) \in \mathcal{L}$.*
- **Pairs are unique by public input:** $\forall (x_{pub}, x_{priv}) \in \mathcal{L}$, *and* $\forall (x'_{pub}, x'_{priv}) \in \mathcal{L}$, $x_{pub} = x'_{pub} \implies x_{priv} = x'_{priv}$.
- **Public input hides the private input:** $\forall (x_{pub}, x_{priv}) \in \mathcal{L}$, $\nexists \mathsf{BruteForceReveal} \in PPT$ *such that* $\mathsf{BruteForceReveal}(x_{pub}) = x_{priv}$ *with more than negligible probability.*

We are now ready to formalize the notion of an Assisted-Human Interactive Proof for an AHIP Language as defined above.

Definition 5-2 (Assisted-Human Interactive Proof) *Let:*

- *$\mathcal{L}$ be an AHIP language of strings $x = (x_{pub}, x_{priv})$, with sampling algorithm $\mathcal{S}_{\mathcal{L}}$ and corresponding relation $\mathcal{R}_{\mathcal{L}}$, which can be described as a PPT circuit* $\mathsf{Rel}_{\mathcal{L}}$.
- *$\mathcal{P}$ be a PPT ITM representing the prover.*
- *$(\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}})$ be a pair representing the honest verifier. $\mathcal{V}_{\mathsf{int}}$ is interactive while $\mathcal{V}_{\mathsf{check}}$ is a deterministic function, both with strict computational limitations designated* complimits. *$\mathcal{V}$ eventually outputs* vstate, *which $\mathcal{V}_{\mathsf{check}}$ uses to check the receipt at the end of the interaction.*
- *$\mathcal{A}$ be a polynomial-time algorithm representing the assistant.*
- $\Big\langle \mathcal{P}(x_{pub}, x_{priv}, w), \mathcal{V}_{\mathsf{int}}(x_{pub}, x_{priv}) \Big\rangle$ *be an interactive protocol for proving membership of (x_{pub}, x_{priv}) in language $\mathcal{L}$, with w a witness.*
- receipt *denote the "secret receipt" which $\mathcal{P}$ outputs after interacting with $\mathcal{V}_{\mathsf{int}}$.*

Then $\Big(\mathcal{P}, (\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}}), \mathcal{A}\Big)$ *is an* **Assisted-Human Interactive Proof** *for $\mathcal{L}$ if the following conditions hold:*

1. **Completeness.** *If $x \in \mathcal{L}$ and all parties are honest, then, with overwhelming probability, both the Verifier and Assistant declare success.*

   Formally there exists a negligible function $\nu(\cdot)$ such that, given security parameter κ, $\forall (x_{pub}, x_{priv}) \in \mathcal{L}, w$ s.t. $\mathcal{R}_{\mathcal{L}}(x, w) = 1$:

$$Pr\Big[(\mathsf{receipt}, \mathsf{vstate}) = \mathsf{output}_{\mathcal{P},\mathcal{V}_{\mathsf{int}}}\Big\langle \mathcal{P}(x_{pub}, x_{priv}, w), \mathcal{V}_{\mathsf{int}}(x_{pub}, x_{priv})\Big\rangle \quad ; \quad b_{\mathcal{V}} = \mathcal{V}_{\mathsf{check}}(\mathsf{vstate}, \mathsf{receipt}) \quad ; \quad b_{\mathcal{A}} = \mathcal{A}(\mathsf{receipt}) \quad : \quad b_{\mathcal{V}} = \mathsf{True} \quad \wedge \quad b_{\mathcal{A}} = \mathsf{True} \quad \Big] \geq 1 - \nu(\kappa)$$

   taken over the coin tosses of $\mathcal{P}$, $\mathcal{V}_{\mathsf{int}}$, and $\mathcal{V}_{\mathsf{check}}$.

2. **Soundness.** *If the input pair (x_{pub}, x_{priv}) is not in the language, then, with probability at least $1/2$, either the Verifier or Assistant will fail for any Prover. Soundness could be defined with overwhelming probability, but $1/2$ suffices in a number of settings.*

   Formally, given security parameter κ, $\forall (x_{pub}, x_{priv}) \notin \mathcal{L}, \forall$ non-uniform PPT ITM $\mathcal{P}^*$,

$$Pr\Big[(\mathsf{receipt}_{\mathcal{P}^*}, \mathsf{vstate}) = \mathsf{output}_{\mathcal{P}^*,\mathcal{V}_{\mathsf{int}}}\Big\langle \mathcal{P}^*(x_{pub}, x_{priv}), \mathcal{V}_{\mathsf{int}}(x_{pub}, x_{priv})\Big\rangle \quad ; \quad b_{\mathcal{V}} = \mathcal{V}_{\mathsf{check}}(\mathsf{vstate}, \mathsf{receipt}_{\mathcal{P}^*}) \quad ; \quad b_{\mathcal{A}} = \mathcal{A}(\mathsf{receipt}_{\mathcal{P}^*}) \quad : \quad b_{\mathcal{V}} = \mathsf{True} \quad \wedge \quad b_{\mathcal{A}} = \mathsf{True} \quad \Big] \leq \frac{1}{2}$$

   taken over the coin tosses of $\mathcal{P}^*$, $\mathcal{V}_{\mathsf{int}}$, and $\mathcal{V}_{\mathsf{check}}$.

3. **Private-Input Hiding.** *Even if the verifier is dishonest, the secret receipt does not yield the private input. Phrased differently, for every corrupt verifier that tries to prove the private input, there is another indistinguishable verifier who claims the same private input even though his private input was different.*

   Formally, there exists a negligible function $\nu(\cdot)$ such that, given security parameter κ, $\forall$ non-uniform PPT $\mathsf{Adv} = (\mathcal{G}^*, \mathcal{V}^*, \mathcal{C}^*)$, *$\exists$PPT $\mathcal{W}^*$ with the same computational abilities as $\mathcal{V}^*$, such that:*

$$
\begin{aligned}
Pr\Big[(x_{priv}, x'_{priv}) \xleftarrow{R} \mathcal{G}^*(\mathsf{Rel}_{\mathcal{L}}) \quad ; \\
(x_{pub}, w) \xleftarrow{R} \mathcal{S}_{\mathcal{L}}(x_{priv}) \quad ; \\
(x'_{pub}, w') \xleftarrow{R} \mathcal{S}_{\mathcal{L}}(x'_{priv}) \quad ; \\
b \xleftarrow{R} \{0,1\}; \\
(\mathsf{receipt}_b, \mathsf{vstate}_b) = \mathsf{output}_{\mathcal{P},\mathcal{V}^*}\Big\langle \mathcal{P}(x_{pub}, x_{priv}, w), \mathcal{V}^*(x_{pub}, x_{priv})\Big\rangle; \\
(\mathsf{receipt}_{\bar{b}}, \mathsf{vstate}_{\bar{b}}) = \mathsf{output}_{\mathcal{P},\mathcal{W}^*}\Big\langle \mathcal{P}(x'_{pub}, x'_{priv}, w'), \mathcal{W}^*(x'_{pub}, x'_{priv}, x_{priv})\Big\rangle; \\
b' = \mathcal{C}^*(\mathsf{receipt}_0, \mathsf{vstate}_0, \mathsf{receipt}_1, \mathsf{vstate}_1) \quad : \quad b = b'\Big] \leq \frac{1}{2} + \nu(\kappa)
\end{aligned}
$$

Note that $\mathcal{W}^*$ will use $\mathcal{V}^*$ as a black box in most constructions. Note also that, to prove that a protocol is realistically private-input hiding, one should not only prove the existence of $\mathcal{W}^*$, but also demonstrate a viable construction.

Private-input hiding *implies that, if a voter has the computational ability to remember the transcript of his interaction with the voting machine, then he can, with the same computational ability, produce an alternative transcript for the same receipt. This alternate transcript will be just as convincing as the real one.*

Note also how $\mathcal{G}^*$, the generator component of the Adversary, takes as input the circuit describing the language's relation. For example, if $\mathcal{L}$ is the language of proper ciphertext/plaintext pairs in a cryptosystem, $\mathcal{G}^*$ receives the cryptosystem's public key as input.

Remark 5-1 *Given the definition of an AHIP language and the fact that the Assistant is expected to learn x_{pub}, the above definition of an AHIP protocol inherently implies that the private-input hiding property is* computational*: we only consider* $\mathsf{Adv} \in PPT$.

Remark 5-2 *In this work, we do not address the scenario where a malicious prover colludes with a verifier willing to reveal his private input. It is unclear whether this can be done algorithmically: we may have to resort to code review and other such means of non-black verification for this task. We leave this issue to future work.*

5.4.5 Witness Hiding

A *private-input hiding* AHIP is also witness hiding. The reverse implication is not always true, however, which explains why our definition considers the private-input hiding property.

Theorem 5-1 *If an interactive protocol $(\mathcal{P}, (\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}}), \mathcal{A})$ is private-input hiding, then it is also witness hiding.*

Proof. We prove this by contradiction. Assume $(\mathcal{P}, \mathcal{V}_{\mathsf{int}})$ is not witness-hiding, then there exists an algorithm $\mathcal{V}^*$ which, after its *normal*, non-rewindable, interaction with $\mathcal{P}$, outputs w, a witness for (x_{pub}, x_{priv}) in the language $\mathcal{L}$, with some non-negligible probability. We can then build $\mathsf{Adv} = (\mathcal{G}^*, \mathcal{V}_2^*, \mathcal{C}^*)$ as follows:

1. $\mathcal{G}^*$ prepares two private inputs x_{priv} and x'_{priv}.
2. $\mathcal{V}_2^*$ receives (x_{pub}, x_{priv}) as input. It runs $\mathcal{V}^*(x_{pub}, x_{priv})$ as a subprocedure when interacting with $\mathcal{P}$. The protocol runs normally and produces a receipt as expected. In addition, because the protocol is not witness hiding by assumption, $\mathcal{V}^*$ returns w such that $\mathcal{R}_{\mathcal{L}}((x_{pub}, x_{priv}), w) = 1$. $\mathcal{V}_2^*$ then outputs $\mathsf{vstate}^* = (x_{pub}, x_{priv}, w)$.
3. $\mathcal{C}^*$ can verify that $(x_{pub}, x_{priv}) \in \mathcal{L}$ using the witness w and the language's relation $\mathcal{R}_{\mathcal{L}}$.
4. Assume there exists $\mathcal{W}^*$ that successfully produces (x'_{pub}, x_{priv}, w) that is successfully verified by $\mathcal{C}^*$. Then $\mathcal{R}_{\mathcal{L}}((x'_{pub}, x_{priv}), w) = 1$, which means that $(x'_{pub}, x_{priv}) \in \mathcal{L}$.
5. We reach a contradiction based on the definition of an AHIP language: if $(x_{pub}, x_{priv}) \in \mathcal{L}$, then there can be no other x'_{priv} such that $(x_{pub}, x'_{priv}) \in \mathcal{L}$.

Thus, an AHIP that exhibits private-input hiding is also witness hiding. □

5.4.6 A General Construction

We now describe the techniques common to both of our protocols, BitProof and BetterBitProof. We give the details of BitProof and BetterBitProof in Sections 5.5 and Section 5.6, respectively.

Why Not Existing ZK Proofs? Recall that typical proof-of-plaintext zero-knowledge proofs [37, 88] are not AHIPs: a human verifier cannot perform the computation required to verify a typical zero-knowledge proof. Simply giving the assistant the entire proof transcript is no solution either: the private input leaks immediately.

One might consider a construction from generic components, where the Prover performs the proof correctly for the real private input j, then simulates the other $z-1$ proofs for private inputs $j^* \neq j$. With all transcripts printed on the receipt, an Assistant would be unable to tell which was truly executed sequentially: only the Verifier would know from her private interaction with the Prover. However, this is also not human-verifiable, for more subtle reasons: the messages of typical proof protocols are far too long for humans to remember and compare (hundreds of bits). Even if we artificially reduce the verifier challenge space, the simulation of the $z-1$ incorrect values of the private input will result in normal-sized prover messages with overwhelming probability.

Interestingly, this "prove-then-simulate" method would be workable if the specific proof protocols were more amenable to human verification. In fact, this is exactly our approach: we define a special method for encrypting bits that lends itself particularly well to human verification. The soundness of our protocols is optimal in the length of the verifier's challenge: a 24-bit challenge (4 ASCII characters) yields soundness of 2^{-24}.

Protocol Setup. The goal of our AHIP proofs is to demonstrate that a list of ciphertexts encodes an integer j between 1 and z. In particular, given z bit-encryptions $c_1, c_2, \ldots, c_z$, where $c_i = \mathcal{E}_{pk}(b_i)$ and $b_i = 1$ for $i = j'$ while $b_i = 0$ otherwise, each of our two protocols demonstrates that $j' = j$. Thus, the private common input is $x_{priv} = j$, the public common input is $x_{pub} = (c_1, \ldots, c_z)$, and the prover's secret input is $(r_1, \ldots, r_z)$, the randomization values used in generating the public common input ciphertexts.

Special Bit Encryptions. Each construction provides a special form of bit encryption: BitEnc for BitProof, and BetterBitEnc for BetterBitProof. These encryption forms lend themselves to proof protocols with optimally-sized first prover message and verifier challenge, given a desired soundness parameter. For clarity, we use the single notation BitEnc in this generic description.

In both schemes, BitEnc uses the same public key pk as the cryptosystem used to encrypt the input ciphertexts, though the resulting ciphertext form is different. The proofs proceed as follows on public common input $x_{pub} = (c_1, \ldots, c_z)$, private common input $x_{priv} = j$, and secret prover input $w = (r_1, \ldots, r_z)$, such that $\forall i \in [1, z] \neq j, b_i = 0$, $b_j = 1$, and $\forall i \in [1, z], c_i = \mathcal{E}_{pk}(b_i; r_i)$:

1. Prover selects $(R_1, R_2, \ldots, R_z)$ in the proper randomization space of BitEnc, and generates $(\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z)$ such that $\forall i \in [1, z], \mathbb{c}_i = \mathsf{BitEnc}_{pk}(b_i; R_i)$.
2. Prover and Verifier engage in a special proof protocol that $\mathbb{c}_i = \mathsf{BitEnc}_{pk}(0)$ for $i \neq j$, and $\mathbb{c}_j = \mathsf{BitEnc}_{pk}(1)$.
3. Prover and Verifier engage in a proof that $\mathbb{c}_i$ and c_i encrypt the same bit (in their respective cryptosystems) for all i. It is worth noting that this proof does not require any verification by the human verifier. Because the semantic security of both cryptosystems hides the value of j, this portion of the proof can be entirely checked by the Assistant.

We now describe how our two constructions, BitProof and BetterBitProof, implement assisted-human interactive proofs for BitEnc and BetterBitEnc, respectively.

5.5 The BitProof Protocol

We begin with the BitProof protocol, which is a slightly modified version of Neff's [29] original voter-intent verification scheme. In Neff's original scheme, the special bit-encryptions are used as the primary representation of the ballot, i.e. as the proof input. In that setting, it is not possible to prove the private-input hiding property, because the bit ciphertexts are *fixed* before the proof begins. Thus, it is not possible to rewind the protocol far enough to change the special bit encryptions without also changing the verifier's challenge. This technical complication prevents the proper reduction to semantic security.

Our modification is simple: the inputs are normal ciphertexts, and the special bit encryptions are generated *during* the proof protocol. This requires us to prove an additional step: that the bit encryptions match the input ciphertexts. However, as we will see in the proof at the end of this section, BitProof is then naturally private-input hiding, assuming El Gamal is semantically secure.

We now present BitEnc, the special form of bit encryption, and BitProof, the AHIP protocol that proves a correct ciphertext/plaintext pair under BitEnc, with the plaintext as the private input that remains hidden from the Assistant.

5.5.1 Bit Encryption

The structure of the special ciphertexts $\mathbb{c}_i = \mathsf{BitEnc}(b_i)$ is such that $\mathcal{P}$ can split up the proof that $\mathbb{c}_j = \mathsf{BitEnc}(1)$ into a human-verifiable component and a machine-verifiable component, such that the machine does not learn j. Specifically, given a bit $b \in \{0,1\}$ and an El Gamal public key pk, BitEnc produces:

$$\mathsf{BitEnc}_{pk}(b; R) = [u_1, v_1], [u_2, v_2], \ldots, [u_\alpha, v_\alpha]$$

such that :

- $R = ((s_1, \ldots, s_\alpha), (t_1, \ldots, t_\alpha), (a_1, \ldots, a_\alpha))$, with:
  - $s_1, \ldots, s_\alpha$ randomization values for pk,
  - $t_1, \ldots, t_\alpha$ randomization values for pk, and
  - $a_1, \ldots, a_\alpha$ single bits.
- $\forall k \in [1, \alpha]$:
  - $u_k = \mathcal{E}_{pk}(a_k; s_k)$
  - if $b = 1$, $v_k = \mathcal{E}_{pk}(a_k; t_k)$
  - if $b = 0$, $v_k = \mathcal{E}_{pk}(\bar{a_k}; t_k)$.

In other words, all the u_k and v_k values are encryptions of either 0 or 1, using $\mathcal{E}_{pk}$. If $b = 1$, both elements of each pair encrypt *the same bit*: $\mathcal{D}(u_i, v_i) = (1,1)$ or $(0,0)$. If $b = 0$, both elements of each pair encrypt *opposite bits*: $\mathcal{D}(u_i, v_i) = (1,0)$ or $(0,1)$. Note that, given a bit b, there are two plaintext choices for each pair within $\mathsf{BitEnc}(b)$. All pairs are not the same: the exact selection of bits is part of the randomization value provided to each invocation of BitEnc.

5.5.2 Protocol

The public common input to the proof is $x_{pub} = (c_1, c_2, \ldots, c_z)$, with $c_i = \mathcal{E}_{pk}(b_i)$, while the private common input is $x_{priv} = j$. Recall that we already assume that $(c_1, c_2, \ldots, c_z)$ is a correct ciphertext set for *some index* j'. The goal of BitProof is to prove that $j = j'$. (This assumption is explained in Section 5.7.)

Communication Model. Recall that the Prover may communicate in two ways: by sending messages directly to the Verifier, or by printing message on the receipt. The messages sent directly to the Verifier are private: they do not appear on the receipt. The messages sent to the receipt are visible to both the Verifier and, later, the Assistant. The receipt has the extra property that the Prover effectively commits to the information, since we assume that even partially printed information on the receipt cannot be modified (Section 5.3).

Inputs. The public common input to the proof, x_{pub}, is printed on the receipt at the beginning of the interaction. Given the physical properties of the receipt, this is effectively a commitment to x_{pub}. The private common input, x_{priv} is given secretly to both the Prover and Verifier.

Protocol BitProof:
Consider the AHIP setup, with prover $\mathcal{P}$, limited verifier $\mathcal{V} = (\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}})$, and assistant $\mathcal{A}$. Let α be an integer parameter which affects the soundness of the proof. The proof inputs are:

- public common input $x_{pub} = (pk, c_1, \ldots, c_z)$, where pk is an El Gamal public key, and

$$\exists j' \in [1, z], c_{j'} = \mathcal{E}_{pk}(1) \text{ and } \forall i \neq j', c_i = \mathcal{E}_{pk}(0)$$

  Denote b_i such that $c_i = \mathcal{E}_{pk}(b_i)$. (Thus, $b_{j'} = 1$, and $b_i = 0$ otherwise.)

- private common input $x_{priv} = j \in [1, z]$.
- secret prover input $(r_1, \ldots, r_z)$ such that, $\forall i \in [1, z], c_i = \mathcal{E}_{pk}(b_i; r_i)$

Thus, the proof protocol is denoted:

$$\Big\langle \mathcal{P}(pk, c_1, c_2, \ldots, c_z, j, r_1, r_2, \ldots, r_z), \mathcal{V}_{\mathsf{int}}(pk, c_1, c_2, \ldots, c_z, j) \Big\rangle$$

and it proceeds as follows:

1. $\mathcal{V}_{\mathsf{int}}$ sends $\mathit{commit}(\mathsf{chal}_{\mathcal{V}})$ to $\mathcal{P}$, where $\mathsf{chal}_{\mathcal{V}}$ is an α-bit string.
2. $\mathcal{P}$ prepares:
   - $(R_1, \ldots, R_z)$, a list of randomization values for BitEnc_{pk}.

- $(\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z)$, such that $\mathbb{c}_i = \mathsf{BitEnc}_{pk}(b_i; R_i)$.
- $\mathsf{ChosenString}$, an α-bit string which concatenates the underlying bit choices $(a_1, \ldots, a_\alpha)$ within R_j, the randomization value for $\mathbb{c}_j$:

$$\forall k \in [1, \alpha], \mathsf{Dec}(u_k^{(j)}) = \mathsf{Dec}(v_k^{(j)}) = \mathsf{ChosenString}[i].$$

$\mathcal{P}$ prints on $\mathsf{receipt}$ $(\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z)$. $\mathcal{P}$ sends $\mathsf{ChosenString}$ to $\mathcal{V}_{\mathsf{int}}$, but *not to the receipt.* We assume that $\mathcal{V}_{\mathsf{int}}$ can remember $\mathsf{ChosenString}$.

3. $\mathcal{V}_{\mathsf{int}}$ sends $\mathsf{chal}_{\mathcal{V}}$ to Prover, including proof that this is consistent with the commitment sent earlier. (If the proof fails, $\mathcal{P}$ aborts.) This proof is a typical non-interactive proof of decommitment that simply unveils $\mathsf{chal}_{\mathcal{V}}$ and the commitment randomness. $\mathcal{V}_{\mathsf{int}}$ is not expected to actually compute this commitment: as described in Section 5.3, a helper organization has pre-computed $\mathsf{chal}_{\mathcal{V}}$ and $\mathit{commit}(\mathsf{chal}_{\mathcal{V}})$ for the Verifier.

4. $\mathcal{P}$ sends α-bit strings $\mathsf{String}^{(1)}, \ldots, \mathsf{String}^{(z)}$ and α-length arrays $\mathsf{RandFactors}^{(1)}, \ldots, \mathsf{RandFactors}^{(z)}$ to the receipt, where, $\forall k \in [1, \alpha]$:

   - if $\mathsf{chal}_{\mathcal{V}}[k] = 0$, $\mathsf{String}^{(i)}$ and $\mathsf{RandFactors}^{(i)}$ reveal the plaintexts of the u's:

$$\forall i \in [1, z], \quad u_k^{(i)} = \mathcal{E}_{pk}\left(\mathsf{String}^{(i)}[k], \mathsf{RandFactors}_k^{(i)}\right)$$

   - if $\mathsf{chal}_{\mathcal{V}}[k] = 1$, $\mathsf{String}^{(i)}$ and $\mathsf{RandFactors}^{(i)}$ reveal the plaintexts of the v's:

$$\forall i \in [1, z], \quad v_k^{(i)} = \mathcal{E}_{pk}\left(\mathsf{String}^{(i)}[k], \mathsf{RandFactors}_k^{(i)}\right)$$

   Note how $\mathsf{String}^{(i)}$ and $\mathsf{RandFactors}^{(i)}$ are parts of the randomization values in R_i, initially generated randomly by the Prover when creating the bit encryption $\mathbb{c}_i$.

5. Recall that $s_k^{(i)}$ denotes the randomization value used to encrypt $u_k^{(i)}$, and $t_k^{(i)}$ denotes the randomization value used to encrypt $v_k^{(i)}$. Both of these are parts of R_i, the randomization value for $\mathbb{c}_i$. For each pair $[u_k^{(i)}, v_k^{(i)}]$, $\mathcal{P}$ performs the following:

   - if $\mathsf{chal}[k] = 1$, define $\beta_k^{(i)} = s_k^{(i)}$, the randomization value of $u_k^{(i)}$, and define $b_k^{(i)}$ as the bit that $v_k^{(i)}$ decrypts to.
   - if $\mathsf{chal}[k] = 0$, define $\beta_k^{(i)} = t_k^{(i)}$, the randomization value of $v_k^{(i)}$, and define $b_k^{(i)}$ as the bit that $u_k^{(i)}$ decrypts to.
   - if $b_k^{(i)} = 0$, compute

$$\rho_k^{(i)} = r_i + \beta_k^{(i)}$$

- if $b_k^{(i)} = 1$, compute

$$\rho_k^{(i)} = r_i - \beta_k^{(i)}$$

Note how $b_k^{(i)}$ corresponds to the "revealed bit" for pair k of bit encryption i, while $\beta_k^{(i)}$ corresponds to the randomization value for the *non-revealed* element within the same pair.

6. $\mathcal{P}$ outputs

$$\begin{aligned}\mathsf{receipt} = \Big(c_1, \ldots, c_z, \mathbb{c}_1, \ldots, \mathbb{c}_z, \mathsf{chal}_{\mathcal{P}},\\ \mathsf{String}^{(1)}, \ldots, \mathsf{String}^{(z)},\\ \mathsf{RandFactors}^{(1)}, \ldots, \mathsf{RandFactors}^{(z)},\\ \{\rho_k^{(1)}\}_{k\in[1,\alpha]}, \ldots, \{\rho_k^{(z)}\}_{k\in[1,\alpha]}\Big)\end{aligned}$$

where $\mathsf{chal}_{\mathcal{P}} = \mathsf{chal}_{\mathcal{V}}$. We use different notation here to point out that a dishonest prover might print a different challenge on the receipt.

7. $\mathcal{V}_{\mathsf{int}}$ outputs $\mathsf{vstate} = (j, \mathsf{chal}_{\mathcal{V}}, \mathsf{ChosenString})$ for $\mathcal{V}_{\mathsf{check}}$ to consider.

8. $\mathcal{V}_{\mathsf{check}}$ checks $\mathsf{chal}_{\mathcal{V}} = \mathsf{chal}_{\mathcal{P}}$ and $\mathsf{String}^{(j)} = \mathsf{ChosenString}$. $\mathcal{V}_{\mathsf{check}}$ outputs True/False accordingly.

9. On input $\mathsf{receipt}$, $\mathcal{A}$ checks:

   (a) the opening up of u's and v's is consistent with $\mathsf{chal}, \mathsf{String}^{(1)}, \ldots, \mathsf{String}^{(m)}$.

   (b) the revealed $u_k^{(i)}$'s and $v_k^{(i)}$'s are correct encryptions of the bits of $\mathsf{String}(i)$ using randomization values $\mathsf{RandFactors}_k^{(i)}$, $\forall k \in [1, \alpha], \forall i \in [1, z]$.

   (c) for each pair $[u_k^{(i)}, v_k^{(i)}]$:

   - if $\mathsf{chal}_{\mathcal{V}}[k] = 1$, call $b_k^{(i)}$ the revealed bit of $v_k^{(i)}$ and $w_k^{(i)} = u_k^{(i)}$.
   - if $\mathsf{chal}_{\mathcal{V}}[k] = 0$, call $b_k^{(i)}$ the revealed bit of $u_k^{(i)}$ and $w_k^{(i)} = v_k^{(i)}$.
   - note how $b_k^{(i)}$ is the revealed bit for pair k of bit encryption i, and $w_k^{(i)}$ is the unopened ciphertext in the same pair.
   - if $b_k^{(i)} = 0$, then check:

$$c_i w_k^{(i)} \stackrel{?}{=} \mathcal{E}_{pk}(1; \rho_k^{(i)})$$

   - if $b_k^{(i)} = 1$, then check:

$$\frac{c_i}{w_k^{(i)}} \stackrel{?}{=} \mathcal{E}_{pk}(0; \rho_k^{(i)})$$

Note that this verification shows that $\mathbb{c}_i$ and c_i encode the same bit. The homomorphic computation yields an encryption of the opposite of the already-revealed bit, if and only if $\mathbb{c}_i$ and c_i encrypt the same bit, assuming that they do encrypt bits to begin with. The revelation of $\rho_k^{(i)}$ enables this check.

If all succeed, $\mathcal{A}$ outputs True, otherwise False.

5.5.3 Proof of AHIP

We now prove that the above protocol is a private-input hiding AHIP. Intuitively, to prove the private-input hiding property, we simulate the environment for the Verifier willing to reveal his input to the Assistant, plugging in alternate ciphertexts for the pairs within the bit encryptions. A distinguisher succeeds in distinguishing the selling Verifier from our construction only if it can break the semantic security of El Gamal. We must assume, of course, that the simulator has the same computational ability as the verifier: if the simulator requires much greater computational ability, then the scheme is clearly coercible based on the inability to simulate. At the same time, it is safe to assume a minimal computational ability for the verifier, i.e. that she can compare short strings and remember one short string: we don't need to simulate the protocol for a completely incapable verifier who cannot perform protocol verification in the first place.

Theorem 5-2 BitProof *is a private-input hiding AHIP protocol.*

Proof. Note, again, that we assume that $(c_1, c_2, \dots, c_z)$ is well-formed, meaning that exactly one ciphertext is the encryption of bit 1, and all others are encryptions of bit 0. The proof is meant to demonstrate that j is indeed the index of the ciphertext that is an encryption of 1.

1. **Completeness.** The honest construction of $\mathbb{c}_j = \mathsf{BitEnc}_{pk}(1)$ gives a sequence of α pairs of ciphertexts where both elements of any given pair encode the same bit. Thus, no matter the value of chal, $\mathsf{String}^{(j)}$ will be fixed for a given $\mathbb{c}_j$, in particular it will be fixed to $\mathcal{P}$'s ChosenString.

   In addition, for all indexes $i \in [1, z]$ and $k \in [1, \alpha]$, if the pair $[u_k^{(i)}, v_k^{(i)}]$ is correctly created, then the homomorphic computation in Step 9c will succeed and produce a ciphertext (of 0 or 1, depending on the case) with the specified randomness $\rho_k^{(i)}$, in which case $\mathcal{A}$ will declare success on this check. Every other check of $\mathcal{A}$ and $\mathcal{V}_{\mathsf{check}}$ simply verifies that $\mathcal{P}$ performed the correct math in revealing the randomness of various ciphertexts. Thus, with probability 1, $\mathcal{V}_{\mathsf{check}}$ and $\mathcal{A}$ output True if $\mathcal{P}$ is honest.

2. **Soundness.** If the prover is cheating, we denote him $\mathcal{P}^*$. As we continue to assume well-formed ciphertext sets $c_1, c_2, \dots, c_z$, and we know that the prover is trying to cheat, $c_j = \mathcal{E}_{pk}(0)$. $\mathcal{P}^*$ now has 2 possible strategies:

(a) $\mathcal{P}^*$ might generate $\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z$ properly, meaning that each $\mathbb{c}_i$ encodes the same plaintext as the corresponding c_i. Clearly, the equivalence checks between $\mathbb{c}_i$ and c_i will now succeed. However, since $c_j = \mathcal{E}_{pk}(0)$, then $\mathbb{c}_j = \mathsf{BitEnc}_{pk}(0)$. Thus, $\mathbb{c}_j$ must be composed of pairs of ciphertexts encoding opposite bits.

Recall that $\mathcal{P}^*$ sends $\mathsf{ChosenString}$ before seeing chal. With $\mathbb{c}_j$ composed of pairs of ciphertexts encoding opposite bits, in particular given that the values of $u_k^{(j)}$ are randomly selected for each k, a given $\mathbb{c}_j$ defines a one-to-one mapping from chal to $\mathsf{String}^{(j)}$. The only way for $\mathcal{P}^*$ to succeed is to encrypt $\mathbb{c}_j = \mathsf{BitEnc}(0)$ with precisely the unique choice of "0,1" and "1,0" pairs such that the verifier's chal turns out to be the single value that maps to $\mathsf{ChosenString}$, so that $\mathsf{String}^{(j)} = \mathsf{ChosenString}$. Assuming a statistically hiding commitment scheme for $\mathit{commit}(\mathsf{chal})$, $\mathcal{P}^*$ can only rely on chance. As the length of chal is α, the probability of successfully tricking $\mathcal{V}_{\mathsf{check}}$ is thus $2^{-\alpha}$. Even with small α, soundness is quite high.

There remains the possibility that $\mathcal{P}^*$ will try to trick $\mathcal{A}$ by attempting to reveal the bits of $\mathsf{String}^{(j)}$ when the pairs of ciphertexts do not actually encrypt those desired bits. Any such attempt by $\mathcal{P}^*$ will be detected by $\mathcal{A}$, who can simply re-perform the $\mathcal{E}$ operations given $\mathsf{String}^{(j)}$ and $\mathsf{RandFactors}^{(j)}$. Thus, it is impossible for $\mathcal{P}$ to successfully provide incorrect values of $\mathsf{String}^{(j)}$ and $\mathsf{RandFactors}^{(j)}$.

(b) $\mathcal{P}^*$ might generate $\mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z$ in some way that, for some i, $\mathbb{c}_i$ and c_i don't encode the same plaintext. Thus, there is at least one index k such that the pair $[u_k^{(i)}, v_k^{(i)}]$ is not a proper pair according to c_i. $\mathcal{P}^*$ will have to reveal either $u_k^{(i)}$ or $v_k^{(i)}$ after $\mathcal{V}_{\mathsf{int}}$ issues chal. If the revealed value is not 0 or 1, $\mathcal{V}_{\mathsf{check}}$ or $\mathcal{A}$ will notice with certainty. If the revealed value is 0 or 1, then the reveal of $\rho_k^{(i)}$ will allow $\mathcal{A}$ to notice the error in the homomorphic check of Step 9c.

3. **Private-Input Hiding.** We show that, if an adversary $\mathsf{Adv} = (\mathcal{G}^*, \mathcal{V}^*, \mathcal{C}^*)$ can distinguish its verifier $\mathcal{V}^*$ from *any* simulator $\mathcal{W}^*$, it can also break semantic security of the underlying public-key cryptosystem $(\mathcal{G}, \mathcal{E}, \mathcal{D})$, in this case El Gamal. Recall that El Gamal and Exponential El Gamal have equivalent semantic security: one cannot break without the other breaking, too. We show here the construction of Adv', a new adversary based on Adv that breaks the semantic security of Exponential El Gamal. Recall that Exponential El Gamal provides an additive homomorphic property, such that ciphertexts for two fixed values can be toggled homomorphically from one to the other. In particular, the homomorphic operation $\mathcal{E}(1)/c$ toggles ciphertext c from $\mathcal{E}(0)$ to $\mathcal{E}(1)$ and back.

First, we construct $\mathcal{W}^*$, the verifier that, with the same computational ability as $\mathcal{V}^*$, should be able to simulate any of its outputs. $\mathcal{W}^*$ will basically play man-in-the-middle between $\mathcal{P}$ and $\mathcal{V}^*$, substituting the private input along the way (effectively option j) and rewinding $\mathcal{V}^*$ as necessary to successfully complete the man-in-the-middle simulation. $\mathcal{P}$ will be creating a receipt for option j', while $\mathcal{W}^*$ substitutes certain messages to convince $\mathcal{V}^*$ that the chosen option was j. $\mathcal{W}^*$ outputs whatever $\mathcal{V}^*$ outputs.

We then show that if $\mathcal{C}^*$ can distinguish the outputs of $\mathcal{W}^*$ and $\mathcal{V}^*$, then we can build Adv' to break semantic security of the underlying cryptosystem.

$\mathcal{W}^*(pk, (c'_1, \dots, c'_z), j', j)$:

(a) initialize $\mathcal{V}^*$ with the appropriate random tape and inputs $(pk, c'_1, \dots, c'_z, j)$. Note that $(c'_1, \dots, c'_z)$ encodes option j', not j. $\mathcal{W}^*$ will now trick $\mathcal{V}^*$ into accepting this receipt for option j.

(b) run $\mathcal{V}^*$ with random, well-formed inputs until it reveals chal. Rewind $\mathcal{V}^*$ until right after it submitted $\mathit{commit}(\mathsf{chal})$ (so it is still committed to this challenge.)

(c) interact with $\mathcal{P}$ normally, giving it the same $\mathit{commit}(\mathsf{chal})$ and chal as $\mathcal{V}^*$ submitted.

   Collect $\mathsf{String}^{(i)}, \mathsf{RandFactors}^{(i)}$ for all $i \in [1, z]$, noting in particular $\mathsf{String}^{(j)}$. Perform all soundness checks against the messages from $\mathcal{P}$, to ensure that $(c'_1, \dots, c'_z)$ indeed encodes j'.

   Recall that, by assumption, we do not consider the case where $\mathcal{P}$ is colluding with the adversary. Thus, in this construction, we assume $\mathcal{P}$ successfully completes the proof.

(d) run $\mathcal{V}^*$, simulating to it all messages it expects from the Prover. In particular, send the just-discovered $\mathsf{String}^{(j)}$ to $\mathcal{V}^*$ when it expects $\mathsf{ChosenString}$. Every subsequent aspect can be simulated, since the chal submitted by $\mathcal{V}^*$ will be the same as was originally extracted. The only difference between the $\mathcal{P}$-to-$\mathcal{W}^*$ messages and $\mathcal{W}^*$-to-$\mathcal{V}^*$ messages is the replacement of $\mathsf{ChosenString}$.

(e) output whatever $\mathcal{V}^*$ outputs.

We stress that, in the above setup, $\mathcal{V}^*$ saw *exactly what it would have seen* in a real interaction to convince it that $\mathsf{receipt}$ encodes choice j, yet $\mathsf{receipt}$ actually encodes choice j'. In addition, $\mathcal{P}$ interfacing with $\mathcal{W}^*$ also saw exactly what it would have seen while interacting with $\mathcal{V}^*$, so $\mathsf{receipt}$ will match what $\mathcal{V}^*$ outputs.

Now we show how, if $\mathcal{C}^*$ can distinguish which of the receipt/output pairs belongs to $\mathcal{V}^*$ and which one belongs to $\mathcal{W}^*$, then we break semantic security of the underlying cryptographic scheme. Our Adv' construction effectively runs the programs for $\mathcal{W}^*$ and $\mathsf{Adv} = (\mathcal{G}^*, \mathcal{C}^*, \mathcal{V}^*)$ as black boxes, simulating to these boxes the actions of $\mathcal{P}$. Let α and z be integer system parameters that will affect the probability of success of Adv'.

Before we construct Adv', we construct $\mathcal{X}$, a special extractor that interacts with $\mathcal{W}^*$ as if it were the prover. We modularize this construction first, because, in the construction of Adv', we will use *two* instances of $\mathcal{W}^*$, and thus two instances of $\mathcal{X}$, one to "run" each $\mathcal{W}^*$. This approach is necessary because, in this reduction, we cannot be certain to provide $\mathcal{V}^*$ with truly correct inputs. Thus, we run two $\mathcal{W}^*$ instances, one which, through a "double swap," will turn out to produce exactly the output *and* receipt that would have been produced by $\mathcal{V}^*$ in a real interaction with the Prover, and another that will behave like a normal $\mathcal{W}^*$, outputting a receipt that doesn't actually match its claim of private input. The construction won't know which one is which, as they will be constructed from the challenge ciphertext provided by the semantic security game. We will then ask $\mathcal{C}^*$ to distinguish them. If it can, our adversary Adv' will break semantic security.

The purpose of $\mathcal{X}$ is to run an instance of $\mathcal{W}^*$ with inputs that are derived from a single ciphertext that $\mathcal{X}$ cannot decrypt, for instance the ciphertext received as the challenge from the semantic security environment (or a related one). The extractor will thus rewind and run $\mathcal{W}^*$ in such a way as to ensure that $\mathcal{W}^*$ accepts the proof and outputs its normal output, even if its input is incorrect.

This may be confusing, as one may recall that $\mathcal{W}^*$ is doing exactly the same thing to the underlying $\mathcal{V}^*$: tricking it into accepting a proof. In fact, $\mathcal{X}$ performs an interesting twist of hand: if the input provided to $\mathcal{W}^*$ is corrupt, then $\mathcal{W}^*$'s behavior reverts to *exactly* the behavior of $\mathcal{V}^*$, with a receipt to match. Of course, neither $\mathcal{W}^*$ nor $\mathcal{X}$ will know that this reversal has occurred. If $\mathcal{C}^*$ figures it out, then it will have broken the semantic security of the underlying cryptosystem.

At a high level, $\mathcal{X}$ behaves like a Prover with one hand tied behind its back: it must use some ciphertext it hasn't created as part of the bit encryptions. To enable it to answer $\mathcal{W}^*$'s challenge, $\mathcal{X}$ will need to rewind $\mathcal{W}^*$ in the usual way.

$\mathcal{X}(pk, z, \alpha, C, j_0, j_1)$:

(a) $\mathcal{X}$ assumes that C is the encryption of a bit b under $\mathcal{E}_{pk}$. Note that $\mathcal{X}$ can compute C^{-1}, the homomorphic inversion of C into an encryption of $\bar{b}$. C^{-1} is an El Gamal pair whose elements are computed as the inverse of the elements of C. All of this can be done *without* knowing b. The integers j_0 and j_1 are in the range $[1, z]$.

(b) $\mathcal{X}$ produces $(c_1, \dots, c_z)$, as follows:

- select El Gamal randomization values $(r_1, \dots, r_z)$
- compute $c_{j_0} = \mathcal{RE}_{pk}(C^{-1}; r_{j_0})$
- compute $c_{j_1} = \mathcal{RE}_{pk}(C; r_{j_1})$
- compute $\forall i \in [1, z], i \neq j_0$ and $i \neq j_1$, $c_i = \mathcal{E}_{pk}(0; r_i)$

Note how, without knowing the value of b, $\mathcal{X}$ has ensured that $c_{j_0} = \mathcal{E}_{pk}(\bar{b})$ and $c_{j_1} = \mathcal{E}_{pk}(b)$. Thus $(c_1, \dots, c_z)$ corresponds to a genuine public common input x_{pub} for private input $x_{priv} = j_b$.

(c) $\mathcal{X}$ initializes $\mathcal{W}^*(pk, (c_1, \dots, c_z), j_0, j_1)$. Note how, at this point, $\mathcal{W}^*$ is being run on correct indices j_0 and j_1 if $b = 0$, and swapped (thus incorrect) indices if $b = 1$.

(d) $\mathcal{X}$ simulates valid, correctly distributed Prover messages while running $\mathcal{W}^*$ until $\mathcal{W}^*$ reveals chal, which happens before $\mathcal{X}$ needs to prove anything real. Then, $\mathcal{X}$ records chal and rewinds $\mathcal{W}^*$ until just after it sends $commit(\mathsf{chal})$.

(e) $\mathcal{X}$ can now prepare the bit encryptions with advance knowledge of chal. For $i \in [1, z]$, $\mathcal{X}$ prepares $\mathbb{c}_i$ as follows:

- if $i \neq j_0$ and $i \neq j_1$, $\mathbb{c}_i = \mathsf{BitEnc}_{pk}(0; R_i)$, with R_i picked randomly.
- if $i = j_0$, then, for $k \in [1, \alpha]$:
  - if $\mathsf{chal}[k] = 0$, $\mathcal{X}$ will have to reveal $u_k^{(i)}$. Thus, it picks random bit $a_k^{(i)}$, El Gamal randomization values $s_k^{(i)}$ and $t_k^{(i)}$, and computes

    $$u_k^{(i)} = \mathcal{E}_{pk}(a_k^{(i)}; s_k^{(i)}).$$

    If $a_k^{(i)} = 1$, $\mathcal{X}$ then computes

    $$v_k^{(i)} = \mathcal{RE}_{pk}(C^{-1}; t_k^{(i)}).$$

    Otherwise, if $a_k^{(i)} = 0$, it computes

    $$v_k^{(i)} = \mathcal{RE}_{pk}(C, t_k^{(i)}).$$

    Recall that C is the encryption of b and C^{-1} is the encryption of $\bar{b}$. Thus, if $b = 0$, then $u_k^{(i)}$ and $v_k^{(i)}$ encrypt the same bit, but if $b = 1$, then $u_k^{(i)}$ and $v_k^{(i)}$ encrypt different bits.

  - if $\mathsf{chal}[k] = 1$, $\mathcal{X}$ will have to reveal $v_k^{(i)}$. Thus, $\mathcal{X}$ performs the opposite assignments: the bit equality between the $u_k^{(i)}$ and $v_k^{(i)}$ remains the same, but $\mathcal{X}$ must be able to reveal the other element. It picks random bit $a_k^{(i)}$, randomization values $s_k^{(i)}$ and $t_k^{(i)}$, and computes

    $$v_k^{(i)} = \mathcal{E}_{pk}(a_k^{(i)}; s_k^{(i)}).$$

    Then, if $a_k^{(i)} = 1$, $\mathcal{X}$ computes

    $$u_k^{(i)} = \mathcal{RE}_{pk}(C^{-1}; t_k^{(i)}).$$

    Otherwise, it computes

    $$u_k^{(i)} = \mathcal{RE}_{pk}(C, t_k^{(i)}).$$

    Thus, again, if $b = 0$, $u_k^{(i)}$ and $v_k^{(i)}$ encrypt the same bit, but if $b = 1$, they encrypt different bits.

  $\mathcal{X}$ thus produces $c_{j_0} = \left([u_1^{(j_0)}, v_1^{(j_0)}], \ldots, [u_\alpha^{(j_0)}, v_\alpha^{(j_0)}]\right)$ with the values just generated. Note how $\mathbb{c}_{j_0} = \mathsf{BitEnc}_{pk}(\bar{b})$, because the elements of each pair encode the same bit when $b = 0$, and different bits if $b = 1$.
- if $i = j_1$, then $\mathcal{X}$ performs a similar set of actions as for $i = j_0$, with one key difference: it inverts its use of C and C^{-1} in the special preparation of the

pairs within $\mathbb{c}_{j_1}$. Thus, the elements within each pair of $\mathbb{c}_{j_1}$ encrypt the same bit if $b = 1$, and different bits if $b = 0$. Note how $\mathbb{c}_{j_1} = \mathsf{BitEnc}_{pk}(b)$.

As a result, $\mathcal{X}$ has created a proper list $(\mathbb{c}_1, \ldots, \mathbb{c}_z)$ for input j_b: when $b = 0$, $\mathbb{c}_{j_0}$ encrypts 1, and $\mathbb{c}_{j_1}$ encrypts 0, and when $b = 1$, the roles are reversed.

In other words, if $b = 0$, $\mathcal{X}$ has provided completely correct inputs to $\mathcal{W}^*$, and if $b = 1$, $\mathcal{X}$ has flipped the indices, and $\mathcal{W}^*$ is being "cheated."

(f) Regardless, $\mathcal{X}$ can continue the simulation for $\mathcal{W}^*$, providing all necessary $\mathsf{String}^{(i)}$, since the pairs within the bit encryptions were set up so that $\mathcal{X}$ knows how to decrypt the right elements within each pair.

(g) Though it may have lied to $\mathcal{W}^*$ regarding whether $(c_1, \ldots, c_z)$ really does encode option j_0, $\mathcal{X}$ did *not* lie regarding the fact that, for all i, $\mathbb{c}_i$ and c_i encrypt the same bit: $\mathbb{c}_{j_0}$ and c_{j_0} both encrypt bit $\bar{b}$, whatever b is, and $\mathbb{c}_{j_1}$ and c_{j_1} both encrypt bit b. That said, correctness doesn't necessarily imply that $\mathcal{X}$ can reveal the $\rho_k^{(i)}$ properly.

Here, we show that $\mathcal{X}$ can in fact reveal $\rho_k^{(i)}$:

- for $i \neq j_0$ and $i \neq j_1$, $\mathcal{X}$ created the c_i and $\mathbb{c}_i$ from "scratch". Thus, it can easily reveal the appropriate $\rho_i^{(k)}$, since it knows all randomization values for all ciphertexts, just like an honest Prover.
- for $i = j_0$, and for $k \in [1, \alpha]$:
  - note how $c_{j_0} = \mathcal{RE}_{pk}(C^{-1}; r_{j_0})$
  - note how, if the revealed $a_k^{(j_0)} = 1$, then the other ciphertext in the pair, $w_k^{(j_0)}$ (which is $u_k^{(j_0)}$ if $\mathsf{chal}[k] = 1$ or $v_k^{(j_0)}$ otherwise), is $\mathcal{RE}_{pk}(C^{-1}; t_k^{(j_0)})$, and recall that the Verifier will check:

$$\frac{c_i}{w_k^{(i)}} \stackrel{?}{=} \mathcal{E}_{pk}(0; \rho_k^{(i)})$$

    Conveniently:

$$\begin{aligned} \frac{c_{j_0}}{w_k^{(j_0)}} &= \frac{\mathcal{RE}_{pk}(C^{-1}; r_{j_0})}{\mathcal{RE}_{pk}(C^{-1}; t_k^{(j_0)})} \\ &= \mathcal{E}_{pk}(0; r_{j_0} - t_k^{(j_0)}) \end{aligned}$$

    thus, $\mathcal{X}$ can simply reveal $\rho_k^{(j_0)} = r_{j_0} - t_k^{(j_0)}$
  - note how, if the revealed $a_k^{(j_0)} = 0$, then the other ciphertext $w_k^{(j_0)}$ is $\mathcal{RE}_{pk}(C; t_k^{(j_0)})$, and recall that the Verifier will check:

$$c_i w_k^{(i)} \stackrel{?}{=} \mathcal{E}_{pk}(1; \rho_k^{(i)})$$

    Conveniently:

$$\begin{aligned} c_{j_0} w_k^{(j_0)} &= \mathcal{RE}_{pk}(C^{-1}; r_{j_0}) \mathcal{RE}_{pk}(C; t_k^{(j_0)}) \\ &= \mathcal{E}_{pk}(0; r_{j_0} + t_k^{(j_0)}) \end{aligned}$$

thus, $\mathcal{X}$ can simply reveal $\rho_k^{(j_0)} = r_{j_0} + t_k^{(j_0)}$

- for $i = j_1$, and for $k \in [1, \alpha]$, the same reasoning applies for revealing $\rho_k^{(j_1)}$. Notice that the only difference is that the C and C^{-1} values are swapped:
  - c_{j_1} is a reencryption of C
  - in case $a_k^{(j_1)} = 1$, the $w_k^{(j_1)}$ are reencryptions of C.
  - in case $a_k^{(j_1)} = 0$, the $w_k^{(j_1)}$ are reencryptions of C^{-1}.

  Thus, $\mathcal{X}$ can reveal all the $\rho_k^{(j_1)}$ appropriately.

(h) Thus, $\mathcal{X}$ has appropriately simulated the Prover for $\mathcal{W}^*$. Note that, if $\mathcal{X}$'s input C was the encryption of 0, then $\mathcal{W}^*$ was given correct inputs, but if C is the encryption of 1, $\mathcal{W}^*$ was given swapped indices as inputs.

(i) $\mathcal{X}$ outputs:

- the receipt if its interaction with $\mathcal{W}^*$, exactly as a Prover would do, and
- the output of $\mathcal{W}^*$, effectively the vstate.

So, what happens if $\mathcal{W}^*$ is given swapped inputs? Interestingly, note that $\mathcal{W}^*$ then simulates the interaction for $\mathcal{V}^*$ with index j_1, which is the *correct* index for the public input generated by $\mathcal{X}$. In other words, if $\mathcal{W}^*$ is successfully run on swapped indices, $\mathcal{X}$ outputs a receipt and the true output that $\mathcal{V}^*$ would produce if it engaged in a real proof based on this x_{pub}, with *correct* private input j_1. Note that this public input generated by $\mathcal{X}$ is correctly distributed. (Sometimes, two wrongs do make a right!)

The Purpose of $\mathcal{X}$. In short, if C is the encryption of 0, then the output of $\mathcal{X}$ is a receipt and verifier state that correspond exactly to a real interaction between $\mathcal{W}^*$ and $\mathcal{P}$, with real private input j_0 and claimed private input j_1. If C is the encryption of 1, then the output of $\mathcal{X}$ is a receipt and verifier state that correspond exactly to an interaction between $\mathcal{V}^*$ and $\mathcal{P}$, with real private input j_1.

The construction for Adv$'$ is then clear. It runs two copies of $\mathcal{X}$ with opposite bit ciphertexts C_0 and C_1. Each $\mathcal{X}$ which will run a copy of $\mathcal{W}^*$. The $\mathcal{X}$ that got $C = \mathcal{E}_{pk}(0)$ will simulate a completely correct run of $\mathcal{W}^*$, while the $\mathcal{X}$ that got $C = \mathcal{E}_{pk}(1)$ will simulate a completely correct run of $\mathcal{V}^*$. If C^* can tell us which one is $\mathcal{V}^*$, then it can distinguish between the two runs of $\mathcal{X}$, and thus between C_0 and C_1.

Adv$'$:

(a) obtain the cryptosystem's public key pk from the semantic security environment. The simulated AHIP proof will thus deal with language $\mathcal{L}_{pk}$, defined as the language of proper ciphertext/plaintext pairs under pk. Specifically, the public input will be the ciphertext, and the private input the plaintext.

(b) run $\mathcal{G}^*(pk)$ to obtain the two indices (j, j') that Adv wishes to use in eventually determining which receipt is which. The index j is the index which should truly match the receipt for $\mathcal{V}^*$, while j' is the index which should truly match the receipt for $\mathcal{W}^*$, though $\mathcal{W}^*$ will claim j.

(c) output plaintexts $(0, 1)$ to the semantic security environment as the two plaintexts Adv' will later attempt to distinguish.

(d) obtain from the semantic security environment a challenge ciphertext $C = \mathcal{E}(b)$ for $b \in \{0, 1\}$. Homomorphically compute $\bar{C} = \mathsf{Enc}(\bar{b})$.

(e) initialize and run $\mathcal{X}_0$ and $\mathcal{X}_1$, two instances of $\mathcal{X}$, as follows:

- $(\mathsf{receipt}_0, \mathsf{vstate}_0) = \mathcal{X}_0(pk, z, \alpha, \bar{C}, j', j)$, and
- $(\mathsf{receipt}_1, \mathsf{vstate}_1) = \mathcal{X}_1(pk, z, \alpha, C, j', j)$

Recall that, if $\bar{C} = \mathcal{E}_{pk}(0)$ (respectively if $C = \mathcal{E}_{pk}(0)$), then $\mathcal{X}_0$ (respectively $\mathcal{X}_1$) will produce a receipt and verifier state that are exactly valid for a run of $\langle \mathcal{P}, \mathcal{W}^* \rangle$ with real private input j', and claimed private input j by $\mathcal{W}^*$. On the other hand, if $\bar{C} = \mathcal{E}_{pk}(1)$ (respectively if $C = \mathcal{E}_{pk}(1)$), then $\mathcal{X}_0$ (respectively $\mathcal{X}_1$) will produce a receipt and verifier state that are exactly valid for a run of $\langle \mathcal{P}, \mathcal{V}^* \rangle$ with private input j.

(f) Compute and output $b' = \mathcal{C}^*(\mathsf{receipt}_0, \mathsf{vstate}_0, \mathsf{receipt}_1, \mathsf{vstate}_1)$

From the above construction, one can see that, if $\mathcal{C}^*$ has a non-negligible advantage in guessing b, then it has effectively determined that $C = \mathcal{E}_{pk}(b)$ with the same advantage.

□

5.6 A More Efficient Implementation: BetterBitProof

We now present a second implementation, BetterBitProof, which provides the very same function with a significantly smaller receipt size for the same soundness. Soundness remains optimal given the length of strings a human verifier is capable of comparing. At a high level, the approach is quite similar to that of BitProof: we present a bit encryption scheme to prepare $(\mathbb{c}_1, \ldots, \mathbb{c}_z)$, and we use a special proof method to demonstrate that this is the encryption of bits such that $\mathbb{c}_j$ encrypts 1 and all others encrypt 0.

First, we provide an intuition of what we seek from this new bit encryption mechanism, BetterBitEnc. Then, we detail the BetterBitEnc algorithm and the algebraic structure of the $SO(2, q)$ group on which it relies. We then describe the associated AHIP protocol, BetterBitProof. Finally, we prove that BetterBitProof is indeed private-input hiding, assuming El Gamal is semantically secure.

5.6.1 An Intuition

Consider the properties of BitEnc, the bit encryption used in the previous section. A fixed BitEnc(0) defines a one-to-one-mapping between the α-bit challenges and α-bit responses. In particular, a given response has exactly one corresponding challenge. However, if we fix BitEnc(1), all α-bit challenges correspond to the same α-bit response. Importantly, this correct mapping from a single challenge to a single response can be verified without revealing whether the bit encryption hides a 0 or a 1.

We seek the same property from a different representation, in particular one that doesn't require two El Gamal ciphertexts per bit of the challenge when encrypted. For this, we look at a specific group structure that can encode any plaintext or challenge in a single element, with a means of encrypting a single element from this group using a constant number of El Gamal ciphertexts. In addition, there should be an efficient mechanism proving that a given challenge maps to a given response, given an encrypted bit. We make use of the homomorphic properties of El Gamal for this verification.

5.6.2 Number Theory of $SO(2, q)$

Background. Recall that an orthogonal matrix is a square matrix whose transpose is its inverse. Then, $O(n, F)$ is the orthogonal group of $n \times n$ orthogonal matrices with matrix elements in field F, and with matrix multiplication as the group operation. The group $O(n, q)$ with q a prime then refers to the orthogonal group of $n \times n$ orthogonal matrices with matrix elements in GF_q. Note how, because a matrix's transpose is its own inverse, all matrices in the group must have determinant 1 or -1. Thus, we define $SO(n, q)$, the *special orthogonal group*, as the subgroup of $O(n, q)$ of matrices with determinant 1.

Important Concepts. In this work, we consider specifically $SO(2, q)$, whose elements are of the form:

$$\begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}$$

with $\alpha^2 + \beta^2 = 1$. $SO(2, q)$ is a cyclic group. Its order is $q - 1$ when $q \equiv 1 \bmod 4$, and $q + 1$ when $q \equiv 3 \bmod 4$. In particular, the order of an $SO(2, q)$ group is always a multiple of 4. (We prove this at the end of this subsection.) We denote 4Λ the group order.

The elements of $SO(2, q)$ can be represented as 2-vectors by the correspondence that associates each matrix with its first row. Thus, we consider elements $\boldsymbol{u}$ of the form $(u_0, u_1) = (\alpha, \beta)$, such that $\alpha^2 + \beta^2 = 1$. The group operation, which we denote $\otimes$, is thus:

$$\boldsymbol{u} \otimes \boldsymbol{v} = (u_0 v_0 - u_1 v_1, u_0 v_1 + u_1 v_0)$$

Geometric Interpretation. One can interpret each of these elements in $SO(2, q)$ as rotations. Consider $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in SO(2, q)$ such that $\boldsymbol{w} = \boldsymbol{u} \otimes \boldsymbol{v}$. Then consider the vector dot product: $\boldsymbol{u} \cdot \boldsymbol{w}$. First, we recall $\boldsymbol{w}$:

$$\boldsymbol{w} = (u_0v_0 - u_1v_1, u_0v_1 + u_1v_0)$$

then, we perform the dot product:

$$\begin{aligned}\boldsymbol{u} \cdot \boldsymbol{w} &= u_0^2v_0 - u_0u_1v_1 + u_0u_1v_1 + u_1^2v_0 \\ &= (u_0^2 + u_1^2)v_0 \\ &= v_0\end{aligned}$$

In other words, the dot product between two vector elements is the first coordinate of the ratio between the two vectors, where by ratio we mean to use the inverse group operation. This property matches the geometric interpretation of rotation transformations, where the dot product yields the effective cosine of the angle between the two vectors (as they are both of norm 1.) Note also that the inverse of $\boldsymbol{c}$ retains the same first coordinate v_0, since, in $SO(2,q)$, the inverse of a matrix is its transpose. This is also consistent with the rotation interpretation: whichever direction one rotates, the resulting dot product between the start and end vectors should be the same given a fixed angle of rotation.

The geometric interpretation can be particularly useful when combined with the dot-product operation. In particular, consider $\boldsymbol{g}$ a generator element of $SO(2,q)$. We know from above that:

$$\boldsymbol{g}^i \cdot \boldsymbol{g}^j = \boldsymbol{g}^{j-i}[0]$$

Then, if we have

$$\boldsymbol{g}^i \cdot \boldsymbol{g}^j = \boldsymbol{g}^i \cdot \boldsymbol{g}^k$$

we conclude that:

$$j - i = \pm(k - i) \bmod 4\Lambda$$

We also note an additional property that further explores this geometric interpretation. We now consider all vectors in $\mathbf{Z}_q^2$, in particular certain vectors that can be computed from $SO(2,q)$ vectors. The following uses the normal $\mathbf{Z}_q^2$ vector space subtraction and dot product:

$$\begin{aligned}(\boldsymbol{g}^{(i+2k)} - \boldsymbol{g}^i) \cdot \boldsymbol{g}^{(i+k)} &= (\boldsymbol{g}^{(i+2k)} \cdot \boldsymbol{g}^{i+k}) - (\boldsymbol{g}^i \cdot \boldsymbol{g}^{(i+k)}) \\ &= \boldsymbol{g}^{-k}[0] - \boldsymbol{g}^k[0] \\ &= 0\end{aligned}$$

In the geometric interpretation, this is to be expected, as $\boldsymbol{g}^{(i+k)}$ bisects $\boldsymbol{g}^{(i)}$ and $\boldsymbol{g}^{(i+k)}$. This also leads to the most important $SO(2,q)$-related lemma for this work.

Lemma 5-1 *Given* $\boldsymbol{t}$ *an element of* $SO(2,q)$*, and* $\alpha_0 \in \mathbf{Z}_q$*, there are exactly two elements in* $SO(2,q)$*,* $\boldsymbol{u}_0$ *and* $\boldsymbol{u}_1$*, such that:*

$$\boldsymbol{u}_0 \cdot \boldsymbol{t} = \boldsymbol{u}_1 \cdot \boldsymbol{t} = \alpha_0$$

Proof. Recall that the vector-representation of elements in $SO(2,q)$ is $(\alpha,\beta) \in \mathbf{Z}_q^2$, where $\alpha^2 + \beta^2 = 1 \bmod q$. Denote $\boldsymbol{\delta} \in SO(2,q)$ such that $\boldsymbol{\delta}[0] = \alpha_0$. There are, of course, exactly two such elements, which we can denote $\boldsymbol{\delta}$ and $\boldsymbol{\delta}^{-1}$

Then, by prior reasoning regarding vector dot product:

$$\boldsymbol{u} \cdot \boldsymbol{t} = \alpha_0 \Longrightarrow \boldsymbol{u} = \boldsymbol{\delta}^{\pm 1} \otimes \boldsymbol{t}$$

Denote $\boldsymbol{g}$ a generator of $SO(2,q)$, $c \in \mathbf{Z}_{4\Lambda}^*$ such that $\boldsymbol{t} = \boldsymbol{g}^c$, and $d \in \mathbf{Z}_{4\Lambda}^*$ such that $\boldsymbol{\delta} = \boldsymbol{g}^d$. The vectors $\boldsymbol{u}_0$ and $\boldsymbol{u}_1$ are thus $\boldsymbol{g}^{(c+d)}$ and $\boldsymbol{g}^{(c-d)}$, and no other elements in $SO(2,q)$ can have the same dot product α_0 with $\boldsymbol{t}$.

□

Details on the structure of $SO(2,q)$**.** Let $R = GL(2,q)$ be the full matrix ring generated by $SO(2,q)$:

$$R = \left\{ \begin{pmatrix} a & b \\ -b & a \end{pmatrix} : a, b \in \mathbf{Z}_q \right\}$$

One easily verifies that $R \equiv \mathbf{Z}_q[X]/(X^2+1)$ under $(X) \leftrightarrow \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$.

Lemma 5-2 *If* q *is prime, and* $q \equiv 3(\mathrm{mod} 4)$*, then* $SO(2,q)$ *is cyclic of order* $q+1$*.*

Proof. X^2+1 is an irreducible element of $\mathbf{Z}_q[X]$, hence R is a field and R^* is cyclic. Since det is multiplicative[2], $SO(2,q)$ is a multiplicative subgroup of R^*, *hence is cyclic.*

To deduce its order, note that the map $\alpha(w) = w^{q-1}$ defines a multiplicative homomorphism, $\alpha : R^* \to R^*$. Since det is multiplicative, and since $a^{q-1} = 1$ for all $a \in \mathbf{Z}_q^*$, $\alpha(R^*) \subset SO(2,q)$. On the other hand, there are only two elements of $\mathbf{Z}_q$ whose determinant (norm) is $1 : \{1, -1\}$. Since $q - 1 = 4k + 2$, $-1 \in \alpha(R^*)$, which implies $SO(2,q) \subset \alpha(R^*)$. Thus we conclude:

1. $SO(2,q) = \{ w^{q-1} : w \in R^* \}$.
2. $|SO(2,q)| = |R^*|/|\ker(\alpha)| = (q^2-1)/(q-1) = q+1$.
3. If G is a generator of the multiplicative group R^*, then $g = G^{q-1}$ is a generator of $SO(2,q)$.

□

[2] Note that det is identical to $N_{R|Z_q}$, the *norm* of R over $\mathbf{Z}_q$ as an algebraic extension.

Lemma 5-3 *If q is prime, and $q \equiv 1(\mathrm{mod}4)$, then $SO(2,q)$ is cyclic of order $q-1$.*

Proof.

We can find $i \in \mathbf{Z}_q$ such that $i^2 = -1$. Define matricies A and B in R by

$$A = \begin{pmatrix} 1/2 & i/2 \\ -i/2 & 1/2 \end{pmatrix} \qquad B = \begin{pmatrix} 1/2 & -i/2 \\ i/2 & 1/2 \end{pmatrix}$$

Since $A^2 = A$, $B^2 = B$ and $AB = 0$, each of the following statements can be verified mechanically.

1. Every element of R can be expressed uniquely as $aA + bB$ where $a, b \in \mathbf{Z}_q$.
2. Under this correspondence, R^* consists of exactly those elements with *both* $a \neq 0$ and $b \neq 0$.
3. Further, since $AB = 0$, this correspondence defines a *multiplicative* homomorphism from R^* to $\mathbf{Z}_q^* \times \mathbf{Z}_q^*$. It follows from 2 that this is an *isomorphism.*
4. Under this coorespondence, $SO(2,q)$ consists of exactly those elements of R^* for which $ab = 1$.
5. Hence $SO(2,q)$ is naturally isomorphic to $\mathbf{Z}_q^*$ via either of the two coordinate projections (i.e. $w \longrightarrow a$ or $w \longrightarrow b$). Since $\mathbf{Z}_q^*$ is cyclic of order $q-1$, so is $SO(2,q)$.

□

5.6.3 Bit Encryption

Consider El Gamal in the q-order subgroup of $\mathbf{Z}_p^*$, where $q|(p-1)$. We will, in particular, make use of Exponential El Gamal, like BitProof, where:

$$\mathcal{E}_{pk}(m; r) = (g^r, g^m y^r)$$

where x is the El Gamal secret key, and $y = g^x \bmod p$ is the El Gamal public key.

Defining Classes within $SO(2,q)$. Let Γ be a subgroup of $SO(2,q)$ of order 4λ, where q is the order of the El Gamal subgroup of the underlying cryptosystem. Recall that the order of $SO(2,q)$ is a multiple of 4, and that elements of $SO(2,q)$ are isomorphic to 2-vectors with components in $\mathbf{Z}_q$. Let γ be a generator of $SO(2,q)$. The parameter λ is selected so that $\lambda = \alpha$, where α is the assumed maximal string length that human verifiers can distinguish. We define:

- $\mathsf{Zero} = \{\zeta_i : \zeta_i = \gamma^{4i+1}; i \in [0, \lambda[\}$,
- $\mathsf{One} = \{\vartheta_i : \vartheta_i = \gamma^{4i-1}; i \in [0, \lambda[\}$,
- $\mathsf{Test} = \{\tau_i : \tau_i = \gamma^{2i}; i \in [0, \lambda[\}$.

As the names imply, we consider any element of Zero a 0-plaintext, and any element of One a 1-plaintext. Test is the challenge domain: challenges from the verifier will be selected from this class. Thus, we have defined our group of plaintexts and challenges.

Relating the Classes. We denote $\boldsymbol{\delta}_{k\,c} = \boldsymbol{\vartheta}_k - \boldsymbol{\zeta}_{c-k}$, using standard vector subtraction in $\mathbf{Z}_q^2$, and we note that, in the geometric interpretation, $\boldsymbol{\tau}_c = \boldsymbol{\gamma}^{2c}$ effectively bisects $\boldsymbol{\vartheta}_k = \boldsymbol{\gamma}^{4k-1}$ and $\boldsymbol{\zeta}_{c-k} = \boldsymbol{\gamma}^{4c-4k+1}$. The geometric interpretation is found in Figure 5-6.

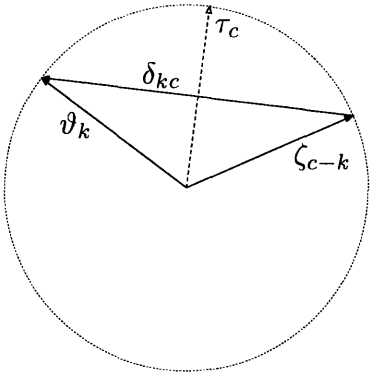


Figure 5-6: The geometric interpretation of the interplay between elements of One, Zero, and Test. Given $\boldsymbol{\vartheta}_k$, an element of One, and a test element $\boldsymbol{\tau}_c$, there is exactly one element of Zero, $\boldsymbol{\zeta}_{c-k}$, such that the corresponding difference vector $\boldsymbol{\delta}_{k\,c}$ is orthogonal to the test vector.

Thus, for a given element $\boldsymbol{t}$ of Test and a given $\boldsymbol{u} \in$ Zero, there is exactly one element $\boldsymbol{u} \in$ One such that $\boldsymbol{u} \cdot \boldsymbol{t} = \boldsymbol{v} \cdot \boldsymbol{t}$ (and no other element of Zero with the same dot product.) This follows from Lemma 5-1 and the careful choice of indices for the classes One, Zero, and Test.

Vector Encryption. We can perform component-wise encryption of vectors in $\mathbf{Z}_q^2$. Using Exponential El-Gamal, we gain the ability to perform homomorphic vector addition of two encrypted vectors, and homomorphic vector dot-product with one encrypted vector and one plaintext vector.

For example, when encrypting $\boldsymbol{s} \in SO(2, q)$, we produce two ciphertexts, one that encrypts $\boldsymbol{s}[0]$, the other that encrypts $\boldsymbol{s}[1]$. We abuse the single-value notation to represent vector encryption:

$$\mathcal{E}_{pk}(\boldsymbol{s}; \boldsymbol{r}) = \Big(\mathcal{E}_{pk}(\boldsymbol{s}[0]; \boldsymbol{r}[0]), \mathcal{E}_{pk}(\boldsymbol{s}[1]; \boldsymbol{r}[1])\Big)$$

Note that, with vector components in exponential El Gamal ciphertext, homomorphic vector addition is trivial using component-wise homomorphic multiplication. In particular, this feature is available because q, the order of the El-Gamal subgroup, matches the order of the vector-component group.

In addition, if we denote $\boldsymbol{Q} = \mathcal{E}_{pk}(\boldsymbol{s}; \boldsymbol{r})$, then we can homomorphically compute an encryption of the dot product $\boldsymbol{s} \cdot \boldsymbol{t}$ where $\boldsymbol{t}$ is a plaintext element of $\mathbf{Z}_q^2$. We use the multiply-by-a-constant homomorphic property and ciphertext-by-ciphertext additive homomorphism:

$$\boldsymbol{Q} \cdot \boldsymbol{t} = (\boldsymbol{Q}[0]^{\boldsymbol{t}[0]} \oplus \boldsymbol{Q}[1]^{\boldsymbol{t}[1]}) = \mathcal{E}_{pk}(\boldsymbol{s} \cdot \boldsymbol{t}; \boldsymbol{r} \cdot \boldsymbol{t})$$

Notice how the result is a single ciphertext, which is to be expected when performing a vector dot-product. Notice also how the randomization value in the resulting ciphertext is also the result of a dot-product, given that the homomorphic operation actually performs an *exponentiation* of the El Gamal ciphertexts. Recall that we are dealing with Exponential El Gamal, where exponentiation to a plaintext multiplies the contained plaintext *and* randomization exponent, and multiplication of ciphertexts yields a homomorphic addition with the sum of the random exponents:

$$\begin{aligned}
\boldsymbol{Q} \cdot \boldsymbol{t} &= (\boldsymbol{Q}[0]^{\boldsymbol{t}[0]} + \boldsymbol{Q}[1]^{\boldsymbol{t}[1]}) \\
&= \mathcal{E}_{pk}(\boldsymbol{s}[0]; \boldsymbol{r}[0])^{\boldsymbol{t}[0]} \times \mathcal{E}_{pk}(\boldsymbol{s}[1]; \boldsymbol{r}[1])^{\boldsymbol{t}[1]} \\
&= \mathcal{E}_{pk}(\boldsymbol{s}[0]\boldsymbol{t}[0]; \boldsymbol{r}[0]\boldsymbol{t}[0]) \times \mathcal{E}_{pk}(\boldsymbol{s}[1]\boldsymbol{t}[1]; \boldsymbol{r}[1]\boldsymbol{t}[1]) \\
&= \mathcal{E}_{pk}\left(\boldsymbol{s}[0]\boldsymbol{t}[0] + \boldsymbol{s}[1]\boldsymbol{t}[1]; \boldsymbol{r}[0]\boldsymbol{t}[0] + \boldsymbol{r}[1]\boldsymbol{t}[1]\right) \\
&= \mathcal{E}_{pk}(\boldsymbol{s} \cdot \boldsymbol{t}; \boldsymbol{r} \cdot \boldsymbol{t})
\end{aligned}$$

Bit Encryption. Let b be the bit we wish to encrypt. We define an $SO(2, q)$-based method of bit encryption. First, pick a random vector $\gamma^{4r'+2b+1}$ using randomness $r' \in \mathbf{Z}_\lambda$. This vector will be in Zero if $b = 0$, and in One if $b = 1$. Then, perform the exponential El-Gamal vector encryption of this vector using randomness $\boldsymbol{r} \in \mathbf{Z}_q^2$:

$$\mathsf{BetterBitEnc}_{pk}(b, (\boldsymbol{r}, r')) \;=\; \mathcal{E}_{pk}(\gamma^{4r'+2b+1}; \boldsymbol{r})$$

In comparison to the prior protocol, BitProof, one can draw clear parallels for the random values (r', r). Random value r' selects the specific element within One, which is the equivalent of the choices of the exact u, v pairs in BitEnc, which, as we saw, depend on the random bits $a_k^{(i)}$ that are part of R, the randomization value for BitEnc. Random value r is the randomness used in the encryption process, which is exactly the same thing as in BitEnc, though here there are only two ciphertexts rather than α, and thus only two randomization values denoted in vector form, $\boldsymbol{r}$.

5.6.4 The BetterBitProof Protocol

The BetterBitProof protocol presents the exact same interface as BitProof, except BitEnc is now replaced by BetterBitEnc, and the exact proof algebra is optimized in $SO(2, q)$. We assume the same communication model and, as we will see, the same form of inputs.

Protocol BetterBitProof:

Consider the AHIP setup, with prover $\mathcal{P}$, verifier $(\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}})$, and assistant $\mathcal{A}$. $(\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}})$ are computationally limited, with sole ability to compare and copy short strings of size $\alpha = \log_2 \lambda$. Consider the proof inputs:

- public common input $x_{pub} = (pk, \gamma, \lambda, c_1, \ldots, c_z)$ where pk is an El Gamal public key, where q is the order of the El Gamal group, and γ is a generator of Γ, a 4λ-order subgroup of $SO(2, q)$ in vector representation. The c_i are Exponential El Gamal ciphertexts such that:

$$\exists j' \in [1, z], c_{j'} = \mathcal{E}_{pk}(1) \text{ and } \forall i \neq j', c_i = \mathcal{E}_{pk}(0)$$

  Denote b_i such that $c_i = \mathcal{E}_{pk}(b_i)$. (Thus $b_{j'} = 1$ and $b_i = 0$ otherwise.)

- private common input $x_{priv} = j \in [1, z]$
- secret prover input $(\omega_1, \ldots, \omega_z)$, such that $\forall i \in [1, z], c_i = \mathcal{E}_{pk}(b_i; \omega_i)$.

The proof protocol is thus denoted exactly as for BitProof, with the same "interface":

$$\left\langle \mathcal{P}(pk, c_1, c_2, \ldots, c_z, j, \omega_1, \omega_2, \ldots, \omega_z), \mathcal{V}_{\mathsf{int}}(pk, c_1, c_2, \ldots, c_z, j) \right\rangle$$

and proceeds as follows:

1. $\mathcal{V}_{\mathsf{int}}$ sends *commit*(chal) to $\mathcal{P}$, where chal $\in \mathbf{Z}_\lambda^*$. This challenge chal should be interpreted as the index of an element of Test.

2. $\mathcal{P}$ prepares:

   (a) $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_z$, randomization vectors in $\mathbf{Z}_q^2$

   (b) $k_j \xleftarrow{R} [1, \lambda[$, a random index into One and the associated element $\boldsymbol{\vartheta}_{k_j}$. $\boldsymbol{\vartheta}_{k_j}$ is effectively a randomly selected "1-vector" which will be used when producing the special bit encryption of b_j,

   (c) $(l_i)_{i \in [1,z], i \neq j}$, indexes into Zero, and the associated elements $(\boldsymbol{\zeta}_{l_i})$, the randomly selected "0-vectors" which will be used in the special bit encryptions of $b_i, i \neq j$.

   (d) $\boldsymbol{Q}_j = \mathcal{E}_{pk}(\boldsymbol{\vartheta}_{k_j}; \boldsymbol{r}_j)$, for the chosen index j,

   (e) $\boldsymbol{Q}_i = \mathcal{E}_{pk}(\boldsymbol{\zeta}_{l_i}; \boldsymbol{r}_i)$, for all $i \in [1, z], i \neq j$,

   then *prints on* receipt the values $(\boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_z)$.

   $\mathcal{P}$ also sends k_j to $\mathcal{V}_{\mathsf{int}}$, but *doesn't include it in the receipt.* This is the prover's commitment to the exact element of One that corresponds to the index j. k_j effectively indicates the plaintext for $\boldsymbol{Q}_j$. (This is the equivalent of ChosenString in BitProof.)

3. $\mathcal{V}_{\mathsf{int}}$ reveals chal to Prover. If it doesn't match *commit*(chal), $\mathcal{P}$ *aborts.*

4. Recall that chal corresponds to $\boldsymbol{\tau}_c$, an element of Test. $\mathcal{P}$ then computes:

   (a) $k_i = c - l_i$, for $i \neq j$, an index into One for all the options that were *not* chosen. These are the simulated elements of One for all the other options, now that the prover knows the challenge chal.

   (b) $l_j = c - k_j$, an index into Zero for the chosen option j. This is the simulated element of Zero for the one selected option j, now that the prover knows chal.

   (c) $(\boldsymbol{\delta}_{k_1 c}, \ldots, \boldsymbol{\delta}_{k_z c})$, the difference vectors between $\boldsymbol{\zeta}_{l_i}$ and $\boldsymbol{\vartheta}_{k_i}$. Note that $\boldsymbol{\delta}_{k_i c}$ is orthogonal to $\boldsymbol{\tau}_c$ (as per the proof of Lemma 5-1.) Note also how these difference vectors are independent of the option j: they depend on the initial randomization values selected for bit encryption *and* on the verifier's challenge.

   (d) $\rho_{i,0} = \boldsymbol{r}_i \cdot \boldsymbol{\delta}_{k_i c} - ||\boldsymbol{\delta}_{k_i c}||^2 \omega_i$; $\rho_{i,1} = \boldsymbol{r}_i \cdot \boldsymbol{\tau}_c$, for all $i \in [1, z]$. These are effectively the revealed random factors of dot products that the verifier will be able to perform homomorphically. We clarify these random factors in the next few paragraphs.

   $\mathcal{P}$ prints to receipt $\left(\{k_i, \rho_{i,0}, \rho_{i,1}\}_{i \in [1,z]}\right)$, which now yields:

$$\mathsf{receipt} = \Big(c_1, c_2, \ldots, c_z, \mathbb{c}_1, \mathbb{c}_2, \ldots, \mathbb{c}_z, \mathsf{chal}_{\mathcal{P}}, k_1, \ldots, k_z,$$
$$\rho_{1,0}, \ldots, \rho_{z,0}, \rho_{1,1}, \ldots, \rho_{z,1}\Big)$$

5. $\mathcal{V}_{\mathsf{int}}$ outputs $\mathsf{vstate} = (j, \mathsf{chal}_{\mathcal{V}}, k)$ for $\mathcal{V}_{\mathsf{check}}$ to consider.

6. $\mathcal{V}_{\mathsf{check}}$ outputs True if both $\mathsf{chal}_{\mathcal{P}} = \mathsf{chal}_{\mathcal{V}}$ and $k = k_j$.

7. $\mathcal{A}$ reads receipt and performs the following checks, noting that we consider $c = \mathsf{chal}$:

   (a) Denote homomorphic subtraction as $\ominus$.
   Compute $\boldsymbol{Q}'_i = \boldsymbol{Q}_i \ominus \mathcal{E}_{pk}(\boldsymbol{\zeta}_{c-k_i}, \boldsymbol{0}), \forall i \in [1, z]$.
   Thus, $\boldsymbol{Q}'_i$ is either the null vector (if $i \neq j$), or the difference vector $\boldsymbol{\delta}_{k_i c}$. In either case, $\boldsymbol{Q}'_i$ is orthogonal to $\boldsymbol{\tau}_c$. Note that this homomorphic subtraction is performed so that $\boldsymbol{Q}'_i$ keeps the same randomization exponents as $\boldsymbol{Q}_i$, which is $\boldsymbol{r}_i$.

   (b) homomorphically compute $\boldsymbol{Q}'_i \cdot \boldsymbol{\tau}_c$.
   The plaintext of this value should be 0, because of the orthogonality just mentioned. The randomness of the dot product should be $\boldsymbol{r}_i \cdot \boldsymbol{\tau}_c = \rho_{i,1}$. $\mathcal{A}$ can simply perform $\mathcal{E}_{pk}(0; \rho_{i,1})$ and check this value against the computed dot-product.

   (c) homomorphically compute $\boldsymbol{Q}'_i \cdot \boldsymbol{\delta}_{k_i c}$.
   As $\boldsymbol{Q}'_i$ is either the null vector or $\boldsymbol{\delta}_{k_i c}$, the very same difference vector against which we are computing the dot product, the result is the encryption of 0 or $||\boldsymbol{\delta}_{k_i c}||^2$, under randomness $\boldsymbol{r}_i \cdot \boldsymbol{\delta}_{k_i c}$. The trick now is to take the original i'th

ciphertext, $c_i = \mathcal{E}_{pk}(b_i; \omega_i)$, and use it to verify that $\mathbb{c}_i$ indeed encodes the same bit as c_i. Recall that $\rho_{i,0} = \boldsymbol{r}_i \cdot \boldsymbol{\delta}_{k_i c} - ||\boldsymbol{\delta}_{k_i c}||^2 \omega_i$. Thus, $\mathcal{A}$ checks that

$$\frac{\boldsymbol{Q}'_i \cdot \boldsymbol{\delta}_{k_i c}}{c_i^{||\boldsymbol{\delta}_{k_i c}||^2}} \overset{?}{=} \mathcal{E}_{pk}(0; \rho_{i,0})$$

If $\mathbb{c}_i$ is the bit encryption of 1, then $\boldsymbol{Q}'_i = \boldsymbol{\delta}_{k_i c}$, and the numerator of the fraction is the Exponential El Gamal encryption of $||\boldsymbol{\delta}_{k_i c}||^2$ with random exponent $\boldsymbol{r}_i \cdot \boldsymbol{\delta}_{k_i c}$. If c_i is the Exponential El Gamal encryption of 1—i.e. it matches $\mathbb{c}_i$ as expected—then the denominator is the Exponential El Gamal encryption of $||\boldsymbol{\delta}_{k_i c}||^2$ with random exponent $||\boldsymbol{\delta}_{k_i c}||^2 \omega_i$, since ω_i was the random exponent for c_i. The equality check is thus successful.

If $\mathbb{c}_i$ is the bit encryption of 0, then $\boldsymbol{Q}'_i$ is the null vector, and the numerator of the fraction is the Exponential El Gamal encryption of 0 with random exponent $\boldsymbol{r}_i \cdot \boldsymbol{\delta}_{k_i c}$. If c_i is the Exponential El Gamal encryption of 0—i.e. it matches $\mathbb{c}_i$ as expected—then the denominator is the Exponential El Gamal encryption of 0 with random exponent $||\boldsymbol{\delta}_{k_i c}||^2 \omega_i$. Again, the equality check is successful.

5.6.5 BetterBitProof is a Private-Input Hiding AHIP

Theorem 5-3 BetterBitProof *is a private-input hiding AHIP.*

Proof. Note, again, that we assume that $(c_1, c_2, \dots, c_z)$ is well-formed, meaning that exactly one ciphertext is the encryption of 1, and all others are encryptions of 0. The proof is meant to demonstrate that j is indeed the index of the ciphertext that is a bit encryption of 1.

1. **Completeness.** The honest construction yields $\mathbb{c}_j = \mathsf{BetterBitEnc}(1)$, and $\mathbb{c}_i = \mathsf{BetterBitEnc}(0)$ otherwise. Thus, given c from chal, $\boldsymbol{Q}'_j$ will be $\mathcal{E}_{pk}(\boldsymbol{\delta}_{k_j c}; \boldsymbol{r}_j)$, while $\boldsymbol{Q}'_i$ will be $\mathcal{E}_{pk}(\boldsymbol{0}; \boldsymbol{r}_i)$ for all other i. In all cases, the dot product with $\boldsymbol{\tau}_c$ will be 0, with randomness $\boldsymbol{r}_i \cdot \boldsymbol{\tau}_c$, which will cause the check for $\rho_{i,1}$ to succeed. In addition, the check that $\mathbb{c}_i$ encodes the same bit as c_i will also succeed, as carefully described in the original protocol regarding the dot-product of $\boldsymbol{Q}_i$ with $\boldsymbol{\delta}_{k_i c}$. $\mathcal{V}_{\mathsf{check}}$'s verification is also trivial: $\mathcal{P}$ commits to k_j, which will clearly show up, if $\mathcal{P}$ is honest, as the j'th value of k on the receipt.

2. **Soundness.** If the prover is cheating, we denote him $\mathcal{P}^*$. As we continue to assume well-formed ciphertext sets, we know that $c_j = \mathcal{E}_{pk}(0)$. Two situations occur: either $\mathbb{c}_j = \mathsf{BetterBitEnc}_{pk}(0)$, or $\mathbb{c}_j$ is the bit-encryption of something else.

   In the first case, $\mathcal{A}$'s verification that, for all i, $\mathbb{c}_i$ and c_i encode the same bit will succeed, but recall that $\mathcal{P}^*$ sends k_j before seeing chal. For a given $\mathbb{c}_j = \mathsf{BetterBitEnc}_{pk}(0) = \boldsymbol{Q}_j$, each test vector $\boldsymbol{\tau}$, indexed by chal, maps to a single "1-vector" $\boldsymbol{\vartheta}$ such that $\boldsymbol{\vartheta} \cdot \boldsymbol{\tau} = \beta$ with β the dot-product of $\boldsymbol{\tau}$ with the Zero element encrypted within $\boldsymbol{Q}_j$. The specific element of $\boldsymbol{\vartheta}$ is the one whose index we compute in the protocol: k_j. For $\mathcal{P}^*$ to succeed, it must guess this k_j correctly before it sees chal. The best it can do is guess randomly, with a probability of success of $1/\lambda = 2^{-\alpha}$.

In the second case, $\mathbb{c}_j$ and c_j do not encode the same bit, in which case the straightforward algebraic dot-product check will cause $\mathcal{A}$ to declare failure with probability 1, since $\mathcal{P}$ will simply not be able to reveal a random factor that lets the homomorphic computation succeed.

3. **Private-Input Hiding.** An adversary that could distinguish between receipts for various values of j could break semantic security of the underlying public-key cryptosystem, which in this case we assume, by construction, is Exponential El Gamal. Assume Adv extracts the private input j. We construct Adv$'$ that breaks the semantic security of El Gamal.

   This construction is very similar to that for BitProof. We construct $\mathcal{W}^*$ that uses $\mathcal{V}^*$ as a black box, extracting the challenge from $\mathcal{V}^*$ in order to trick $\mathcal{V}^*$ into accepting a receipt for the wrong value of j. This is doable because, knowing chal, one can encrypt an element of Zero but claim the index for the corresponding element of One. Note, for emphasis, that this correspondence between elements of Zero and One is defined by the specific challenge chal, which must thus be extracted before any real simulation ensues. This allows $\mathcal{W}^*$ to successfully convince $\mathcal{V}^*$ that the receipt encodes a particular j (even though it encodes j').

   Then, like for BitProof, we construct a special extractor $\mathcal{X}$, whose job it is to simulate the Prover in an interaction with $\mathcal{W}^*$ based on an Exponential El Gamal encrypted bit C, whose plaintext is unknown to $\mathcal{X}$. $\mathcal{X}$ can clearly rewind $\mathcal{W}^*$ to get chal ahead of time. In addition, $\mathcal{X}$ must be able to create randomly distributed encrypted elements of One or Zero, where the class membership of the element depends on the plaintext of the input ciphertext C, which $\mathcal{X}$ cannot decrypt. This dependence must be such that, when $\mathcal{A}$ performs its last verification – that $\mathbb{c}_i$ and c_i encode the same bit –, the random exponent of c_i is cancelled out, because $\mathcal{X}$ will only know the appropriate randomization values. We now describe how $\mathcal{X}$ can accomplish these operations.

Creating the Public Input. Recall that $\mathcal{X}$ receives $C = \mathcal{E}_{pk}(b)$ as input, as well as parameters z and indices j_0 and j_1, both in $[1, z]$. It is trivial for $\mathcal{X}$ to homomorphically compute $C^{-1} = \mathcal{E}_{pk}(\bar{b})$. Note that C^{-1} has randomization exponent $-r$ if the randomization exponent of C is r. In this section, we now denote $\bar{C} = C^{-1}$, because will need to exponentiate this value, and a double-exponent notation risks confusion. Next, $\mathcal{X}$ can create $(c_1, \ldots, c_z)$ exactly like in the BitProof case, for $i \in [1, z]$:

- if $i \neq j_0$ and $i \neq j_1$, select r_i and compute $c_i = \mathcal{E}_{pk}(0; r_i)$.
- if $i = j_0$, select r_{j_0}, and compute $c_{j_0} = \mathcal{RE}_{pk}(\bar{C}, r_{j_0})$.
- if $i = j_1$, select r_{j_1}, and compute $c_{j_1} = \mathcal{RE}_{pk}(C, r_{j_1})$.

Thus, the generated public input $(c_1, \ldots, c_z)$ matches private input $x_{priv} = j_b$, though $\mathcal{X}$ does not know b.

Creating Encrypted Elements of One and Zero. Now, given a challenge vector $\boldsymbol{\tau}_c$ corresponding to $c = \mathsf{chal}$, $\mathcal{X}$ needs to sample encryptions of elements $\boldsymbol{\vartheta}_k$ or $\boldsymbol{\zeta}_{c-k}$, depending on b, for any value of k. In addition, $\mathcal{X}$ must be able to reveal the randomization value of the related dot-product. To do this, $\mathcal{X}$ must sample the encrypted elements with just the right randomization values so that, when the dot-product is performed, the randomization values unknown to $\mathcal{X}$ cancel out. Thus, $\mathcal{X}$ proceeds as follows, given c and k:

- consider the plaintext vectors $\boldsymbol{\vartheta}_k$ and $\boldsymbol{\zeta}_{c-k}$
- select random exponents $\boldsymbol{s} = (s_0, s_1)$, and compute:

$$\boldsymbol{Q}_{k,c} = \left(\mathcal{RE}_{pk}\left(C^{\boldsymbol{\vartheta}_k[0]}(\bar{C})^{\boldsymbol{\zeta}_{c-k}[0]}; s_0\right), \mathcal{RE}_{pk}\left(C^{\boldsymbol{\vartheta}_k[1]}\bar{C}^{\boldsymbol{\zeta}_{c-k}[1]}; s_1\right)\right)$$

  Note how, if C encrypts 1, $\boldsymbol{Q}_{k,c}$ is the encryption of $\boldsymbol{\vartheta}_k$, but if C encrypts 0, $\boldsymbol{Q}_{k,c}$ is the encryption of $\boldsymbol{\zeta}_{c-k}$. In both cases, the randomization exponents for the vector are:

$$(s_0 + r(\boldsymbol{\vartheta}_k[0] - \boldsymbol{\zeta}_{c-k}[0]), s_1 + r(\boldsymbol{\vartheta}_k[1] - \boldsymbol{\zeta}_{c-k}[1]))$$

  Recall that the difference vector $\boldsymbol{\delta}_{k\,c} = \boldsymbol{\vartheta}_k - \boldsymbol{\zeta}_{c-k}$. Thus, we can rewrite the random exponents of our sampled $\boldsymbol{Q}_{k,c}$ as follows, noting again for emphasis that r is unknown to $\mathcal{X}$.

$$(s_0 + r\boldsymbol{\delta}_{k\,c}[0], s_1 + r\boldsymbol{\delta}_{k\,c}[1])$$

- Recall the computation that the Assistant will perform:

$$\boldsymbol{Q}'_{k,c} = \boldsymbol{Q}_{k,c} - \mathcal{E}_{pk}(\boldsymbol{\zeta}_{c-k}; \boldsymbol{0})$$

  which will yield an encryption of either the null vector if C encrypts 0, or the difference vector $\boldsymbol{\delta}_{k\,c}$ if C encrypts 1. In both cases, the random exponents remains unchanged, since this is a homomorphic subtraction of a trivial encryption.

- When it comes time to perform the vector dot product, the plaintext of $\boldsymbol{Q}'_{k,c} \cdot \boldsymbol{\tau}_c$ will clearly be the null vector, given that $\boldsymbol{Q}'_{k,c}$ is either the null vector itself, or the difference vector $\boldsymbol{\delta}_{k\,c}$. More interestingly, the single random exponent of this dot product is:

$$
\begin{aligned}
(s_0 + r\boldsymbol{\delta}_{k\,c}[0])\boldsymbol{\tau}_c[0] + (s_1 + r\boldsymbol{\delta}_{k\,c}[1])\boldsymbol{\tau}_c[1] &= \boldsymbol{s} \cdot \boldsymbol{\tau}_c + r(\boldsymbol{\delta}_{k\,c} \cdot \boldsymbol{\tau}_c) \\
&= \boldsymbol{s} \cdot \boldsymbol{\tau}_c
\end{aligned}
$$

Thus, we have successfully sampled an encryption of either $\boldsymbol{\zeta}_{c-k}$ or $\boldsymbol{\vartheta}_k$ given parameters c and k, based the ciphertext C. The extractor $\mathcal{X}$ can produce $\boldsymbol{Q}_i$ for $i \in [1, z]$ as follows, knowing $c = \mathsf{chal}$ which it already extracted from $\mathcal{W}^*$:

- if $i \neq j_0$ and $i \neq j_1$, pick a random element from Zero, and compute $\boldsymbol{Q}_i$ as its encryption with fresh random exponents.
- if $i = j_0$, then pick a random $k_0 \in [1, \lambda[$ and compute $\boldsymbol{Q}_{k_0,c}$ with C and $\bar{C}$ reversed.
- if $i = j_1$, then pick a random $k_1 \in [1, \lambda[$ and compute $\boldsymbol{Q}_{k_1,c}$ as described above, with C and $\bar{C}$ as described.

Thus, if $C = \mathcal{E}_{pk}(b)$, then the $(\boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_z)$ correspond to a proper set of encrypted vectors for private input j_b. The extractor $\mathcal{X}$ is able to reveal the randomization value $\rho_{i,1} = \boldsymbol{s}_i \cdot \boldsymbol{\tau}_c$. This means that $\mathcal{X}$ can now simulate the Prover's actions to $\mathcal{W}^*$, except for one last step.

The remaining step is to reveal $\rho_{i,0}$ for $i \in [1, z]$, as a means of proving that $\mathbb{c}_i = \boldsymbol{Q}_i$ encrypts the same bit as c_i, the input ciphertext. For $i \neq j_0$ and $i \neq j_1$, all random factors are known to $\mathcal{X}$, so there is no difficulty in revealing the proper $\rho_{i,0}$. We consider now the cases of j_0 and j_1.

Recall that $\mathcal{A}$ will verify $\rho_{i,0}$ as follows

$$
\frac{\boldsymbol{Q}'_i \cdot \boldsymbol{\delta}_{k_i\,c}}{c_i^{||\boldsymbol{\delta}_{k_i\,c}||^2}}
$$

Conveniently, we generated $\boldsymbol{Q}'_{k_0,c}$ and $\boldsymbol{Q}'_{k_1,c}$ such that they will correctly cancel out the plaintext portion of the ciphertexts in the above equation, and yield an encryption of 0. As always, the interesting element is the randomization value. Recall that, with r the random exponent for C, the random exponent for c_{j_0} is $r_0 - r$, and the random exponent for c_{j_1} is $r_1 + r$ (this should be clear from how c_{j_0} and c_{j_1} were computed).

Thus, for j_0, the random exponent for the denominator above is:

$$
||\boldsymbol{\delta}_{k_0\,c}||^2(r_0 - r))
$$

Meanwhile, the random exponent for $\boldsymbol{Q}'_{k_0,c}$, which inverted the use of C and $\bar{C}$ from our generic equation for the random exponent of $\boldsymbol{Q}_{k,c}$, is:

$$(s_0 - r\boldsymbol{\delta}_{k_0 c}[0], s_1 - r\boldsymbol{\delta}_{k_0 c}[1])$$

Thus, the random exponent for the numerator of the fraction in the case of index j_0 is:

$$\boldsymbol{s} \cdot \boldsymbol{\delta}_{k_0 c} - r(||\boldsymbol{\delta}_{k_0 c}||^2)$$

The resulting random factor for the whole fraction thus sees the unknown r cancel out, and yields:

$$\rho_{j_0,0} = s_0 \cdot \boldsymbol{\delta}_{k_0 c} + r_0||\boldsymbol{\delta}_{k_0 c}||^2$$

Notice how $\mathcal{X}$ can compute this value of $\rho_{j_0,0}$, and can thus complete the simulation for j_0. One can go through the similar computation to show that, because the r also cancels out in the case of j_1:

$$\rho_{j_1,0} = s_1 \cdot \boldsymbol{\delta}_{k_1 c} + r_1||\boldsymbol{\delta}_{k_1 c}||^2$$

and $\mathcal{X}$ is able to full simulate the entire protocol perfectly.

Finally, we create Adv', the semantic-security adversary, exactly like the one for $\mathsf{BitProof}$. Using two instances of $\mathcal{X}$, each of which invokes its instance of $\mathcal{W}^*$ with a homomorphically flipped ciphertext C, Adv' can use $\mathcal{C}^*$ on the outputs of both instances of $\mathcal{X}$ to distinguish whether C encrypts 0 or 1, thus breaking the semantic security of Exponential El Gamal.

□

5.7 Ensuring Well-Formed Ciphertexts/Ballots

In both $\mathsf{BitProof}$ and $\mathsf{BetterBitProof}$, we have assumed that encryption inputs $(c_1, c_2, \ldots, c_z)$ are *well-formed*, meaning that, for exactly one value of j, $c_j = \mathcal{E}_{pk}(1)$, and for all other values i, $c_i = \mathcal{E}_{pk}(0)$. The protocols have then focused on proving that the *correct value of* j has been used. Here, we examine this assumption of well-formedness, and show why it is reasonable.

5.7.1 The Voting Setting

In the voting setting, these ciphertext tuples $(c_1, c_2, \ldots, c_z)$ are, in fact, encrypted ballots, where the j position indicates the candidate of choice. Typically, these ballots will then be processed through a mixnet, in which case they are eventually decrypted. It is quite reasonable to assume that a malformed ballot will thus be detected at tallying time, at which point one can ask the mix servers to back-track to the responsible voting machine which will then get sanctioned in some way. The pressure on manufacturers to thus produce well-formed ballots should be enough to ensure this aspect of the encrypted ballot.

5.7.2 Forcing Well-Formed Ciphertexts

If the assumption of eventual decryption and tracing is not enough, one can augment BitProof or BetterBitProof with an additional proof of well-formed ciphertext tuples. As it turns out, it is relatively easy to perform this proof in the AHIP setting: because the well-formed nature of the ciphertexts is certainly not secret, one can fully rely on $\mathcal{A}$ to check it. The only delicate aspect is to ensure that, in proving the well-formed nature of the ciphertext tuple, $\mathcal{P}$ should not reveal j.

The approach for such a proof is to provide a proof of partial knowledge using the technique of Cramer et al. [44] to prove that one of the following assertions is true:

- $c_1 = \mathcal{E}_{pk}(1)$, and $\forall i \neq 1, c_i = \mathcal{E}_{pk}(0)$, or
- $c_2 = \mathcal{E}_{pk}(1)$, and $\forall i \neq 2, c_i = \mathcal{E}_{pk}(0)$, or

  $\vdots$
- $c_z = \mathcal{E}_{pk}(1)$, and $\forall i \neq m, c_i = \mathcal{E}_{pk}(0)$

These proofs can either be performed interactively by having $\mathcal{V}$ provide a longer challenge to account for this additional proof, or by using some non-interactive zero-knowledge proof purely printed on the receipt, letting $\mathcal{A}$ deal with it entirely. $(\mathcal{V}_{\mathsf{int}}, \mathcal{V}_{\mathsf{check}})$ then only deals with checking that the right index j has been selected.

5.8 Conclusion

We suggest that voting research should consider *ballot casting assurance* as a complement to universal verifiability. While universal verifiability provides global auditability that votes are counted correctly, ballot casting assurance provides individual assurance to Alice that her vote "made it" to the input of the tally process as intended. If an error is detected, Alice can revote until verifiable rectification.

We achieve ballot casting assurance with cryptographic schemes that implement secret receipts. In particular, we proposed a model for cryptographic proofs where the verifier is "only human," and any assistant must not learn the crucial private input. We call these protocols Assisted-Human Interactive Proofs (AHIPs). A number of questions remain. How usable are these systems in real-world tests? How will revoting really work? In any case, we believe that this issue should be considered as an integral part of voting system design and evaluation.

Chapter 6

Public Mixing

This chapter is an extension of "How to Shuffle in Public" [7], which is joint work with Douglas Wikström. This work is currently in submission.

6.1 Introduction

In ballot-preserving voting schemes, the anonymization step implemented by a mixnet is the most complicated and time-intensive portion. Though recent mixnet algorithms [124, 70] are remarkably efficient, the required trusted computing base remains complex: the shuffle is performed on election day, but the shuffle's details must remain secret. In practice, it is not clear that this can be done securely, as a number of applied security experts have noted [104].

In this chapter, we strive to reduce the trusted computing base of a mixnet, so that the process on election day can be simplified. We use two major strategies. First, we shift some computation from the private space to the public space, so that more tasks can be performed by untrusted components and verified by straight-forward re-computation. Second, we perform proofs *before the election*, so that election day becomes more predictable and less error-prone.

There are limits to these strategies. It is clear that the election process—casting, anonymization, tallying, and decryption—cannot be performed entirely by public computation: an adaptive attacker could determine any single individual's vote by running the victim's encrypted vote through an entirely simulated election where the adversary knows all other vote plaintexts. This is an inherent limitation of the abstract voting functionality: an adversary who knows how a number of individuals voted can gain information about other votes simply by following the normal election protocol.

Instead, we focus our attention on the shuffling phase: can we mix and rerandomize the ciphertexts using public computation? If so, can we perform all proofs on this public source code ahead of time? There is no inherent obstacle to this goal. An adversary can simulate the ciphertext shuffling all he wants, voter privacy will be preserved by the decryption step, which remains a private operation performed by trusted officials.

This goal is much like tallying by homomorphic aggregation, where anyone can perform the homomorphic sum, but only officials can eventually decrypt the tally. In a sense, we are looking to perform "homomorphic shuffling," the encrypted computation of a permutation.

Program Obfuscation. One can interpret our goal as the program obfuscation of a reencryption mixnet. As we have seen in Chapter 2, program obfuscation is the process of "muddling" a program's instructions to prevent reverse-engineering while preserving proper function. Barak et al. first formalized obfuscation as simulatability from black-box access [12]. Goldwasser and Tauman-Kalai extended this definition to consider auxiliary inputs [78]. Though simple programs have been successfully obfuscated [31, 173]. generalized program obfuscation has been proven impossible in even the weakest of settings for both models (by their respective authors).

Ostrovsky and Skeith [132] consider a weaker model, public-key obfuscation, where the obfuscated program's output is encrypted. In this model, they achieve the more complex application of private stream searching. It is this model we consider for our constructions, as it captures the security properties we need quite naturally. However, for completeness, we also point out that our constructions can be modeled using the proposals of Barak et al. and Goldwasser and Tauman-Kalai.

6.1.1 Contributions

We show how to obfuscate the shuffle phase of a mixnet. We begin with general assumptions and show how any additive homomorphic cryptosystem can provide public mixing. Unfortunately, this generic construction results in an extremely inefficient mixnet, because the verifiable decryption step portion requires generic zero-knowledge proofs. We focus then on specific cryptosystems with additional useful properties. We show how special—and distinct—properties of the Boneh-Goh-Nissim [23] and Paillier [133] cryptosystems enable public mixing with sufficient efficiency to be practical.

We formalize our constructions in the public-key obfuscation model of Ostrovsky and Skeith, whose indistinguishability property closely matches the security requirements of a mixnet. We also show how to prove, in zero-knowledge, the correct obfuscation of a shuffle. Finally, we describe a protocol that allows a set of parties to jointly and robustly generate an obfuscated randomly chosen shuffle. These constructions require considerably more exponentiations, $O(\kappa N^2)$ instead of $O(\kappa N)$, than private mixnet techniques. However, they remain reasonably practical for precinct-based elections, where voters are anonymized in smaller batches and all correctness proofs can be carried out in advance.

We also provide an overview of a different formalization for these same constructions, using the obfuscation model of Barak et al. This approach is quite a bit more contrived than the prior construction, as it turns out that the obfuscation property of Barak et al. and Goldwasser-Tauman doesn't fully capture the security we need from our mixnet functionality. Thus, we are forced to prove a separate security property in addition to correct obfuscation.

Organization. In this extended introduction, we provide brief descriptions of our techniques, including the construction from general assumptions, the BGN construction, and the Paillier construction. The rest of the chapter delves into technical detail. Section 6.2 reviews some preliminary notation and the public-key obfuscation definition. Section 6.3 explores the construction from general assumptions, while sections 6.4 and 6.5 describe the BGN and Paillier shuffles, respectively. Section 6.6 shows efficient protocols for proving the correctness of these shuffle programs, while section 6.7 shows how a number of mutually distrusting parties can jointly generate a shuffle program. Section 6.9 describe the efficiency of these schemes, and Section 6.10 details the proof that these constructions realize a UC mixnet. Finally, section 6.11 briefly explores how we could have obfuscated these programs using the original model of Barak et al.

6.1.2 Notation

As defined in Chapter 2, we use $\oplus$ to denote homomorphic addition, $\otimes$ to denote homomorphic multiplication, and exponent notation to denote a constant number of repeated homomorphic operations. Thus, for additive homomorphic cryptosystems, exponent notation indicates homomorphic multiplication by a constant, while, for multiplicative homomorphic cryptosystems, it denotes homomorphic exponentiation by a constant.

We denote Σ_N the set of permutations of N elements, and $\Lambda^\pi = (\lambda_{ij}^\pi)$ the column-representation permutation matrix for permutation π. In other words, $\lambda_{ij}^\pi = 1$ if $j = \pi(i)$, and 0 otherwise. This allows our notation of vector-by-matrix multiplication to be quite natural: $v \times M$, without transpose notation.

We denote by κ_c and κ_r additional security parameters such that $2^{-\kappa_c}$ and $2^{-\kappa_r}$ are negligible, which determines the bit-size of challenges and random paddings in our protocols.

6.1.3 Overview of Techniques

The protocols presented here achieve homomorphic multiplication with a permutation matrix, followed by a provable threshold decryption step typical of mixnets. Roughly, the semantic security of the encryption scheme hides the permutation. We begin here with the broad strokes of our constructions to provide an intuitive understanding of the protocols. The generic protocol is represented in Figure 6-1.

Generic Construction. Consider two semantically-secure cryptosystems, $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ and $\mathcal{CS}' = (\mathcal{G}', \mathcal{E}', \mathcal{D}')$, where $\mathcal{CS}'$ is additively homomorphic and the plaintext space of $\mathcal{CS}'$ can accommodate any ciphertext from $\mathcal{CS}$. We use $\oplus$ to denote the homomorphic addition, and exponent notation to denote homomorphic multiplication by a constant. A few highly interesting properties emerge:

$$\begin{aligned}
\mathcal{E}'_{pk'}(1)^{\mathcal{E}_{pk}(m)} &= \mathcal{E}'_{pk'}(1 \cdot \mathcal{E}_{pk}(m)) = \mathcal{E}'_{pk'}(\mathcal{E}_{pk}(m)) \\
\mathcal{E}'_{pk'}(0)^{\mathcal{E}_{pk}(m)} &= \mathcal{E}'_{pk'}(0 \cdot \mathcal{E}_{pk}(m)) = \mathcal{E}'_{pk'}(0) \\
\mathcal{E}'_{pk'}(0) \oplus \mathcal{E}'_{pk'}(\mathcal{E}_{pk}(m)) &= \mathcal{E}'_{pk'}(\mathcal{E}_{pk}(m+0)) = \mathcal{E}'_{pk'}(\mathcal{E}_{pk}(m)) \ .
\end{aligned}$$

Consider $\tilde{\Lambda}^\pi$, the element-wise encryption of a permutation matrix $\Lambda^\pi = (\lambda^\pi_{ij})$ under $\mathcal{E}'$:

$$(\tilde{\lambda}^\pi_{ij}) \stackrel{\text{def}}{=} \left(\mathcal{E}'_{pk'}(\lambda^\pi_{ij})\right)$$

Consider inputs (c_j) to a shuffle as ciphertexts under $\mathcal{E}$:

$$(c_j) \stackrel{\text{def}}{=} \left(\mathcal{E}_{pk}(m_j)\right)$$

Homomorphic matrix multiplication can then be performed using the algebraic properties noted above for multiplication and addition. Note that this construction works because the homomorphic addition expects *no more than one non-zero* element to add. In other words, this techniques works only on a permutation matrix:

$$\begin{aligned}
(c'_j) &\stackrel{\text{def}}{=} \bigoplus_{i=1}^{N} (\tilde{\lambda}^\pi_{ij})^{c_i} \\
&= \bigoplus_{i=1}^{N} \left(\mathcal{E}'_{pk'}(\lambda^\pi_{ij} c_i)\right) \\
&= \left(\mathcal{E}'_{pk'}(c_{\pi(j)})\right) \\
&= \left(\mathcal{E}'_{pk'}\left(\mathcal{E}_{pk}(m_{\pi(j)})\right)\right)
\end{aligned}$$

The shuffled result is doubly encrypted, and needs sequential decryption using $\mathcal{D}'_{sk'}$ and $\mathcal{D}_{sk}$. Unfortunately, in a mixnet construction, decryption must be verifiable, and double-decryption is particularly inefficient to prove without revealing the intermediate ciphertext. Of course, revealing the intermediate ciphertext $\mathcal{E}_{pk}(m_j)$ is not an option, as it exactly matches the corresponding input and immediately leaks the permutation. As a result, the only construction based on generic assumptions requires generic zero-knowledge proofs, which are prohibitively expensive.

To solve this problem, we propose two specific constructions. The BGN-based construction enables homomorphic matrix multiplication with singly-encrypted results. The Paillier-based constrution enables homomorphic matrix multiplication with reencryption on the inner ciphertext, which allows us to reveal the intermediate ciphertext at decryption time for a much more efficient proof.

$$\underset{\Lambda^{\pi}}{\begin{bmatrix} \lambda_{1,1} & \cdots & \lambda_{1,N} \\ \vdots & \ddots & \vdots \\ \lambda_{N,1} & \cdots & \lambda_{N,N} \end{bmatrix}} \xrightarrow{\boxed{\mathcal{E}}} \underset{\tilde{\Lambda}^{\pi}}{\begin{bmatrix} \tilde{\lambda}_{1,1} & \cdots & \tilde{\lambda}_{1,N} \\ \vdots & \ddots & \vdots \\ \tilde{\lambda}_{N,1} & \cdots & \tilde{\lambda}_{N,N} \end{bmatrix}}$$

$$\begin{bmatrix} c_1 & \cdots & c_N \end{bmatrix} \quad \otimes \quad \begin{bmatrix} \tilde{\lambda}_{1,1} & \cdots & \tilde{\lambda}_{1,N} \\ \vdots & \ddots & \vdots \\ \tilde{\lambda}_{N,1} & \cdots & \tilde{\lambda}_{N,N} \end{bmatrix} \quad = \quad \begin{bmatrix} c'_{\pi(1)} & \cdots & c'_{\pi(N)} \end{bmatrix}$$

Figure 6-1: Public Mixing: a permutation π is encoded in matrix form as Λ^{π}, then elementwise encrypted as $\tilde{\Lambda}^{\pi}$. Shuffling is accomplished by homomorphic matrix multiplication, which is implemented in different ways depending on the underlying cryptosystem.

BGN Construction. The BGN cryptosystem offers a normal additive homomorphism and a *one-time multiplicative homomorphism.* If ciphertexts are homomorphically multiplied, they are mapped into a different group with its own encryption and decryption algorithms using the same keypair. Thus, verifiability remains a single-layer proof. The specific homomorphic properties are thus:

$$\begin{aligned} \mathcal{E}_{pk}(m_1) \otimes \mathcal{E}_{pk}(m_1) &= \mathcal{E}'_{pk}(m_1 m_2) \\ \mathcal{E}_{pk}(m_1) \oplus \mathcal{E}_{pk}(m_2) &= \mathcal{E}_{pk}(m_1 + m_2) \\ \mathcal{E}'_{pk}(m_1) \oplus \mathcal{E}'_{pk}(m_2) &= \mathcal{E}'_{pk}(m_1 + m_2). \end{aligned}$$

Here, both the matrix and the inputs can be encrypted using the *same* encryption algorithm and public key:

$$(\tilde{\lambda}^{\pi}_{ij}) \stackrel{\text{def}}{=} (\mathcal{E}_{pk}(\lambda^{\pi}_{ij}))$$

$$(c_j) \stackrel{\text{def}}{=} (\mathcal{E}_{pk}(m_j))$$

and the matrix multiplication uses both homomorphisms, noting that only one multiplication is required on the arithmetic path from one input to one output:

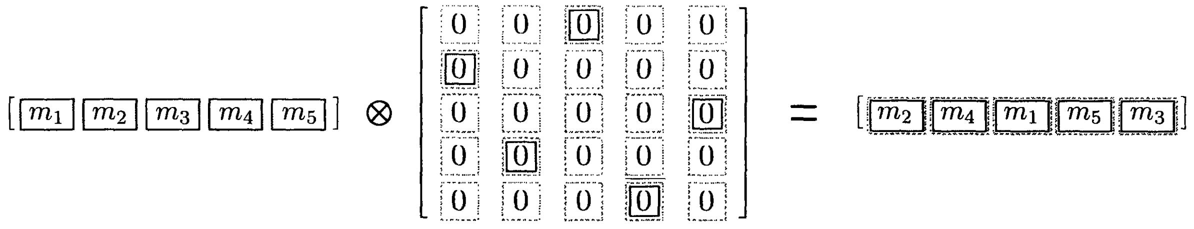


Figure 6-2: Paillier Shuffle: Two layers of encryption, an outer layer shown in orange, and an inner layer shown in blue, are used to provide mixing. The inputs to be shuffled are encrypted using the inner layer only. The 0's of the permutation matrix are encrypted using the outer layer only. The 1's of the permutation matrix are encrypted as double-layer encryptions of 0. The resulting ciphertexts are also double-layer encryptions of the now-shuffled plaintexts. Not diagrammed here is the fact that the inner-layer of the input ciphertexts is reencrypted by the homomorphic matrix multiplication (intuitively, by the inner-layer encryption of the double-encrypted zeros.)

$$\begin{aligned}
(c'_j) &\overset{\text{def}}{=} \bigoplus_{i=1}^{N} (\tilde{\lambda}^{\pi}_{ij}) \otimes c_i \\
&= \bigoplus_{i=1}^{N} \left(\mathcal{E}_{pk}(\lambda^{\pi}_{ij}) \otimes \mathcal{E}_{pk}(m_i)\right) \\
&= \bigoplus_{i=1}^{N} \left(\mathcal{E}'_{pk'}(\lambda^{\pi}_{ij} m_i)\right) \\
&= \left(\mathcal{E}'_{pk'}(m_{\pi(j)})\right)
\end{aligned}$$

This list of singly-encrypted ciphertexts under $\mathcal{E}'$ lends itself to efficient, provable decryption.

Paillier Construction. The Paillier cryptosystem supports layered encryption, where a ciphertext can be encrypted again using the same public key. Most interestingly, the homomorphic properties are preserved in the inner layer; in addition to the generic layered homomorphic properties we have the special relation

$$\begin{aligned}
\mathcal{E}'_{pk}\left(\mathcal{E}_{pk}(0;r)\right)^{\mathcal{E}_{pk}(m;s)} &= \mathcal{E}'_{pk}\left(\mathcal{E}_{pk}(0;r)\mathcal{E}_{pk}(m;s)\right) \\
&= \mathcal{E}'_{pk}\left(\mathcal{E}_{pk}(m;r+s)\right).
\end{aligned}$$

Thus, we can use $\mathcal{E}'$ encryption for the permutation matrix, except that, instead of $\mathcal{E}'_{pk}(1)$ to represent a one, we use $\mathcal{E}'_{pk}(\mathcal{E}_{pk}(0,r))$ with a random r:

$$(\tilde{\lambda}^{\pi}_{ij}) = \left(\mathcal{E}'_{pk}\left(\lambda^{\pi}_{ij}\mathcal{E}_{pk}(0; r_{ij})\right)\right)$$

The inputs are normal encryptions under $\mathcal{E}$:

$$(c_j) \overset{\text{def}}{=} \left(\mathcal{E}_{pk}(m_j; s_j)\right)$$

And the matrix multiplication is similar to the general assumption construction:

$$\begin{aligned}
(c'_j) &\overset{\text{def}}{=} \bigoplus_{i=1}^{N}(\tilde{\lambda}^{\pi}_{ij})^{c_i} \\
&= \bigoplus_{i=1}^{N}\left(\mathcal{E}'_{pk'}(\lambda^{\pi}_{ij}\mathcal{E}_{pk}(0; r_{ij})c_i)\right) \\
&= \left(\mathcal{E}'_{pk'}(\mathcal{E}_{pk}(0; r_{\pi(j),j})c_{\pi(j)})\right) \\
&= \left(\mathcal{E}'_{pk'}\left(\mathcal{E}_{pk}(m_{\pi(j)}; s_{\pi(j)} + r_{\pi(j),j})\right)\right)
\end{aligned}$$

Notice how the input ciphertext $\mathcal{E}_{pk}(m_j)$ is effectively reencrypted by the encryption of 0 already wrapped inside the outer-encryption. Thus, decryption can reveal this reencrypted intermediate ciphertext safely, and prove decryption in two efficient steps. The Paillier mixing by homomorphic matrix multiplication is represented in Figure 6-2.

6.1.4 Sizing the Matrix and Filling in Empty Slots

In a typical election setting, and in many other applications of public mixing, one may not know ahead of time the exact number N of inputs that will need to be shuffled. In these situations, an obfuscated shuffle program must be created with N as the upper bound of what can be reasonably expected. In the case of voting, a precinct would need a shuffle program sized for the number of eligible voters.

Then, when it comes time to mix $N' < N$ inputs, the $N - N'$ empty slots are filled in with the trivial encryption of a null message, where the null plaintext is agreed upon ahead of time. These null messages will be recognized upon decryption and discarded. Of course, as everyone will know which inputs and which outputs were null, some information regarding the original permutation will be revealed. However, the rest of the permutation—the portion that actually matters— remains completely secret.

6.2 Preliminaries

In this chapter, we make significant use of homomorphic cryptosystems, as defined in Chapter 2. We review here some additional concepts that are specific to the contributions of this chapter.

6.2.1 Functionalities

We consider sets of of circuits that perform a type of operation. These circuits are grouped into sets according to security parameter, and these sets are grouped into a single family of all circuits for the given "functionality."

Definition 6-1 (Functionality) *A functionality is a family $\mathcal{F} = \{\mathcal{F}_\kappa\}_{\kappa \in \mathrm{N}}$ of sets of circuits such that there exists a polynomial $s(\cdot)$ such that $|F| \leq s(\kappa)$ for every $F \in \mathcal{F}_\kappa$.*

Merely defining a functionality's existence is not enough to provide constructions. A functionality must also be sampleable, meaning that there exists a *PPT* sampling algorithm for constructing an instance of such a functionality with the appropriate security parameter.

Definition 6-2 (Sampleable Functionality) *A functionality is sampleable if there exists $\mathcal{S} \in PPT$ such that $\mathcal{S}(1^k)$ outputs $F \in \mathcal{F}_\kappa$, and the distribution of $\mathcal{S}(1^k)$ is uniform over $\mathcal{F}_\kappa$.*

6.2.2 Public-Key Obfuscation

We use a variation of the definition of public-key obfuscation of Ostrovsky and Skeith [132]. Our definition differs in that the functionality takes the public and secret keys as input. We note, of course, that a reasonable obfuscator will not hard-wire any secret-key information into its output program, as this would hardly be secure. However, we consider that the secret key may be necessary to create the obfuscated program.

Intuitively, a public-key obfuscator produces a program of reasonable size whose function is identical to the original functionality, except the outputs are encrypted. For security, the obfuscator should offer a kind of indistinguishability: two obfuscations of different functionalities should be indistinguishable. We now formalize this intuition.

Definition 6-3 (Public-Key Obfuscator) *An algorithm $\mathcal{O} \in PPT$ is a public-key obfuscator for a functionality $\mathcal{F}$ with respect to a cryptosystem $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ if there exists a decryption algorithm $\mathcal{D}' \in \mathrm{PT}$ and a polynomial $s(\cdot)$ such that for every $\kappa \in \mathbb{N}$, $F \in \mathcal{F}_\kappa$, $(pk, sk) \in \mathcal{G}(1^\kappa)$, and $x \in \{0,1\}^*$,*

1. *Correctness.* $\mathcal{D}'_{sk}(\mathcal{O}(1^\kappa, pk, sk, F)(x)) = F(pk, sk, x)$.
2. *Polynomial blow-up.* $|\mathcal{O}(1^\kappa, pk, sk, F)| \leq s(|F|)$.

Example 1 *Suppose we want to obfuscate the functionality that "multiplies by a." Select $\mathcal{CS}$, a multiplicative homomorphic cryptosystem and generate $(pk, sk) \in \mathcal{G}(1^\kappa)$ such that $a \in \mathsf{M}_{pk}$. The functionality is simply $F_a(pk, sk, x) = ax$, where $x \in \mathsf{M}_{pk}$. An obfuscator for the functionality $\mathcal{F}$ of such circuits is simply a circuit containing the hard-coded ciphertext $\mathcal{E}_{pk}(a)$ which, on input $x \in \mathsf{M}_{pk}$, outputs $\mathcal{E}_{pk}(a)^x = \mathcal{E}_{pk}(ax)$.*

Remark 6-1 *The above definition considers decryption algorithm $\mathcal{D}' \neq \mathcal{D}$ to accommodate obfuscators which require a slightly different decryption algorithm than the one that would decrypt the inputs to the functionality. Consider, specifically, the BGN multiplicative homomorphism, which takes encryptions under $\mathcal{E}$ and produces encryptions under $\mathcal{E}'$.*

We now consider the security property desired from an obfuscator, in exactly the same way as Ostrovsky and Skeith. First, we define an indistinguishability experiment much like that of semantic security, where the adversary selects two functionalities from the prescribed family and is challenged with the obfuscation of one of them. If no adversary can distinguish between the experiment that obfuscates the first functionality and the experiment that obfuscates the second functionality, the obfuscator is said to be polynomially indistinguishable.

Experiment 2 (Indistinguishability, $\mathsf{Exp}^{\mathsf{oind}-b}_{\mathcal{F},\mathcal{CS},\mathcal{O},\mathsf{Adv}}(\kappa)$)

$$\begin{array}{rcl} (pk, sk) & \leftarrow & \mathcal{G}(1^\kappa) \\ (F_0, F_1, \mathsf{state}) & \leftarrow & \mathsf{Adv}(\mathsf{choose}, pk), \\ d & \leftarrow & \mathsf{Adv}(\mathcal{O}(1^\kappa, pk, sk, F_b), \mathsf{state}) \end{array}$$

If $F_0, F_1 \in \mathcal{F}_\kappa$ return d, otherwise 0.

Definition 6-4 (Indistinguishability) *A public-key obfuscator $\mathcal{O}$ for a functionality $\mathcal{F}$ with respect to a cryptosystem $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ is polynomially indistinguishable if there exists a negligible function $\nu(\cdot)$ such that:*

$$|\Pr[\mathsf{Exp}^{\mathsf{oind}-0}_{\mathcal{F},\mathcal{CS},\mathcal{O},\mathsf{Adv}}(\kappa) = 1] - \Pr[\mathsf{Exp}^{\mathsf{oind}-1}_{\mathcal{F},\mathcal{CS},\mathcal{O},\mathsf{Adv}}(\kappa) = 1]| < \nu(\kappa)$$

The obfuscator in Example 1 is polynomially indistinguishable if $\mathcal{CS}$ is semantically secure.

6.2.3 Shuffles

We now review the two major types of shuffle: decryption and reencryption shuffles. For our purposes in this chapter, we formulate these as functionalities that we intend to obfuscate.

The most basic form of a shuffle is the decryption shuffle, as first introduced by Chaum [39]. A decryption shuffle takes a list of ciphertexts, decrypts them and outputs them in permuted order. Thus, given an input size N, the decryption shuffle is indexed by its permutation π.

Definition 6-5 (Decryption Shuffle) *A $\mathcal{CS}$-*decryption shuffle*, for a cryptosystem $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ is a functionality $DS_N = \{DS_{N(\kappa),\kappa}\}_{\kappa \in \mathbb{N}}$, where $N(\kappa)$ is a polynomially bounded and polynomially computable function such that for every $\kappa \in \mathbb{N}$, $DS_{N(\kappa),\kappa} = \{DS_\pi\}_{\pi \in \Sigma_{N(\kappa)}}$, and for every $(pk, sk) \in \mathcal{G}(1^\kappa)$, and $c_1, \ldots, c_{N(\kappa)} \in \mathsf{C}_{pk}$ the circuit DS_π is defined by*

$$DS_\pi(pk, sk, (c_1, \ldots, c_{N(\kappa)})) = (\mathcal{D}_{sk}(c_{\pi(1)}), \ldots, \mathcal{D}_{sk}(c_{\pi(N(\kappa))})) \ .$$

Another common way to implement a mixnet is to use the re-encryption-permutation paradigm introduced by Park et al. [134]. Using this approach, the ciphertexts are first re-encrypted and permuted, and then decrypted. In this definition, we capture the actions of the re-encryption and permutation phase. Given an input size N, a reencryption shuffle is indexed by its permutation π and a set of N randomization values $r_1, r_2, \ldots, r_N$. As the public key pk is not part of the functionality definition, the $\{r_i\}$ are sampled as bit strings and later mapped into R_{pk}.

Definition 6-6 (Re-encryption Shuffle) *A $\mathcal{CS}$-*re-encryption shuffle*, for a homomorphic cryptosystem $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ is a functionality $RS_N = \{RS_{N(\kappa),\kappa}\}_{\kappa\in\mathbb{N}}$, where $N(\kappa)$ is a polynomially bounded and polynomially computable function such that for every $\kappa \in \mathbb{N}$, $RS_{N(\kappa),\kappa} = \{RS_{\pi,r}\}_{\pi\in\Sigma_{N(\kappa)}, r\in(\{0,1\}^*)^{N(\kappa)}}$, and for every $(pk, sk) \in \mathcal{G}(1^\kappa)$, and $c_1, \ldots, c_{N(\kappa)} \in \mathsf{C}_{pk}$ the circuit $RS_{\pi,r}$ is defined by*

$$RS_\pi(pk, sk, (c_1, \ldots, c_{N(\kappa)})) = (\mathcal{RE}_{pk}(c_{\pi(1)}; r_1), \ldots, \mathcal{RE}_{pk}(c_{\pi(N(\kappa))}; r_{N(\kappa)})) \ .$$

6.3 Constructing and Obfuscating a Generic Decryption Shuffle

We show that, in principle, all that is needed to achieve obfuscated shuffling is an additively homomorphic cryptosystem. Consider two semantically-secure cryptosystems, $\mathcal{CS} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ and $\mathcal{CS}' = (\mathcal{G}', \mathcal{E}', \mathcal{D}')$, with $\mathcal{CS}'$ exhibiting an additive homomorphism denoted $\oplus$. Suppose that ciphertexts from $\mathcal{CS}$ can be encrypted under $\mathcal{CS}'$ for all $(pk, sk) \in \mathcal{G}(1^\kappa)$ and $(pk', sk') \in \mathcal{G}'(1^\kappa)$, i.e., $\mathsf{C}_{pk} \subseteq \mathsf{M}'_{pk'}$. The following operations are then possible and, more interestingly, indistinguishable thanks to the semantic security of both cryptosystems:

$$\begin{aligned} \mathcal{E}'_{pk'}(1)^{\mathcal{E}_{pk}(m)} &= \mathcal{E}'_{pk'}(\mathcal{E}_{pk}(m)) \\ \mathcal{E}'_{pk'}(0)^{\mathcal{E}_{pk}(m)} &= \mathcal{E}'_{pk'}(0). \end{aligned}$$

6.3.1 The Obfuscator

Consider a permutation matrix $\Lambda^\pi = (\lambda^\pi_{ij})$ corresponding to a permutation π. Consider its element-wise encryption under $\mathcal{CS}'$ with public key pk' and a corresponding matrix of randomization values $(r_{ij}) \in \mathsf{R}'^{N^2}_{pk'}$, i.e., $\tilde{\Lambda}^\pi = (\mathcal{E}'_{pk'}(\lambda^\pi_{ij}, r_{ij}))$. Then, given $\mathbf{c} = (c_1, c_2, \ldots, c_N) \in \mathsf{C}^N_{pk}$, it is possible to perform homomorphic matrix multiplication with a permutation matrix as:

$$\begin{aligned}\mathbf{c} \star \tilde{\Lambda}^{\pi} &\overset{\text{def}}{=} \left(\bigoplus_{i=1}^{N} (\tilde{\lambda}_{ij}^{\pi})^{c_i} \right) \\ &= \left(\bigoplus_{i=1}^{N} \left(\mathcal{E}'_{pk'}(c_i \lambda_{ij}^{\pi}) \right) \right) \\ &= \left(\mathcal{E}'_{pk'}(c_{\pi(j)}) \right) \\ &= \left(\mathcal{E}'_{pk'} \left(\mathcal{E}_{pk}(m_{\pi(j)}) \right) \right)\end{aligned}$$

Thus, we have:

$$\mathcal{D}_{sk}(\mathcal{D}'_{sk'}(\mathbf{c} \star \tilde{\Lambda}^{\pi})) = (m_{\pi(1)}, m_{\pi(2)}, \ldots, m_{\pi(N)}) \ .$$

We now define an obfuscator based on this homomorphic matrix multiplication. As the two layers of decryption must be performed at once by the mixnet, the obfuscator's decryption algorithm must perform *both* decryption layers, and the outputs of the obfuscated functionality will be plaintexts. In other words, we are dealing here with the obfuscation of a decryption shuffle.

Definition 6-7 (Obfuscator) *The obfuscator $\mathcal{O}$ for the decryption shuffle DS_N:*

- *takes input $(1^{\kappa}, (pk, pk'), (sk, sk'), DS_{\pi})$, where $(pk, sk) \in \mathcal{G}(1^{\kappa})$, $(pk', sk') \in \mathcal{G}'(1^{\kappa})$ and $DS_{\pi} \in DS_{N(\kappa),\kappa}$,*
- *computes $\tilde{\Lambda}^{\pi} = \mathcal{E}'_{pk'}(\Lambda^{\pi})$, and*
- *outputs a circuit with the hard-coded encrypted matrix $\tilde{\Lambda}^{\pi}$ and which, on input $\mathbf{c} = (c_1, \ldots, c_{N(\kappa)})$ computes and outputs $\mathbf{c}' = \mathbf{c} \star \tilde{\Lambda}^{\pi}$.*

Technically, this is a decryption shuffle of a new cryptosystem $\mathcal{CS}'' = (\mathcal{G}'', \mathcal{E}, \mathcal{D})$, where $\mathcal{CS}''$ executes the original key generators and outputs $((pk, pk'), (sk, sk'))$ and the original algorithms $\mathcal{E}$ and $\mathcal{D}$ simply ignore (pk', sk'). We give a reduction without any loss in security for the following straight-forward proposition.

Proposition 6-1 *If $\mathcal{CS}'$ is semantically secure then $\mathcal{O}$ is polynomially indistinguishable.*

Proof. We prove this by contradiction.

Assume the existence of Adv, a successful arbitrary against the polynomial indistinguishability experiment with obfuscator $\mathcal{O}$. We now construct an adversary Adv$'$ which defeats the semantic security of the cryptosystem $\mathcal{CS}'$ using Adv as a black box. In this description, "the experiment" refers to the semantic security experiment which Adv$'$ is trying to defeat. Meanwhile, Adv$'$ simulates the $\mathcal{O}$ indistinguishability experiment to Adv:

- Adv' receives pk from the experiment and forwards it to Adv.

- Adv outputs two shuffles, DS_{π_0} and DS_{π_1}, which imply permutation matrices Λ^{π_0} and Λ^{π_1}; Adv' outputs plaintexts 0 and 1 to the experiment.

- on input from the experiment $c = \mathcal{E}'_{pk}(b)$, Adv' prepares $\bar{c} = \mathcal{E}'_{pk}(\bar{b})$ homomorphically. Adv' then constructs an encrypted matrix $\tilde{\Lambda}^{\pi_2}$ as follows:

  - for values of i, j where $\Lambda^{\pi_0}_{ij} = \Lambda^{\pi_1}_{ij} = b_{ij}$, $\tilde{\Lambda}^{\pi_2}_{ij} = \mathcal{E}'_{pk}(b_{ij})$.
  - for values of i, j where $\Lambda^{\pi_0}_{ij} \neq \Lambda^{\pi_1}_{ij}$,
    $\tilde{\Lambda}^{\pi_2}_{ij} = \mathcal{RE}_{pk}(c)$ when $\Lambda^{\pi_0}_{ij} = 0$, and
    $\tilde{\Lambda}^{\pi_2}_{ij} = \mathcal{RE}_{pk}(\bar{c})$ when $\Lambda^{\pi_0}_{ij} = 1$.
  - thus, $\tilde{\Lambda}^{\pi_2}$ is the encryption of Λ^{π_b}.

  Adv' then provides $\tilde{\Lambda}^{\pi_2}$ as the simulated challenge to Adv.

- on response b' from Adv, Adv' outputs guess b' to the experiment.

Then, by construction, Adv' succeeds in distinguishing the two semantic security experiments with the exact same probability that Adv succeeds in breaking the indistinguishability of the obfuscator. By contradiction, the proposition follows.

□

6.3.2 Limitations of the Generic Construction

An encrypted matrix is not necessarily an encrypted *permutation* matrix. Before it can be used, $\tilde{\Lambda}^{\pi}$ requires a proof that it is indeed the encryption of a permutation matrix. In this generic construction, this can be accomplished by proving that each element is the encryption of either 0 or 1, and that the homomorphic column- and row-wise sums are encryptions of 1, using a Chaum-Pedersen-like proof of plaintext [37] and Cramer et al.'s proof of partial knowledge [44]. Unfortunately, this approach requires $O(N^2)$ *proofs.*

Even if we prove that $\tilde{\Lambda}^{\pi}$ is correctly formed, the post-shuffle verifiable decryption of $\mathcal{E}'_{pk'}(\mathcal{E}_{pk}(m_i))$ to m_i is prohibitively expensive: revealing the inner ciphertext is out of the question, as it would leak the shuffle permutation, which leaves us only with generic proof techniques. Instead of exploring this area further, we turn to vastly more promising efficient constructions.

6.4 Obfuscating a BGN Decryption Shuffle

We now show how to obfuscate a decryption shuffle for the Boneh-Goh-Nissim (BGN) cryptosystem [23] by exploiting both its additive homomorphism and its one-time multiplicative homomorphism.

6.4.1 The BGN Cryptosystem

Denote the BGN cryptotystem as $\mathcal{CS}^{\mathsf{bgn}} = (\mathcal{G}^{\mathsf{bgn}}, \mathcal{E}^{\mathsf{bgn}}, \mathcal{D}^{\mathsf{bgn}})$. BGN operates in two groups $\mathbb{G}_1$ and $\mathbb{G}_2$, both of order $n = q_1 q_2$, where q_1 and q_2 are distinct prime integers. We use multiplicative notation in both $\mathbb{G}_1$ and $\mathbb{G}_2$, and denote g a generator in $\mathbb{G}_1$. The groups $\mathbb{G}_1$ and $\mathbb{G}_2$ exhibit a polynomial-time computable bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \longrightarrow \mathbb{G}_2$ such that $e(g, g)$ generates $\mathbb{G}_2$. Bilinearity implies that $\forall u, v \in \mathbb{G}_1$ and $\forall a, b \in \mathbf{Z}, e\left(u^a, v^b\right) = e(u, v)^{ab}$.

Key generation. On input 1^κ, $\mathcal{G}^{\mathsf{bgn}}$ generates parameters $(q_1, q_2, \mathbb{G}_1, g, \mathbb{G}_2, e(\cdot,\cdot))$ as above such that $n = q_1 q_2$ is a κ-bit integer. It chooses $u \in \mathbb{G}_1$ randomly, defines $h = u^{q_2}$, and outputs a public key $pk = (n, \mathbb{G}_1, \mathbb{G}_2, e(\cdot,\cdot), g, h)$ and secret key $sk = q_1$.

Encryption. On input pk and $m \in \mathbf{Z}_n$, $\mathcal{E}^{\mathsf{bgn}}$ selects $r \xleftarrow{R} \mathbf{Z}_n$ and outputs $c = g^m h^r \in \mathbb{G}_1$.

Decryption. On input $sk = q_1$ and $c \in \mathbb{G}_1$, $\mathcal{D}^{\mathsf{bgn}}$ outputs $m' = \log_{g^{q_1}}(c^{q_1})$.

Because decryption computes a discrete logarithm, the plaintext space must be restricted considerably. Corresponding algorithms $\mathcal{E}'^{\mathsf{bgn}}$ and $\mathcal{D}'^{\mathsf{bgn}}$ perform encryption and decryption in $\mathbb{G}_2$ using the generators $G = e(g, g)$ and $H = e(g, h)$. The BGN cryptosystem is semantically secure under the *Subgroup Decision Assumption*, which states that no $\mathsf{Adv} \in \mathrm{PT}^*$ can distinguish between a uniform distribution on $\mathbb{G}_1$ and a uniform distribution on the unique order q_1 subgroup in $\mathbb{G}_1$.

Homomorphisms. The BGN cryptosystem is additively homomorphic:

$$\begin{aligned}
c_1 \oplus c_2 &\stackrel{\text{def}}{=} c_1 c_2 \\
&= \mathcal{E}^{\mathsf{bgn}}_{pk}(m_1; r_1)\mathcal{E}^{\mathsf{bgn}}_{pk}(m_2; r_2) \\
&= g^{m_1} h^{r_1} g^{m_2} h^{r_2} \\
&= g^{m_1+m_2} h^{r_1+r_2} \\
&= \mathcal{E}^{\mathsf{bgn}}_{pk}(m_1 + m_2; r_1 + r_2)
\end{aligned}$$

In addition, the BGN cryptosystem offers a multiplicative homomorphism for ciphertexts in $\mathbb{G}_1$, using the bilinear map. We denote $\alpha = \log_g u$ where u is the base chosen during key generation:

$$
\begin{aligned}
c_1 \otimes c_2 &\overset{\text{def}}{=} e\left(c_1, c_2\right) \\
&= e\left(\mathcal{E}^{\mathsf{bgn}}_{pk}(m_1; r_1), \mathcal{E}^{\mathsf{bgn}}_{pk}(m_2; r_2)\right) \\
&= e\left(g^{m_1} h^{r_1}, g^{m_2} h^{r_2}\right) \\
&= e\left(g, g\right)^{m_1 m_2} e\left(g, h\right)^{m_1 r_2 + m_2 r_1} e\left(h, h\right)^{r_1 r_2} \\
&= G^{m_1 m_2} H^{m_1 r_2 + m_2 r_1} \left(H^{\alpha q_2}\right)^{r_1 r_2} \\
&= \mathcal{E}'^{\mathsf{bgn}}_{pk}(m_1 m_2; m_1 r_2 + m_2 r_1 + \alpha q_2 r_1 r_2)
\end{aligned}
$$

The result is a ciphertext in $\mathbb{G}_2$ which cannot be efficiently converted back to an equivalent ciphertext in $\mathbb{G}_1$, under the Bilinear One-Way Assumption, which derives from the DDH assumption on $\mathbb{G}_2$. Thus, the multiplicative homomorphism can be evaluated only once, after which only homomorphic additions are possible. Thus, we also define $c_1 \oplus c_2 \overset{\text{def}}{=} c_1 c_2$ for $c_1, c_2 \in \mathbb{G}_2^2$.

6.4.2 The Obfuscator

Our obfuscator is based on the observation that matrix multiplication only requires an arithmetic circuit with multiplication depth 1. Thus, the BGN cryptosystem can be used for homomorphic matrix multiplication. Consider a $N_1 \times N_2$-matrix $\tilde{A} = (\tilde{a}_{ij}) = (\mathcal{E}^{\mathsf{bgn}}_{pk}(a_{ij}))$ and a $N_2 \times N_3$-matrix $\tilde{B} = (\tilde{b}_{jk}) = (\mathcal{E}^{\mathsf{bgn}}_{pk}(b_{jk}))$, and let $A = (a_{ij})$ and $B = (b_{jk})$. We define homomorphic matrix multiplication by

$$\tilde{A} \star \tilde{B} \overset{\text{def}}{=} \left(\bigoplus_{j=1}^{N_2} \tilde{a}_{ij} \otimes \tilde{b}_{jk} \right)$$

and conclude that:

$$\mathcal{D}'^{\mathsf{bgn}}_{sk}(\tilde{A} \star \tilde{B}) = \left(\sum_{j=1}^{N_2} a_{ij} b_{jk} \right) = AB.$$

As in the general case, the resulting elements of this homomorphic matrix multiplication are meant to be decrypted each in one operation and proof – though in this case it really is one layer of decryption, rather than two layers that need to be unwrapped and proved as one!

Thus, we construct an obfuscator for a BGN-based decryption shuffle. This obfuscator simply samples the randomization values required to encrypt the permutation matrix, then hard-codes this matrix into the obfuscated circuit, so that it can perform the homomorphic matrix multiplication.

Definition 6-8 (Obfuscator) *The obfuscator $\mathcal{O}^{\mathsf{bgn}}$ for the decryption shuffle DS_N^{bgn} takes input $(1^\kappa, pk, sk, DS_\pi^{\mathsf{bgn}})$, where $(pk, sk) \in \mathcal{G}^{\mathsf{bgn}}(1^\kappa)$ and $DS_\pi^{\mathsf{bgn}} \in DS_{N(\kappa),\kappa}^{\mathsf{bgn}}$, computes $\tilde{\Lambda}^\pi = \mathcal{E}_{pk}^{\mathsf{bgn}}(\Lambda^\pi)$, and outputs a circuit with $\tilde{\Lambda}^\pi$ hard-coded such that, on input $\mathbf{c} = (c_1, \ldots, c_{N(\kappa)})$, it outputs $\mathbf{c}' = \mathbf{c} \star \tilde{\Lambda}^\pi$.*

We have the following corollary from Proposition 6-1.

Corollary 6-1 *The obfuscator $\mathcal{O}^{\mathsf{bgn}}$ for DS_N^{bgn} is polynomially indistinguishable if the BGN cryptosystem is semantically secure.*

6.5 Obfuscating a Paillier Re-encryption Shuffle

We now show how to obfuscate a re-encryption shuffle for the Paillier cryptosystem [133] by exploiting its additive homomorphism and its generalization introduced by Damgård et al. [48]. In doing so, we expose a previously unnoticed homomorphic property of this generalized Paillier construction. We note that this property may be useful for other attempts at obfuscation.

6.5.1 The Paillier Cryptosystem

We denote the Paillier cryptosystem $\mathcal{CS}^{\mathsf{pai}} = (\mathcal{G}^{\mathsf{pai}}, \mathcal{E}^{\mathsf{pai}}, \mathcal{D}^{\mathsf{pai}})$ and refer the reader to Chapter 2 which describes its setup. Recall that the Paillier cryptosystem is semantically secure under the *Decision Composite Residuosity Assumption.* A Paillier public key is (n, g) with plaintext space $\mathbf{Z}_n$ and ciphertext space $\mathbf{Z}_{n^2}$. Recall also the Paillier encryption operation: $c = \mathcal{E}_{pk}^{\mathsf{pai}}(m; r) = g^m r^n \bmod n^2$.

Generalized Paillier. Damgård et al. [48] generalize the Paillier cryptosystem, replacing computations modulo n^2 with computations modulo n^{s+1} and plaintext space $\mathbf{Z}_n$ with $\mathbf{Z}_{n^s}$. Damgård et al. prove that the semantic security of the generalized scheme follows from the semantic security of the original scheme for $s > 0$ polynomial in the security parameter, though we only exploit the cases $s = 1, 2$. We write $\mathcal{E}_{n^{s+1}}^{\mathsf{pai}}(m) = g^m r^{n^s} \bmod n^{s+1}$ for generalized encryption to make explicit the value of s used in a particular encryption. Similarly we write $\mathcal{D}_{p,s}^{\mathsf{pai}}(c)$ for the decryption algorithm and we use $\mathsf{M}_{n^{s+1}}$ and $\mathsf{C}_{n^{s+1}}$ to denote the corresponding message and ciphertext spaces.

Alternative Encryption. There are natural and well-known alternative encryption algorithms for the Paillier cryptosystem. It is easy to see that one can pick the random element $r \in \mathbf{Z}_{n^s}^*$ instead of in $\mathbf{Z}_n^*$. If h_s is a generator of the group of n^sth residues, then we may define encryption of a message $m \in \mathbf{Z}_{n^s}$ as $g^m h_s^r \bmod n^s$. This alternative form is particularly useful for practical optimizations in computing Paillier ciphertexts.

Homomorphisms. The Paillier cryptosystem is additively homomorphic. Furthermore, the recursive structure of the Paillier cryptosystem allows a ciphertext $\mathcal{E}^{\mathsf{pai}}_{n^2}(m) \in \mathsf{C}_{n^2} = \mathbf{Z}^*_{n^2}$ to be viewed as a plaintext in the group $\mathsf{M}_{n^3} = \mathbf{Z}_{n^2}$ that can be encrypted using a generalized version of the cryptosystem, i.e., we can compute $\mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(m)\right)$. Furthermore, *the nested cryptosystems preserve the group structures over which they are defined.* In other words we have

$$\begin{aligned}
\mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0,r)\right)^{\mathcal{E}^{\mathsf{pai}}_{n^2}(m,s)} &= \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0,r)\mathcal{E}^{\mathsf{pai}}_{n^2}(m,s)\right) \\
&= \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(m,r+s)\right) .
\end{aligned}$$

Note how this homomorphic operation is similar to the operations described under general assumptions, except the inner "1" has been replaced with an encryption of 0. As a result, though the output is also a double-encrypted m_i, a re-encryption has occurred on the inner ciphertext.

6.5.2 The Obfuscator

We use the additive homomorphism and the special homomorphic property exhibited above to define a form of homomorphic matrix multiplication for encrypted Paillier matrices. AS in the general case—and unlike the BGN construction—this matrix multiplication only works when one of the matrices is a permutation matrix.

Given an N-permutation matrix $\Lambda^\pi = (\lambda^\pi_{ij})$, and randomness $r, s \in (\mathbf{Z}^*_n)^{N\times N}$, we prepare the element-wise encryption of this matrix under $\mathcal{E}^{\mathsf{pai}}_{n^3}$, with $\mathcal{E}^{\mathsf{pai}}_{n^2}$ encryptions of 0 where 1's would appear:

$$\tilde{\Lambda}^\pi = (\tilde{\lambda}^\pi_{ij}) = \left(\mathcal{E}^{\mathsf{pai}}_{n^3}(\lambda^\pi_{ij}\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r_{ij});s_{ij})\right) .$$

Consider $\mathbf{c} = (c_1, \ldots, c_N)$ a vector of encrypted inputs under $\mathcal{E}^{\mathsf{pai}}_{n^2}$, such that $c_i = \mathcal{E}^{\mathsf{pai}}_{n^2}(m_i;t_i)$. The matrix multiplication operation is then defined using the above interesting operations:

$$\begin{aligned}
\mathbf{c} \star \tilde{\Lambda}^{\pi} &\overset{\text{def}}{=} \left(\bigoplus_{i=1}^{N} (\tilde{\lambda}^{\pi}_{ij})^{c_i} \right) \\
&= \left(\bigoplus_{i=1}^{N} \mathcal{E}^{\mathsf{pai}}_{n^3} \left(\lambda^{\pi}_{ij} \mathcal{E}^{\mathsf{pai}}_{n^2}(0; r_{ij}); s_{ij} \right)^{c_i} \right) \\
&= \left(\bigoplus_{i=1}^{N} \mathcal{E}^{\mathsf{pai}}_{n^3} \left(c_i \lambda^{\pi}_{ij} \mathcal{E}^{\mathsf{pai}}_{n^2}(0; r_{ij}); s_{ij}^{c_i} \right) \right) \\
&= \left(\mathcal{E}^{\mathsf{pai}}_{n^3} \left(c_{\pi(j)} \mathcal{E}^{\mathsf{pai}}_{n^2}(0; r_{\pi(j),j}); \prod_{i=1}^{N} s_{ij}^{c_i} \right) \right) \\
&= \left(\mathcal{E}^{\mathsf{pai}}_{n^3} \left(\mathcal{E}^{\mathsf{pai}}_{n^2}(m_{\pi(j)}; t_{\pi(j)} r_{\pi(j),j}); \prod_{i=1}^{N} s_{ij}^{c_i} \right) \right)
\end{aligned}$$

Noting that we have effectively reencrypted the inner ciphertext, this can be written in the following simpler form for clarity:

$$\mathbf{c} \star \tilde{\Lambda}^{\pi} = \left(\mathcal{E}^{\mathsf{pai}}_{n^3} \left(\mathcal{E}^{\mathsf{pai}}_{n^2}(m_{\pi(j)}) \right) \right)$$

Note that this immediately gives:

$$\mathcal{D}^{\mathsf{pai}}_{p,2}(\mathcal{D}^{\mathsf{pai}}_{p,3}(\mathbf{c} \star \tilde{\Lambda}^{\pi})) = (m_{\pi(1)}, \ldots, m_{\pi(N)}) \ .$$

In other words, we can do homomorphic matrix multiplication with a permutation matrix using layered Paillier, but we stress that the above matrix multiplication does *not* work for general matrices. We are now ready to define the obfuscator for the Paillier-based shuffle.

We note that there is a functional difference between this shuffle and the two prior constructions. Here, the decryption algorithm of the obfuscation process only needs to remove the outer layer of encryption, leaving ciphertext outputs under $\mathcal{E}^{\mathsf{pai}}_{n^2}$. The second decryption is technically not part of the obfuscation phase. Thus, we are effectively obfuscation a *reencryption* shuffle here. We note again, for clarity, that this approach is *not* feasible in the case of general assumptions, because revealing this intermediate ciphertext immediately leaks the permutation.

The obfuscator for this shuffle needs to sample N randomization factors for the inner encryptions of 0, which represent the 1's of the permutation matrix, and N^2 randomization factors for the outer encryptions of every matrix element.

Definition 6-9 (Obfuscator) *The obfuscator $\mathcal{O}^{\mathsf{pai}}$ for the re-encryption shuffle RS^{pai}_N takes input a tuple $(1^\kappa, n, sk, RS^{\mathsf{pai}})$, where $(n, p) \in \mathcal{G}^{\mathsf{pai}}(1^\kappa)$ and $RS^{\mathsf{pai}} \in RS^{\mathsf{pai}}_{N(\kappa),\kappa}$, computes*

$\tilde{\Lambda}^\pi = (\mathcal{E}^{\mathsf{pai}}_{n^3}(\lambda^\pi_{ij}\mathcal{E}^{\mathsf{pai}}_{n^2}(0, r_{ij}), s_{ij}))$, *and outputs a circuit with hardcoded* $\tilde{\Lambda}^\pi$ *that, on input* $\mathbf{c} = (c_1, \ldots, c_{N(\kappa)})$, *outputs* $\mathbf{c}' = \mathbf{c} \star \tilde{\Lambda}^\pi$.

Proposition 6-2 *The obfuscator* $\mathcal{O}^{\mathsf{pai}}$ *for* RS^{pai}_N *is polynomially indistinguishable if the Paillier cryptosystem is semantically secure.*

Proof.

This proof proceeds exactly like the proof of Proposition 6-1, except the two plaintexts submitted to the semantic security are 0 and some randomly selected encryption of 0 (instead of 1). As this proof is slightly trickier than the previous one, we present the details. We prove this by contradiction.

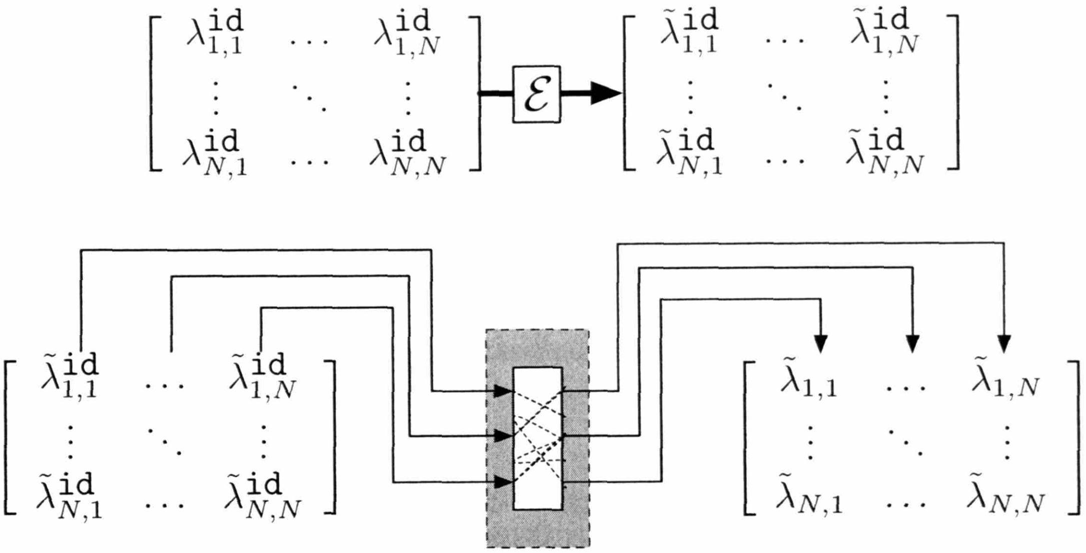
Assume the existence of Adv, a successful arbitrary against the polynomial indistinguishability experiment with obfuscator $\mathcal{O}^{\mathsf{pai}}$. We now construct an adversary Adv′ which defeats the semantic security of the cryptosystem $\mathcal{CS}^{\mathsf{pai}}$, specifically using $\mathcal{E}^{\mathsf{pai}}_{n^3}$, using Adv as a black box. In this description, "the experiment" refers to the semantic security experiment which Adv′ is trying to defeat. Meanwhile, Adv′ simulates the $\mathcal{O}^{\mathsf{pai}}$ indistinguishability experiment to Adv:

- Adv′ receives pk from the experiment and forwards it to Adv.

- Adv′ receives from Adv two shuffles, RS_{π_0} and RS_{π_1}, which imply permutation matrices Λ^{π_0} and Λ^{π_1}. Adv′ picks $r \in \mathsf{R}_{pk,n^2}$ and outputs messages $m_0 = 0$ and $m_1 = \mathcal{E}^{\mathsf{pai}}_{n^2}(0; r)$ to the experiment. Note how m_1 is a Paillier ciphertext, treated here as a plaintext (since Adv′ will attempt to distinguish ciphertexts under $\mathcal{E}^{\mathsf{pai}}_{n^3}$.)

- on input from the experiment $c = \mathcal{E}^{\mathsf{pai}}_{n^3}(m_b)$ for bit b, Adv′ prepares $\bar{c} = \mathcal{E}^{\mathsf{pai}}_{n^3}(m_{\bar{b}})$ homomorphically. Recall that $\mathcal{E}^{\mathsf{pai}}_{n^3}$ is additively homomorphic, so it is easy to homomorphically compute $m_1 - m_b$, which toggles the ciphertext between m_0 and m_1. Adv′ then constructs an encrypted matrix $\tilde{\Lambda}^{\pi_2}$ as follows:

  - for values of i, j where $\Lambda^{\pi_0}_{ij} = \Lambda^{\pi_1}_{ij} = 0$, $\tilde{\Lambda}^{\pi_2}_{ij} = \mathcal{E}^{\mathsf{pai}}_{n^3}(0)$ with fresh randomness.
  - for values of i, j where $\Lambda^{\pi_0}_{ij} = \Lambda^{\pi_1}_{ij} = 1$, $\tilde{\Lambda}^{\pi_2}_{ij} = \mathcal{E}^{\mathsf{pai}}_{n^3}(\mathcal{E}^{\mathsf{pai}}_{n^2}(0))$ with fresh randomness.
  - for values of i, j where $\Lambda^{\pi_0}_{ij} = 0$ but $\Lambda^{\pi_1}_{ij} = 1$, $\tilde{\Lambda}^{\pi_2}_{ij} = c^{\mathcal{E}^{\mathsf{pai}}_{n^2}(0)}\mathcal{E}^{\mathsf{pai}}_{n^3}(0)$ with fresh randomness.
  - for values of i, j where $\Lambda^{\pi_0}_{ij} = 1$ but $\Lambda^{\pi_1}_{ij} = 0$, $\tilde{\Lambda}^{\pi_2}_{ij} = \bar{c}^{\mathcal{E}^{\mathsf{pai}}_{n^2}(0)}\mathcal{E}^{\mathsf{pai}}_{n^3}(0)$ with fresh randomness.
  - thus, $\tilde{\Lambda}^{\pi_2}$ is the encryption of Λ^{π_b} ; note how we perform outer *and inner* reencryption on the ciphertext received from the experiment. This is crucial to simulate the obfuscation indistinguishability experiment to Adv, since, in the real experiment, the inner encryptions of 0 are all randomly selected, as are the randomization values for the outer encryptions.



$\lambda^{\mathtt{id}}_{1,1}$
$\lambda^{\mathtt{id}}_{1,N}$
$\lambda^{\mathtt{id}}_{N,1}$
$\lambda^{\mathtt{id}}_{N,N}$
$\mathcal{E}$
$\tilde{\lambda}^{\mathtt{id}}_{1,1}$
$\tilde{\lambda}^{\mathtt{id}}_{1,N}$
$\tilde{\lambda}^{\mathtt{id}}_{N,1}$
$\tilde{\lambda}^{\mathtt{id}}_{N,N}$
$\tilde{\lambda}_{1,1}$
$\tilde{\lambda}_{1,N}$
$\tilde{\lambda}_{N,1}$
$\tilde{\lambda}_{N,N}$

Figure 6-3: Proof of Correct Construction for a Single Prover: Starting with the trivial encryption of the identity matrix, the prover demonstrates knowledge of a permutation and randomization values in shuffling the columns of the matrix. This shows that the resulting matrix is indeed a permutation matrix.

Adv′ then provides $\tilde{\Lambda}^{\pi_2}$ as the simulated challenge to Adv.

- on response b' from Adv, Adv′ outputs guess b' to the experiment.

Then, by construction, Adv′ succeeds in distinguishing the two semantic security experiments with the exact same probability that Adv succeeds in breaking the indistinguishability of the obfuscator. By contradiction, the proposition follows.

□

6.6 Proving Correctness of Obfuscation

We now focus on proving the correctness of a BGN or Paillier obfuscated shuffle. We assume, for now, that a single party generates the encrypted matrix, though the techniques described here are immediately applicable to the distributed generation and proofs of the shuffles, which we detail in the next section.

6.6.1 Overview of the Proof Techniques

Shuffling Columns of a Matrix. We note the interesting property that, starting with a column-based permutation matrix Λ^{π}, shuffling the columns of the matrix according to a new permutation π' yields another permutation matrix, $\Lambda^{\pi' \circ \pi}$, corresponding to the composed permutation $\pi' \circ \pi$. In particular, a column shuffle of the identity matrix according to permutation π yields the column-based permutation matrix π. Intuitively, this is how we will prove that an encrypted matrix is a permutation matrix: by proving that it is the result of a column-shuffle of the identity matrix, as diagrammed in Figure 6-3.

More precisely, for either cryptosystem, start with a trivially encrypted identity matrix, $\tilde{\Lambda}^{\text{id}} = \mathcal{E}_{pk}(\Lambda^{\text{id}}, 0^*)$, and let the prover perform a zero-knowledge proof of knowledge of the permutation π and randomization values (r_{ij}) that map $\tilde{\Lambda}^{\text{id}}$ to $\tilde{\Lambda}^{\pi}$, the encrypted permutation whose correctness we want to prove. The formal relation for this proof is defined as follows.

Definition 6-10 *Denote by $\mathcal{R}_{mrp}$ the relation consisting of pairs $((1^\kappa, pk, \tilde{A}, \tilde{A}'), r)$ such that $\tilde{A} = (\tilde{a}_{ij}) \in \mathsf{C}_{pk}^{N\times N}$, $\tilde{A}' = (\tilde{a}'_{ij}) = \big(\mathcal{RE}_{pk}(a_{i,\pi(j)}, r_{ij})\big)$, $r \in \mathsf{R}_{pk}^{N\times N}$, and $\pi \in \Sigma_N$.*

Generating the Initial Identity Matrix. In the BGN shuffle, the starting identity matrix we seek can be simply $\tilde{A} = \tilde{\Lambda}^{\text{id}} = \mathcal{E}_{pk}(\Lambda^{\text{id}}, 0^*)$. The prover then shuffles and reencrypts the columns of $\tilde{\Lambda}^{\text{id}}$ and proves knowledge of a witness in $\mathcal{R}_{mrp}$.

Recall, however, that, where the BGN matrix is composed of encryptions of 1, the Paillier matrix contains outer encryptions of *different* inner encryptions of zero. These inner ciphertexts must remain secret from outside observer, as they are exactly the randomization values from the inputs to the outputs of the overall shuffle. If they leak, so does the shuffle permutation. Thus, in the Paillier case, we begin by generating and proving correct a list of N double-encryptions of zero. We construct a proof of double-discrete log with 1/2-soundness that must be repeated a number of times. This repetition remains "efficient enough" because we only need to perform a linear number of sets of repeated proofs. We then use these N double-encrypted zeros as the diagonal of our identity matrix, completing it with trivial outer encryptions of zero.

In both cases, once we have an acceptable encrypted identity matrix, we shuffle and re-encrypt its columns, and provide a zero-knowledge proof of knowledge of the permutation and re-encryption factors. The proof technique for column shuffling is essentially a batch proof [15] for shuffling N-tuples based on Neff's single-value shuffle proof [124].

6.6.2 Proving a Shuffle of the Columns of a Ciphertext Matrix

Consider the simpler and extensively studied problem of proving that a list of ciphertexts have been correctly re-encrypted and permuted, sometimes called a "proof of shuffle." The formal relation for this proof is as follows.

Definition 6-11 *Denote by $\mathcal{R}_{rp}$ the relation consisting of pairs $((1^\kappa, pk, \mathbf{c}, \mathbf{c}'), r)$ such that $\mathbf{c} = (c_i) \in \mathsf{C}_{pk}^{N}$ and $\mathbf{c}' = \mathcal{RE}_{pk}((c_{\pi(i)}), r)$ for some $r \in \mathsf{R}_{pk}^{N}$ and $\pi \in \Sigma_N$.*

There are several known efficient methods [124, 70, 87, 180] for constructing a protocol for this relation, all of which are reviewed in Chapter 3. Although these protocols differ slightly in their properties, they all essentially provide HVZK proofs of knowledge. Thus, we assume here, for clarity, that there exists an honest verifier zero-knowledge proof of knowledge π_{rp} for the above relation.

Such a protocol can then be extended to prove a shuffle of *lists of ciphertexts*, which is exactly what each matrix column is. We use the batch proof technique of Bellare et al. [15] for this purpose. Interestingly, this technique has not been applied to the mixnet setting to date. Thus, we provide a detailed description and proof in this setting.

Intuitively, we want to reduce each list of ciphertexts—effectively each column of the matrix—down to a single aggregate element. Then, we are left with N aggregate elements for the inputs and N aggregate elements for the outputs. We can then use π_{rp} to prove knowledge of the shuffle permutation and randomization values for these aggregate elements. Using a careful method for aggregation, we can show that a proof of shuffle on the aggregate elements is equivalent to a proof of shuffle on the entire lists. More specifically, the extractor for π_{rp} can be used to extract the entire matrix of randomization values.

The specific method of aggregation is a dot product operation with a random vector selected by the verifier. If the input and output matrices are set before this test vector is selected and if the output matrix is not a column-shuffle of the input matrix, then the probability that the output dot-products will equal *any* permutation of the input dot products is negligible. Neff [124] specifically addresses this issue (for different purposes) in his mixnet shuffle. We now formalize the details of this protocol and its proofs.

Protocol 6-1 (Matrix Reencryption Permutation)
COMMON INPUT. *public key pk and* $\tilde{A}, \tilde{A}' \in \mathsf{C}_{pk}^{N\times N}$
PRIVATE INPUT. $\pi \in \Sigma_N$ *and* $r \in \mathsf{R}_{pk}^{N\times N}$ *such that* $\tilde{A}' = \left(\mathcal{RE}_{pk}(\tilde{a}_{i,\pi(j)}; r_{ij})\right)$.

1. *$\mathcal{V}$ chooses* $\mathbf{u} = (u_i) \in [0, 2^{\kappa_c} - 1]^N$ *randomly and hands it to $\mathcal{P}$.*
2. *They both compute* $\mathbf{c} = (c_j) = \left(\bigoplus_{i=1}^N \tilde{a}_{ij}^{u_i}\right)$ *and* $\mathbf{c}' = (c'_j) = \left(\bigoplus_{i=1}^N (\tilde{a}'_{ij})^{u_i}\right)$.
3. *They run* π_{rp} *on common input* $(pk, \mathbf{c}, \mathbf{c}')$ *and private input* $\pi, \mathbf{r}' = \left(\sum_{i=1}^N r_{ij} u_i\right)$.

Proposition 6-3 *Protocol 6-1 is a public-coin honest verifier zero-knowledge proof of knowledge.*

Proof. Completeness and the fact that the protocol is public-coin follow by inspection. We now concentrate on the more interesting properties.

Zero-Knowledge. The simulator picks $\mathbf{u}$ randomly, as in the protocol, then computes $\mathbf{c}$ and $\mathbf{c}'$, again as defined in the protocol, and finally invokes the simulator of the subprotocol π_{rp}. It follows that the simulated view is indistinguishable from the real view of the verifier.

Soundness. The knowledge extractor below does not require that the common input be valid, only that $C, C' \in \mathsf{C}_{pk}^{N\times N}$. Thus, soundness follows from the existence of the knowledge extractor if it has negligible knowledge error.

Knowledge Extraction. At a high level, the knowledge extractor runs and rewinds the protocol a few times, each time with a different test vector $\mathbf{u}$. For each test vector, the subprotocol extractor is executed to extract $\pi, \mathbf{s} = (s_j)$. Once N linearly independent $\mathbf{u}_i$ are obtained for the same permutation π, we can extract the entire matrix of randomization values (r_{ij}) from the N vectors $\mathbf{s}_i$, using fairly straight-forward linear algebra.

We now formalize this intuition. We denote η the order of the plaintext space. Specifically, in the case of BGN, $\eta = n$, while in the case of Paillier, $\eta = n^2$, since we are looking at the n^3 outer layer encryption. We construct our extractor $\mathcal{E}_{\pi_{mrp}}$ on fixed input $(pk, \tilde{A}, \tilde{A}')$:

1. for $l = 1, \dots, N$:

   (a) Start the simulation of an execution between $\mathcal{V}$ and $\mathcal{P}$. Denote by $\mathbf{u}_l = (u_{l,j})$ the random vector chosen by the simulator. Denote by $(pk, \mathbf{c}_l, \mathbf{c}'_l)$ the common input to the subprotocol π_{rp} induced by $\mathbf{u}_l$.

   (b) If there does not exists $(b_{k,l}) \in \mathbf{Z}^{N \times N}$ such that, if $\delta_{kj} = \sum_{l'=1}^{l} b_{k,l'} u_{l',j}$, $\delta_{kj} \equiv 1 \bmod \eta$ for $k = j \leq l$ and $\delta_{kj} \equiv 0 \bmod \eta$ for $k \neq j, k \leq l, j \leq l$, then go to Step 1a. (This is the equivalent of checking for "linear independence," when we don't have a vector space.)

   (c) Run the knowledge extractor $\mathcal{E}_{\pi_{rp}}$ on input $(pk, \mathbf{c}_l, \mathbf{c}'_l)$ and obtain $(\pi_l, \mathbf{s}_l)$ with $\mathbf{s}_l = (s_{l,j})$

2. Compute $B = (b_{k,l}) \in \mathbf{Z}^{N \times N}$ such that $\delta_{kj} = \sum_{l=1}^{N} b_{k,l} u_{l,j} \equiv \Lambda_N^{\text{id}} \bmod \eta$. Compute $B' = B - \Lambda^{\text{id}}$ over $\mathbf{Z}^{N \times N}$. (Of course, B' is congruent to the 0-matrix modulo η, but we consider B' over the integers for now.)

3. Consider $S = (s_{l,j})$, the matrix composed of the extracted randomization value vectors $\mathbf{s}_l$ (arranged as rows of S.) The next step depends on the underlying cryptosystem used.

   In BGN Case. Compute $R = (r_{kj}) = B \times S$, and output (π, R).

   In Paillier Case. Compute

$$R = (r_{kj}) = \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)} / \tilde{a}'_{ij})^{b'_{ki}/\eta} \prod_{l=1}^{N} s_{l,j}^{b_{k,l}} \right)$$

   where the division b_{ki}/η is taken over the integers, and output (π, R).

We first show correctness of our extractor. By construction, we have, for a given test vector $\mathbf{u}_l$ and column j of $\tilde{A}$:

$$\prod_{i=1}^{N} (\tilde{a}'_{ij})^{u_{l,i}} = c'_{l,j} = c_{l,\pi(j)} \mathcal{E}_{pk}(0, s_{l,j}) = \mathcal{E}_{pk}(0, s_{l,j}) \prod_{i=1}^{N} \left(\tilde{a}_{i,\pi(j)}\right)^{u_{l,i}} \tag{6.1}$$

Now, we use B, the special values we computed, to effectively "cancel out" the test vectors $\mathbf{u}_l$. We begin with the left side of Equation 6.1, applying the exponents $(b_{k,l})$ to each element of the vector $\mathbf{c}_l$ and taking the product of all resulting elements, for all $k \in [1, N]$:

$$\begin{aligned}
\prod_{l=1}^{N} \left(\prod_{i=1}^{N} (\tilde{a}'_{ij})^{u_{li}} \right)^{b_{k,l}} &= \prod_{i=1}^{N} \left(\prod_{l=1}^{N} (\tilde{a}'_{ij})^{b_{k,l} u_{li}} \right) \\
&= \prod_{i=1}^{N} \left((\tilde{a}'_{ij})^{\sum_{i=1}^{N} b_{k,l} u_{li}} \right) \\
&= \prod_{i=1}^{N} \left((\tilde{a}'_{ij})^{\delta_{ki}} \right) \\
&= \tilde{a}'_{kj} \prod_{i=1}^{N} \left((\tilde{a}'_{ij})^{b'_{ki}} \right)
\end{aligned}$$

Then, we perform the same operation to the right side of Equation 6.1:

$$\begin{aligned}
\prod_{l=1}^{N} \left(\mathcal{E}_{pk}(0, s_{l,j}) \prod_{i=1}^{N} \left(\tilde{a}_{i,\pi(j)} \right)^{u_{l,i}} \right)^{b_{k,l}} &= \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}} \right) \left(\prod_{i=1}^{N} \prod_{l=1}^{N} (\tilde{a}_{i,\pi(j)})^{b_{k,l} u_{l,i}} \right) \\
&= \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}} \right) \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)})^{\sum_{l=1}^{N} b_{k,l} u_{l,i}} \right) \\
&= \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}} \right) \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)})^{\delta_{ki}} \right) \\
&= \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}} \right) \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)})^{b'_{ki}} \right) \tilde{a}_{k,\pi(j)}
\end{aligned}$$

Thus, setting both halves equal to one another again, we get:

$$\begin{aligned}
\tilde{a}'_{kj} \prod_{i=1}^{N} \left((\tilde{a}'_{ij})^{b'_{ki}} \right) &= \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}} \right) \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)})^{b'_{ki}} \right) \tilde{a}_{k,\pi(j)} \\
\tilde{a}'_{kj} &= \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}} \right) \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)} / \tilde{a}'_{ij})^{b'_{ki}} \right) \tilde{a}_{k,\pi(j)}
\end{aligned}$$

Thus, the randomization values extracted are:

$$r_{kj} = \left(\prod_{l=1}^{N} \mathcal{E}_{pk}(0, s_{l,j})^{b_{k,l}}\right) \left(\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)}/\tilde{a}'_{ij})^{b'_{ki}}\right)$$

We conclude the argument differently depending on the cryptosystem used.

In BGN Case. Note that $b'_{ki} = 0 \bmod \eta$ for all k and i, and the order of any ciphertext divides η. Thus, the second product equals 1 in the ciphertext group. Furthermore, the randomizer space is $\mathbf{Z}_\eta$ so we have

$$\tilde{a}'_{kj} = \mathcal{E}_{pk}\left(0, \sum_{l=1}^{N} b_{k,l}s_{l,j}\right) \tilde{a}_{k,\pi(j)} \quad .$$

In Paillier Case. Again $b'_{ki} = 0 \bmod \eta$ for all k and i, but the order of a ciphertext may be larger than η. However, we know that b'_{ki} is a multiple of η, which means that we can make the following interpretation:

$$\prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)}/\tilde{a}'_{ij})^{b'_{ki}} = \mathcal{E}_{pk}\left(0, \prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)}/\tilde{a}'_{ij})^{b'_{ki}/\eta}\right)$$

Thus, the randomization values in this case are:

$$r_{kj} = \prod_{i=1}^{N} (\tilde{a}_{i,\pi(j)}/\tilde{a}'_{ij})^{b'_{ki}/\eta} \prod_{l=1}^{N} s_{l,j}^{b_{k,l}}$$

We remark that this randomization factor is an element in $\mathbf{Z}^*_{n^3}$ and not in $\mathbf{Z}^*_n$ as expected. However, it remains a witness of re-encryption using one of the alternative Paillier encryption algorithms.

□

The above proof makes three assumptions that we now explore in further detail, to ensure that this extractor is precisely correct. First, we show that we can indeed find test vectors $\mathbf{u}_l$ for the condition we seek. Second we show that the behavioral distribution of this extractor is correct as it simulates the protocol to the prover's code—otherwise the prover may abort. Finally, we analyze the efficiency of this extraction and verify that its success probability is good enough.

Finding "Linearly Independent" Vectors. We need to be certain that finding N vectors $\mathbf{u}_l$ such that there exists matrix B as defined above is within reach. We show that this is possible given the following lemma.

Lemma 6-1 *Let η be a product of $\kappa/2$-bit primes, let N be polynomially bounded in κ, and let $u_1, \ldots, u_{l-1} \in \mathbf{Z}^N$ such that $u_{jj} = 1 \bmod \eta$ and $u_{ji} = 0 \bmod \eta$ for $1 \le i,j \le l-1 < N$ and $i \ne j$. Let $u_l \in [0, 2^{\kappa_c} - 1]^N$ be randomly chosen, where $2^{-\kappa_c}$ is negligible. Then the* *probability that there exists $a_1, \ldots, a_l \in \mathbf{Z}$ such that if we define $u'_l = \sum_{j=1}^{l} a_j u_j \bmod \eta$, then $u'_{l,l} = 1 \bmod \eta$, and $u'_{l,i} = 0 \bmod \eta$ for $i < l$ is overwhelming in κ.*

Proof.

Note that $b = u_{l,l} - \sum_{j=1}^{l-1} u_{l,j} u_{j,l}$ is invertible with overwhelming probability, and, when it is, we view its inverse $b^{-1} \bmod \eta$ as an integer and define $a_j = -b^{-1} u_{l,j} \bmod \eta$ for $j < l$ and $a_l = b^{-1}$. For $i < l$ this gives $u_{l,i} = \sum_{j=1}^{l} a_j u_{j,i} = b^{-1}(1 - a_i u_{ii}) = 0 \bmod \eta$ and for $i = l$ this gives $u_{l,l} = \sum_{j=1}^{l} a_j u_{j,l} = b^{-1}(u_{l,l} - \sum_{j=1}^{l-1} u_{l,j} u_{j,l}) = 1 \bmod \eta$.

□

Remark 6-2 *When the plaintexts are known, and this is the case when C is an encryption of the identity matrix, slightly more efficient techniques can be used. This is sometimes called a "shuffle of known plaintexts" (see [124, 87, 180]).*

6.6.3 Proving Double Re-encryption

Recall that, in the Paillier case, we cannot simply create a trivial encryption of a permutation matrix, because the inner encryption of 0 must be random and must remain secret from all external observers. The solution is to create these double encryptions of 0 *first*, and then to complete the identity matrix with trivial single-outer encryptions of 0.

Intuitively, we start with the trivial double encryption of 0, $\tilde{a}$. The prover then shows that he correctly reencrypted the inner and outer layers of $\tilde{a}$ to yield $\tilde{a}'$. Note that this trivial double encryption of 0 is none other than the Paillier generator:

$$\mathcal{E}^{\mathsf{pai}}_{n^3}(\mathcal{E}^{\mathsf{pai}}_{n^2}(0;0);0) = g^{h_1^0 \bmod n^2} h_2^0 \bmod n^3 = g.$$

The following relation captures the problem of proving correctness of a double-re-encryption, and Figure 6-4 illustrates the proof protocol and creation of a list of N double-reencrypted 0s.

Definition 6-12 *Denote by $\mathcal{R}^{\mathsf{pai}}_{dr}$ the relation consisting of pairs $((1^\kappa, n, \tilde{a}, \tilde{a}'), (r, s))$, such that $\tilde{a}' = \tilde{a}^{h_1^r \bmod n^2} h_2^s \bmod n^3$ with $r, s \in [0, N2^{\kappa_r}]$.*

The protocol here is a typical construction with soundness 50%, which must be iterated (in parallel) κ_c times to make the error probability negligible.

Protocol 6-2 (Double Re-encryption)
Common Input. *A modulus n and $\tilde{a}, \tilde{a}' \in C_{n^3}$*
Private Input. *$r, s \in [0, n2^{\kappa_r}]$ such that $\tilde{a}' = \tilde{a}^{h_1^r \bmod n^2} h_2^s \bmod n^3$.*

1. *$\mathcal{P}$ chooses $r' \in [0, n2^{2\kappa_r}]$ and $s' \in [0, n^3 2^{2\kappa_r}]$, computes $\alpha = \tilde{a}^{h_1^{r'} \bmod n^2} h_2^{s'} \bmod n^3$, and hands α to $\mathcal{V}$.*

$$\begin{tikzcd}
\tilde{a} \arrow[rr] \arrow[dr, dashed, "b=0"'] & & \tilde{a}' \\
& \alpha \arrow[ur, dashed, "b=1"'] &
\end{tikzcd}$$

$$\boxed{\begin{array}{l}
g \longrightarrow \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r_1);s_1\right) \\
g \longrightarrow \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r_2);s_2\right) \\
\vdots \qquad\qquad\qquad \vdots \\
g \longrightarrow \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r_N);s_N\right)
\end{array}}$$

$$\tilde{a}' = \mathcal{RE}^{\mathsf{pai}}_{n^3}\left(\tilde{a}^{\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r)}; s\right)$$

$$\alpha = \mathcal{RE}^{\mathsf{pai}}_{n^3}\left(\tilde{a}^{\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r')}; s'\right)$$

$$\tilde{a}' = \mathcal{RE}^{\mathsf{pai}}_{n^3}\left(\alpha^{\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r'-r)}; s' - s\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r'-r)\right)$$

Figure 6-4: Proof of Correct Double-Reencryption. In the n^3 Generalized Paillier scheme, we can perform double reencryption of ciphertexts $\tilde{a}$. Because we are proving a double-discrete logarithm, our only choice is to provide a triangulation proof with 50% soundness. In the diagram above, the prover performs a second double-reencryption of $\tilde{a}$ into α, then, depending on the verifier challenge bit b, reveals the reencryption exponents for α, or the "difference" in reencryption exponents between α and $\tilde{a}'$. The right-hand side of the figure shows the double-reencryption of N instances of g, which is the trivial double-encryption of 0. These double-reencryptions will serve as the diagonal in the identity matrix, whose columns then get shuffled to generate an encrypted permutation matrix.

2. *$\mathcal{V}$ chooses $b \in \{0,1\}$ randomly and hands b to $\mathcal{P}$.*

3. *$\mathcal{P}$ defines $(e,f) = (r' - br, s' - b(h_1^e \bmod n^2)s)$. Then it hands (e,f) to $\mathcal{V}$.*

4. *$\mathcal{V}$ checks that $\alpha = ((\tilde{a}')^b\tilde{a}^{1-b})^{h_1^e \bmod n^2} h_2^f \bmod n^3$.*

Proposition 6-4 *Protocol 6-2 is a public-coin honest verifier zero-knowledge proof of knowledge for $\mathcal{R}^{\mathsf{pai}}_{dr}$.*

Proof. Completeness and the public-coin property follow by inspection. The honest verifier zero-knowledge simulator simply picks $e, f \in [0, n2^{\kappa_r}]$ and $b \in \{0,1\}$ randomly and defines $\alpha = ((\tilde{a}')^b\tilde{a}^{1-b})^{h_1^e} h_2^f \bmod n^3$. The resulting view is statistically close to a real view, since $2^{-\kappa_r}$ is negligible.

For soundness, note that if we have $\tilde{a}^{h_1^{e_1}} h_2^{f_1} = \alpha = (\tilde{a}')^{h_1^{e_2}} h_2^{f_2} \bmod n^3$ with $e_1, f_1, e_2, f_2 \in \mathbf{Z}$, then we can divide by $h_2^{f_1}$ and take the $h_1^{e_1}$th root on both sides. This gives

$$\tilde{a} = (\tilde{a}')^{h_1^{e_2-e_1}} h_2^{(f_2-f_1)/h_1^{e_1}} \bmod n^3 \ ,$$

which implies that the basic protocol is special 1/2-sound. The protocol is then iterated in parallel κ_c times which gives negligible error probability $2^{-\kappa_c}$. The proof of knowledge property follows immediately from special soundness. □

6.7 Distributed Generation and Obfuscation of a Shuffle

We now explain how to sample and obfuscate the BGN and Paillier shuffles in a distributed way. We consider a number of parties, call them mix servers $\{\mathcal{M}_j\}$, who want to jointly generate an obfuscated shuffle with the typical mixnet properties:

- **robustness:** a cheating mix server is caught, even if there is only one honest verifier.
- **privacy:** if at least one mix server is honest, then privacy is ensured.

The techniques presented in this section are simply an extension of the proof methods used when a single party wants to obfuscate a shuffle. First, we present an intuitive description of how these proof methods can be adapted. Then, we formalize these protocols in the Universally Composable (UC) framework of Canetti [32] and show that our protocols can be used trivially to realize an ideal mixnet functionality in this model. Finally, we prove the security of these constructions.

6.7.1 An Intuitive Overview of the Protocols

Recall that, in both the Paillier and BGN cases, a single prover can demonstrate correct obfuscation by proving, in zero-knowledge, knowledge of the permutation and randomization values that map the columns of an encrypted identity matrix to those of the claimed encrypted permutation matrix $\tilde{\Lambda}^{\pi}$. In the BGN case, the starting matrix can be the trivial encryption of the identity matrix, $\mathcal{E}_{pk}(\Lambda^{\mathrm{id}})$. In the Paillier case, the starter identity matrix must be proven correct using proofs of correct double-encryption of 0's, because the inner encryptions of 0's on the diagonal must remain secret.

Shuffling the Columns Again. The proof of correct column shuffle for the matrix can be iterated. Starting with an encrypted identity matrix, each mix server $\mathcal{M}_j$ shuffles, in turn, the columns of the matrix produced by the previous mix server $\mathcal{M}_{j-1}$. The first mix server uses the identity matrix previously described. Figure 6-5 illustrates this process.

Reencrypting the Double-Encrypted 0's Again. In the Paillier case, the mix servers must also collaborate to jointly produce a list of N double-encrypted 0s, such that the resulting inner-layer encryptions of 0 are hidden from the mix servers themselves, as long as one is honest. Again, the process described for the single prover can simply be iterated. Each mix server $\mathcal{M}_j$ takes the output ciphertexts from the previous mix server $\mathcal{M}_{j-1}$, then

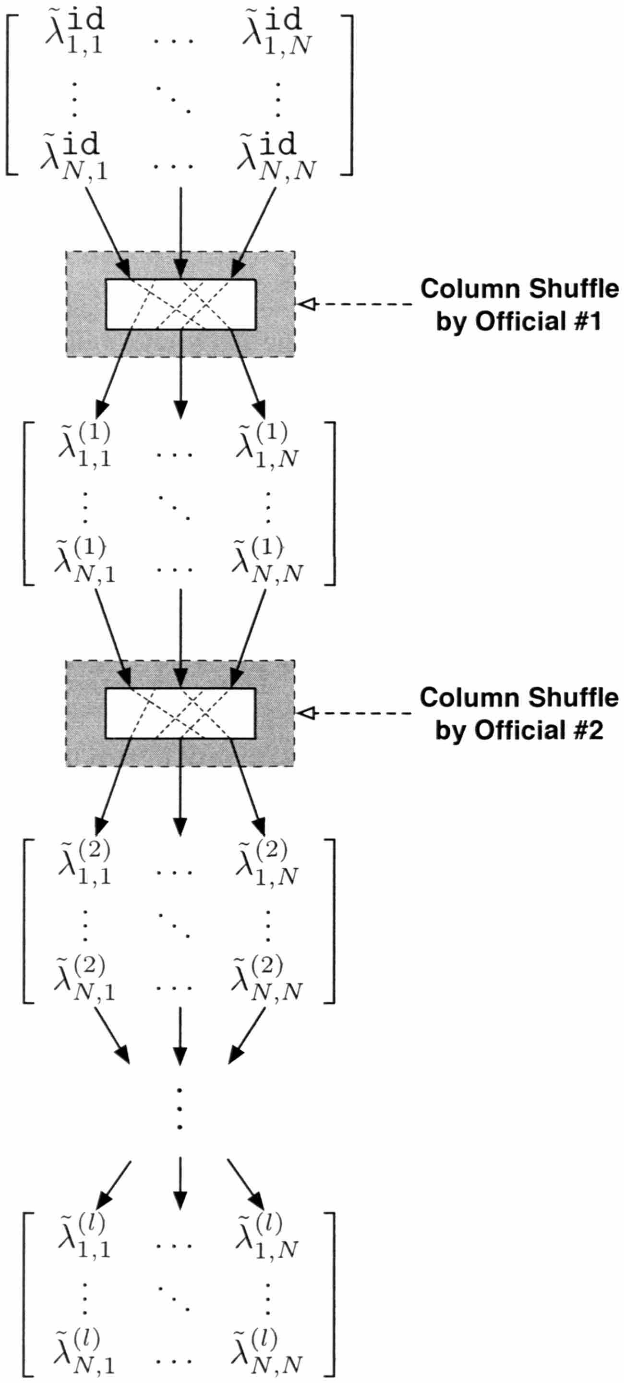


Figure 6-5: Multiple officials shuffling the columns of the encrypted permutation matrix. The encrypted matrix effectively captures all of the shuffle actions and is then ready to "reapply" them by homomorphic matrix multiplication.

reencrypts the inner and outer layers of the N ciphertexts, then proves knowledge of both inner and outer randomization values in the same way as the single-prover method. The first mix server starts with the trivial double-encryption of 0. No shuffling is required here, only chained double-reencryptions. See Figure 6-6.

6.7.2 UC Modeling

The UC model is reviewed in Chapter 2.

Why the UC Model. Our protocols are meant to be used in a mixnet setting, where decryption eventually occurs and plaintexts are produced. If we provide only game-based modeling to prove the indistinguishability of our obfuscation techniques, we still have to prove that this security "composes" nicely with the decryption step that is inevitable in any real setting. We could, of course, prepare new game-based definitions that take this into account. However, this would require that we consider the exact details of the provable decryption step and the various ways in which shuffling and decryption can interact to achieve all of the security properties expected of a mixnet.

The purpose of the Universally Composable framework of Canetti [32] is to simplify and strengthen these kinds of proofs. A protocol proven secure in the UC framework can be safely composed with another protocol proven secure in the UC framework. In particular, given the existing UC mixnet definition of Wikström [179] and the definition of other common protocols in the UC framework, we can focus the modeling and description to our specific contributions, providing only ideal specifications of the other functionalities we expect to compose with our constructions.

Modeling as a Subprotocol. In the UC model, the only natural use of our construction is as a subprotocol used to realize a mixnet: the pure shuffle function is not particularly useful as a reusable component. Thus, we show how to use our protocols to trivially realize an ideal mixnet functionality $\mathcal{F}_{\mathrm{MN}}$. We study the properties of our protocol in a UC hybrid model, containing an ideal authenticated bulletin board $\mathcal{F}_{\mathrm{BB}}$, an ideal coin-flipping functionality $\mathcal{F}_{\mathrm{CF}}$, an ideal zero-knowledge proof of knowledge of a plaintext $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$, and an ideal key generator $\mathcal{F}_{\mathrm{KG}}$ with threshold decryption support.

UC Specifications. Note that our protocol varies slightly depending on the cryptosystem used: the BGN construction is a *decryption* shuffle, while the Paillier construction is a *re-encryption* shuffle. We only indicate which cryptosystem is used when necessary and keep our notation generic otherwise, e.g. we write $\mathcal{CS}$ instead of $\mathcal{CS}^{\mathsf{bgn}}$ or $\mathcal{CS}^{\mathsf{pai}}$. To simplify the exposition, we also say that a public-coin protocol is used "in a distributed way" when the challenge is taken from the ideal coin-flipping functionality.

Protocol 6-3 (Generation of Paillier Double-Encrypted Zeros)
COMMON INPUT. *A public key n and integer N.*
Mix-server $\mathcal{M}_j$ proceeds as follows. There are k mix servers.

$$\boxed{\tilde{a}' = \mathcal{RRE}(\tilde{a}; r'; s') = \mathcal{RE}^{\mathsf{pai}}_{n^3}\left(\tilde{a}^{\mathcal{E}^{\mathsf{pai}}_{n^2}(0;r')}; s'\right)}$$

$$\begin{tikzcd}
d_{0,1} = g \arrow[r] & d_{1,1} = \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0; r_1); s_1\right) \\
d_{0,2} = g \arrow[r] & d_{1,2} = \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0; r_2); s_2\right) \\
\vdots & \vdots \\
d_{0,N} = g \arrow[r] & d_{1,N} = \mathcal{E}^{\mathsf{pai}}_{n^3}\left(\mathcal{E}^{\mathsf{pai}}_{n^2}(0; r_N); s_N\right)
\end{tikzcd}$$

$$\begin{tikzcd}
d_{0,1} \arrow[r] & \boxed{\mathcal{RRE}} \arrow[r] & d_{1,1} \arrow[r] & \boxed{\mathcal{RRE}} \arrow[r] & d_{2,1} & \cdots & d_{l,1} \\
d_{0,2} \arrow[r] & \boxed{\mathcal{RRE}} \arrow[r] & d_{1,2} \arrow[r] & \boxed{\mathcal{RRE}} \arrow[r] & d_{2,2} & \cdots & d_{l,2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
d_{0,N} \arrow[r] & \boxed{\mathcal{RRE}} \arrow[r] & d_{1,N} \arrow[r] & \boxed{\mathcal{RRE}} \arrow[r] & d_{2,N} & \cdots & d_{l,N}
\end{tikzcd}$$

$$\begin{tikzcd}
d_{l,1} \arrow[r, dashed] & d_{l,1} & \mathcal{E}(0) & \dots & \mathcal{E}(0) \\
d_{l,2} \arrow[r, dash, dashed] & \mathcal{E}(0) \arrow[r, dashed] & d_{l,2} & \dots & \mathcal{E}(0) \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
d_{l,N} \arrow[r, dash, dashed] & \mathcal{E}(0) \arrow[r, dash, dashed] & \mathcal{E}(0) \arrow[r, dash, dashed] & \cdots \arrow[r, dashed] & d_{l,N}
\end{tikzcd}$$

Figure 6-6: Multiple officials sequentially perform double reencryption on a list of values. Double Reencryption is represented in the top equation. The starting list of values is composed of trivial encryptions of 0, which is the generator g. Triangulation proofs like the ones in Figure 6-4 are performed for every such double reencryption. The final list of values is then used as the diagonal to form an encrypted identity matrix.

1. *Define* $\mathbf{d}_0 = (\mathcal{E}^{\mathsf{pai}}_{n^3}(\mathcal{E}^{\mathsf{pai}}_{n^2}(0, 0^*)), \ldots, \mathcal{E}^{\mathsf{pai}}_{n^3}(\mathcal{E}^{\mathsf{pai}}_{n^2}(0, 0^*)))$ *of length* N*. This is the starting vector of* N *trivial encryptions of 0.*

2. *For* $l = 1, \ldots, k$ *do:*

   (a) *If* $l \neq j$*, then it is not this current mix server's turn. Wait until the mix server whose turn it is,* $\mathcal{M}_l$ *has posted his output:* $(\mathcal{M}_l, \mathtt{DoubleReencrypt}, \mathbf{d}_l)$*,* $\mathbf{d}_l \in \mathsf{C}_{n^3}$*, appears on* $\mathcal{F}_{\mathrm{BB}}$*. Execute the verifier of Protocol 6-2 in a distributed way. If it fails, then set* $\mathbf{d}_l = \mathbf{d}_{l-1}$*, effectively removing mixserver* $\mathcal{M}_l$ *from the running.*

   (b) *If* $l = j$*, then it is this mix server's turn:*

      i. *Choose* $r_j, s_j \in [0, n2^{\kappa_r}]^N$ *randomly, compute* $\mathbf{d}_j = (d_{j-1,i}^{h_1^{r_{j,i}} \bmod n^2} h_2^{s_{j,i}} \bmod n^3)$*, and publish* $(\mathtt{DoubleReencrypt}, \mathbf{d}_j)$ *on* $\mathcal{F}_{\mathrm{BB}}$*.*

      ii. *Prove using Protocol 6-2 in a distributed way knowledge of* $r_j, s_j \in [0, n2^{\kappa_r}]^N$ *such that* $(n, \mathbf{d}_{j-1}, \mathbf{d}_j) \in \mathcal{R}^{\mathsf{pai}}_{dr}$*.*

3. *Output* $\mathbf{d}_k$*.*

Protocol 6-4 (Generation and Obfuscation of a Shuffle)
COMMON INPUT. *A public key pk and integer* N*.*
Mix-server $\mathcal{M}_j$ *proceeds as follows. There are k mix servers.*

1. IN BGN CASE. *Define* $\tilde{\Lambda}^0 = \mathcal{E}_{pk}(\Lambda^{\mathrm{id}}, 0^*)$*.*

   IN PAILLIER CASE. *Execute Protocol 6-3 and obtain its output* $\mathbf{d}$*. Then form a matrix* $\tilde{\Lambda}^0 = (\tilde{\lambda}^0_{ij})$*, with* $\tilde{\lambda}^0_{ii} = d_i$*, and* $\tilde{\lambda}^0_{ij} = \mathcal{E}^{\mathsf{pai}}_{n^3}(0, 0^*)$ *for* $i \neq j$*.*

2. *For* $l = 1, \ldots, k$ *do:*

   (a) *If* $l \neq j$*, then wait until* $(\mathcal{M}_l, \mathtt{ColumnShuffle}, \tilde{\Lambda}^l)$*,* $\tilde{\Lambda}^l = (\tilde{\lambda}^l_{i,j}) \in \mathsf{C}_{pk}^{N \times N}$*, appears on* $\mathcal{F}_{\mathrm{BB}}$*. Execute the verifier of Protocol 6-1 in a distributed way. If it fails, then set* $\tilde{\Lambda}^l = \tilde{\Lambda}^{l-1}$*.*

   (b) *If* $l = j$*, then do:*

      i. *Choose* $r_j \in \mathsf{R}_{pk}^{N \times N}$ *randomly, choose* $\pi_j \in \Sigma_N$ *randomly, compute* $\tilde{\Lambda}^j = \mathcal{RE}_{pk}\left((\tilde{\lambda}^{j-1}_{i,\pi_j(t)}); r_j\right)$*, and publish* $(\mathtt{ColumnShuffle}, \tilde{\Lambda}^j)$ *on* $\mathcal{F}_{\mathrm{BB}}$*.*

      ii. *Prove, using Protocol 6-1 in a distributed way, knowledge of* $r_j \in \mathsf{R}_{pk}^{N \times N}$ *such that* $\left((pk, \tilde{\Lambda}^{j-1}, \tilde{\Lambda}^j), r_j\right) \in \mathcal{R}_{mrp}$*.*

3. *Output* $\tilde{\Lambda}^k$*.*

6.8 Trivial Mix-Net From Obfuscated Shuffle

We now give the promised trivial realization of an ideal mixnet. The ideal mixnet functionality $\mathcal{F}_{\text{MN}}$ we consider is essentially the same as that used by Wikström [179, 180, 181]. It simply accepts inputs and waits until a majority of the mix-servers requests that the mixing process starts. Finally, it sorts the inputs and outputs the result. Recall that this is the ideal functionality definition, which we must prove is realized by our protocols of the previous section, using the hybrid model previously described.

Recall that $\mathcal{C}_\mathcal{I}$ is the UC designation of the communication medium, across which all messages pass. Recall also that all messages sent to $\mathcal{C}_\mathcal{I}$ must first designate a recipient. Conversely, messages received from $\mathcal{C}_\mathcal{I}$ are tagged with their authenticated sender. Finally, recall that, in the ideal model of UC functionalities, the ideal adversary $\mathcal{S}$ has the power to delay all messages sent to the functionality. $\mathcal{S}$ is effectively the gatekeeper to the ideal functionality, although it does not see the inputs of honest senders, of course.

6.8.1 Ideal Mixnet Functionality

Functionality 2 (Mix-Net) *The ideal functionality for a mixnet, $\mathcal{F}_{\text{MN}}$, running with mix-servers $\mathcal{M}_1, \ldots, \mathcal{M}_k$, senders $\mathcal{P}_1, \ldots, \mathcal{P}_N$, and ideal adversary $\mathcal{S}$ proceeds as follows*

1. *Initialize the following storage:*

    - *list $L = \emptyset$, the list of messages received from senders,*
    - *set $J_S = \emptyset$, the set of senders who have sent a message, and*
    - *set $J_M = \emptyset$, the set of mix servers who have asked the mix to run.*

2. *Repeatedly wait for messages from $\mathcal{C}_\mathcal{I}$:*

    - *Upon receipt of $(\mathcal{P}_i, \textbf{\textit{Send}}, m_i)$ with $m_i \in \{0,1\}^\kappa$ and $i \notin J_S$:*
        - *set $L \leftarrow L \cup \{m_i\}$*
        - *set $J_S \leftarrow J_S \cup \{i\}$*
    - *Upon receipt of $(\mathcal{M}_j, \textbf{\textit{Run}})$:*
        - *$J_M \leftarrow J_M \cup \{j\}$*
        - *If $|J_M| > k/2$, then sort the list L lexicographically to form a list L', and send:*
            - *$\{(\mathcal{M}_l, \texttt{Output}, L')\}_{l=1}^k$*

          and ignore further messages.

        If $|J_M| \leq k/2$, send $(\mathcal{S}, \mathcal{M}_j, \textbf{\textit{Run}})$, and keep waiting for messages.

6.8.2 Other Ideal Functionalities for the Hybrid Construction

The functionalities of the hybrid world are fairly natural. The bulletin board $\mathcal{F}_{\mathrm{BB}}$ is authenticated; everybody can write to it, and nobody can delete anything. It also stores the order in which messages appear. The coin-flipping functionality $\mathcal{F}_{\mathrm{CF}}$ waits for a coin-request and then simply outputs the requested number of random coins. The zero-knowledge proof of knowledge of a plaintext $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$ allows a sender to submit a public key pk, a ciphertext c, a message m, and a random string $r \in \{0,1\}^*$, and simply tells the mix-servers if $c = \mathcal{E}_{pk}(m, r)$ or not. The key generation functionality generates a key pair $(pk, sk) = \mathcal{G}(1^\kappa)$ and outputs the public key pk. Then if it receives more than $k/2$ requests to decrypt a certain list of ciphertexts, it decrypts it using sk and outputs the result. We now review formal definitions of these functionalities.

Recall that, at any point, the ideal adversary may delay messages to the ideal functionality. We consider that this delay capability is built into the communication network $\mathcal{C}_{\mathcal{I}}$, as described in Chapter 2. However, there are certain behaviors of the following functionalities which are not generic according to the properties of $\mathcal{C}_{\mathcal{I}}$. When these occur, we describe them in greater detail.

Functionality 3 (Bulletin Board) *The ideal bulletin board functionality, $\mathcal{F}_{\mathrm{BB}}$, running with parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and ideal adversary $\mathcal{S}$ proceeds as follows. Intuitively, the ideal adversary is, as always, able to delay any message. In addition, in the case of the bulletin board, the ideal adversary can choose to delay certain messages* based on their content, *since the adversary sees bulletin board content. To model this accordingly, the ideal functionality has two databases of received messages: the first, D_1 contains messages received but not yet* approved *by the ideal adversary, and the second, D_2, contains messages received and approved.*

1. *Initialize the following storage:*
    - *two databases D_1, D_2 indexed on integers, which will be used to store messages posted to the bulletin board.*
    - *two counters c_1 and c_2 that respectively index into D_1 and D_2.*
2. *Repeatedly wait for a message:*
    - *Upon receiving $(\mathcal{P}_i, \texttt{Write}, m_i)$, $m_i \in \{0,1\}^*$:*
        - *store $D_2[c_2] = (\mathcal{P}_i, m_i)$,*
        - *set $c_2 \leftarrow c_2 + 1$, and*
        - *send $(\mathcal{S}, \textbf{Input}, c_2, \mathcal{P}_i, m_i)$.*
    - *Upon receiving $(\mathcal{S}, \textbf{AcceptInput}, c)$ check if a tuple $(\mathcal{P}_i, m_i)$ is stored in $D_2[c]$. If so, then:*
        - *store $D_1[c_1] = (\mathcal{P}_i, m_i)$,*

- *set* $c_1 \leftarrow c_1 + 1$, *and*
- *send* $(\mathcal{S}, \textbf{AcceptInput}, c)$.

- *Upon receiving* $(\mathcal{P}_j, \textbf{Read}, c)$, *check if a tuple* $(\mathcal{P}_i, m_i)$ *is stored in* $D_1[c]$.
  If so, send
  $((\mathcal{S}, \mathcal{P}_j, \textbf{Read}, c, \mathcal{P}_i, m), (\mathcal{P}_j, \textbf{Read}, c, \mathcal{P}_i, m_i))$.
  If not, send
  $((\mathcal{S}, \mathcal{P}_j, \textbf{NoRead}, c), (\mathcal{P}_j, \textbf{NoRead}, c))$.

Functionality 4 (Coin-Flipping) *The ideal Coin-Flipping functionality,* $\mathcal{F}_{\text{CF}}$, *with mix-servers* $\mathcal{M}_1, \ldots, \mathcal{M}_k$, *and adversary* $\mathcal{S}$ *proceeds as follows:*

1. *Initialize* $J_\kappa = \emptyset$ *for all* κ.
2. *On receipt of* $(\mathcal{M}_j, \textbf{GenerateCoins}, \kappa)$, *set* $J_\kappa \leftarrow J_\kappa \cup \{j\}$.
   If $|J_\kappa| = k$, *then set* $J_\kappa \leftarrow \emptyset$, *choose* $c \in \{0,1\}^\kappa$ *randomly, and send* $((\mathcal{S}, \textbf{Coins}, c), \{(\mathcal{M}_j, \textbf{Coins}, c)\}_{j=1}^{k})$.

Functionality 5 (Key Generator) *The ideal key generator* $\mathcal{F}_{\text{KG}}$, *running with mix-servers* $\mathcal{M}_1, \ldots, \mathcal{M}_k$, *senders* $\mathcal{P}_1, \ldots, \mathcal{P}_N$, *and ideal adversary* $\mathcal{S}$ *proceeds as follows:*

1. *Initialize a set* $J_D = \emptyset$. *Compute* $(pk, sk) = \mathcal{G}(1^\kappa)$.
2. *send* $((\mathcal{S}, \textbf{PublicKey}, pk), \{(\mathcal{M}_j, \textbf{PublicKey}, pk)\}_{j=1}^{k}, \{(\mathcal{P}_i, \textbf{PublicKey}, pk)\}_{i=1}^{N})$.
3. *Repeatedly wait for messages.*
   Upon reception of $(\mathcal{M}_j, \textbf{Decrypt}, c)$, *set* $J_D \leftarrow J_D \cup \{(\mathcal{M}_j, c)\}$.
   If $|\{j : (\mathcal{M}_j, c) \in J_D\}| > k/2$, *then send*
   $((\mathcal{S}, \textbf{PublicKey}, pk), \{(\mathcal{M}_j, \textbf{PublicKey}, pk)\}_{j=1}^{k})$.

Functionality 6 (Zero-Knowledge Proof of Knowledge) *Let* $\mathcal{L}$ *be a language given by a binary relation* $\mathcal{R}$. *The ideal* zero-knowledge proof of knowledge *functionality* $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$ *of a witness* w *to an element* $x \in \mathcal{L}$, *running with provers* $\mathcal{P}_1, \ldots, \mathcal{P}_N$, *and verifiers* $\mathcal{M}_1, \ldots, \mathcal{M}_k$, *proceeds as follows.*

1. *Initialize a database* D.
2. *Repeatedly wait for messages:*
   - *Upon receipt of* $(\mathcal{P}_i, \textbf{Prover}, x, w)$,
     store $D[(\mathcal{P}_i, x)] = w$, *and send* $(\mathcal{S}, \mathcal{P}_i, \textbf{Prover}, x, R(x, w))$.
     Ignore further messages from $\mathcal{P}_i$.

- *Upon receipt of* $(\mathcal{M}_j, \texttt{Question}, \mathcal{P}_i, x)$*:*
  (a) *if* $J_{\mathcal{P}_i,x}$ *is not initialized, set* $J_{\mathcal{P}_i,x} = \emptyset$*,*
  (b) *set* $J_{\mathcal{P}_i,x} \leftarrow J_{\mathcal{P}_i,x} \cup \{j\}$*.*
  (c) *Let* w *be* $D[(\mathcal{P}_i, x)]$*, or the empty string if nothing is stored.*

  If $|J_{\mathcal{P}_i,x}| = k$*, send*
  $((\mathcal{S}, \mathcal{M}_j, \texttt{Verifier}, \mathcal{P}_i, x, R(x,w)), \{(\mathcal{M}_j, \texttt{Verifier}, \mathcal{P}_i, x, R(x,w))\}_{j=1}^{k})$.

  If$|J_{\mathcal{P}_i,x}| < k$*, send*
  $(\mathcal{S}, \mathcal{M}_j, \texttt{Question}, \mathcal{P}_i, x)$.

We care about this zero-knowledge proof of knowledge functionality for the purpose of proving knowledge of a plaintext and randomization factor that correspond to a public ciphertext. Specifically, the relation for which we prove knowledge is defined below.

Definition 6-13 *Let* $\mathcal{CS}$ *be a cryptosystem. Then denote by* $\mathcal{R}_{kp}$ *the relation consisting of pairs* $((pk, c), (m, r))$ *such that* $c = \mathcal{E}_{pk}(m, r)$ *and* $m \in \{0,1\}^{\kappa_m}$*.*

6.8.3 Realization of Mixnet Functionality

We now consider the hybrid model with the ideal functionalities just defined, and we use our concrete protocols to realize a mixnet. Note that, given the features of our protocol, this realization is now quite trivial. The complexity is in the proof of security.

Protocol 6-5 (Trivial Mix-Net)
Sender $\mathcal{P}_i$ *proceeds as follows.*

1. *Wait for* $(\texttt{PublicKey}, pk)$ *from* $\mathcal{F}_{\mathrm{KG}}$*.*
2. *Wait for an input* $m \in \{0,1\}^{\kappa_m}$*, choose* $r \in \{0,1\}^*$ *randomly, and compute* $c = \mathcal{E}_{pk}(m, r)$*. Then publish* $(\texttt{Send}, c)$ *on* $\mathcal{F}_{\mathrm{BB}}$ *and hand* $(\texttt{Prover}, (pk, c), (m, r))$ *to* $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$*.*

Mix-server $\mathcal{M}_j$ *proceeds as follows.*
Offline Phase

1. *Wait for* $(\texttt{PublicKey}, pk)$ *from* $\mathcal{F}_{\mathrm{KG}}$*.*
2. *Execute Protocol 6-4 (with either BGN or Paillier as appropriate) and denote the output obfuscated (decryption/re-encryption) shuffle by* $\tilde{\Lambda}^{\pi}$*.*

Online Phase

1. *Initialize*

- $J_M = \emptyset$, *to keep track of the set of mix servers who have sent the* ***Run*** *message.*
- $J_S = \emptyset$, *to keep track of the senders who have sent an input.*

2. *repeatedly wait for new inputs or the next new message on* $\mathcal{F}_{\mathrm{BB}}$.

   - *On input* (***Run***), *send* (***Write***, ***Run***) *to* $\mathcal{F}_{\mathrm{BB}}$.
   - *If* $(\mathcal{M}_j, \textbf{\textit{Run}})$ *appears on* $\mathcal{F}_{\mathrm{BB}}$, *then set* $J_M \leftarrow J_M \cup \{j\}$. *If* $|J_M| > k/2$, *go to Step 3.*
   - *If* $(\mathcal{P}_\gamma, \textbf{\textit{Send}}, c_\gamma)$ *appears on* $\mathcal{F}_{\mathrm{BB}}$ *for* $\gamma \notin J_S$ *then do:*

     (a) *Set* $J_S \leftarrow J_S \cup \{\gamma\}$.

     (b) *Send* $(\textbf{\textit{Question}}, \mathcal{P}_\gamma, (pk, c_\gamma))$ *to* $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$ *and wait for a reply* $(\textbf{\textit{Verifier}}, \mathcal{P}_\gamma, (pk, c_\gamma), b_\gamma)$ *from* $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$.

3. *Let* $J_S' \subset J_S$ *be the set of* γ *such that* $b_\gamma = 1$. *Form a list of trivial encryptions* $\mathbf{c}_{pad} = (\mathcal{E}_{pk}(0, 0^*), \ldots, \mathcal{E}_{pk}(0, 0^*))$ *of length* $N - |J_S'|$. *Then form the list* $\mathbf{c} = (c_\gamma)_{\gamma \in J_S} \| \mathbf{c}_{pad}$, *and compute* $\mathbf{c}' = \mathbf{c} \star \tilde{\Lambda}^{\pi}$.

4. In BGN Case. *Hand* $(\textbf{\textit{Decrypt}}, \mathbf{c}')$ *to* $\mathcal{F}_{\mathrm{KG}}$ *and wait until it returns* $(\textbf{\textit{Decrypted}}, \mathbf{m})$. *Form a new list* $\mathbf{m}'$ *by sorting* $\mathbf{m}$ *lexicographically and removing* $N - |J_S'|$ *copies of* 0. *Then output* $(\textbf{\textit{Output}}, \mathbf{m}')$.

   In Paillier Case. *Hand* $(\textbf{\textit{Decrypt}}, \mathbf{c}')$ *to* $\mathcal{F}_{\mathrm{KG}}$ *and wait until it returns* $(\textbf{\textit{Decrypted}}, \mathbf{c}'')$. *Hand* $(\textbf{\textit{Decrypt}}, \mathbf{c}'')$ *to* $\mathcal{F}_{\mathrm{KG}}$ *and wait until it returns* $(\textbf{\textit{Decrypted}}, \mathbf{m})$. *Form a new list* $\mathbf{mess}'$ *by sorting* $\mathbf{mess}$ *lexicographically and removing* $N - |J_S'|$ *copies of* 0. *Then output* $(\textbf{\textit{Output}}, \mathbf{m}')$.

Remark 6-3 *The decryption step at the end of the protocol can be implemented efficiently in a distributed and verifiable way using known methods (e.g. [48, 181]).*

Proposition 6-5 *Protocol 6-5 securely realizes* $\mathcal{F}_{\mathrm{MN}}$ *in the* $(\mathcal{F}_{\mathrm{BB}}, \mathcal{F}_{\mathrm{CF}}, \mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}, \mathcal{F}_{\mathrm{KG}})$*-hybrid model with respect to static adversaries corrupting any minority of the mix-servers and any set of senders under the polynomial indistinguishability of the BGN or Paillier cryptosystem respectively.*

Proof. The proof for this UC construction is given in Section 6.10. □

Construction	Sample & Obfuscate	Prove	Precompute	Evaluate
BGN with $N = 350$	14 (0.5h)	3 (0.1h)	NA	588 (19.6h)
Paillier with $N = 2000$	556 (18.5h)	290 (9.7h)	3800 (127h)	533 (17.8h)

Figure 6-7: The table gives the complexity of the operations in terms of 10^4 modular κ-bit exponentiations and in parenthesis the estimated running time in hours assuming that $\kappa = 1024$, $\kappa_c = \kappa_r = 50$, and that one exponentiation takes 12 msec to compute (a 1024-bit exponentiation using GMP [85] takes 12 msec on our 3 GHz PC). We use maximal values of N for each scheme that yield practical times in a real election setting.

6.9 Complexity Estimates

Our constructions clearly require $O(N^2)$ exponentiations, but we give estimates that show that the constant hidden in the big-O notation is reasonably small in some practical settings. For simplicity we assume that the cost of squaring a group element equals the cost of multiplying two group elements and that computing an exponentiation using a κ_e-bit integer modulo a κ-bit integer corresponds to κ_e/κ full exponentiations modulo a κ-bit integer. We optimize using fixed-base exponentiation and simultaneous exponentiation (see [114]). We assume that evaluating the bilinear map corresponds to computing 6 exponentiations in the group $\mathbb{G}_1$ and we assume that such one such exponentiation corresponds to 8 modular exponentiations. This seems reasonable, although we are not aware of any experimental evidence. In the Paillier case we assume that multiplication modulo n^s is s^2 times as costly as multiplication modulo n. We assume that the proof of a shuffle requires $8N$ exponentiations (this is conservative).

Most exponentiations when sampling and obfuscating a shuffle are fixed-base exponentiations. The only exception is a single exponentiation each time an element is double-re-encrypted, but there are only N such elements. In the proof of correct obfuscation the bit-size κ_c of the elements in the random vector u used in Protocol 6-1 is much smaller than the security parameter, and simultaneous exponentiation is applicable. In the Paillier case, simultaneous exponentiation is applicable during evaluation, and pre-computation lowers the on-line complexity. Unfortunately, this is not work in the BGN case due to the bilinear map. For practical parameters we get the estimates in Fig. 6-7. To compute these estimates, we prepared a short Scheme program, presented at the end of this section.

The BGN construction is only practical when N is small and the maximal number of bits in any submitted ciphertext is small. The Paillier construction on the other hand is practical for normal sized voting precincts in the USA: full length messages can be accommodated, and given one week of precomputing, evaluating the obfuscated shuffle can be done overnight. We note that all constructions are easily parallelized, i.e., larger values of N can be accommodated or the running time can be reduced directly by using more computers.

```
;; Program to estimate the complexity of sampling, obfuscating,
;; proving correctness of an obfuscation, and evaluating.

;; isbgn:   If equal to 1 we compute BGN complexity and otherwise Paillier
;; secp:    Main security parameter, number of bits in Paillier modulus.
;; secpr:   Bit-size of random padding in Schnorr proofs without mod-reduction.
;; secpc:   Bit-size of challenge elements.
;; logb:    Number of bits in each "chunk" in fixed-base exponentiation.
;; wsmall:  Width of simultaneous exponentiation when we have small exponents.
;; wbig:    Width of simultaneous exponentiation when we have big exponents.
;; oneexp:  The time is takes to compute one modular secp-bit exponentiation.
;; N:       Number of senders

(define (performance isbgn secp secpr secpc logb wsmall wbig oneexp N)

  ;; Displays time in minutes and hours to compute one exponentation
  ;; modulo a secp-bit integer
  (define (display-as-time noexp)
    (display (/ (truncate (round (/ (* oneexp noexp) 36))) 100)))

  ;; The cost in terms of multiplications to evaluate the bilinear map.
  (define (bmap-cost) (* 6 (* 1.5 secp)))

  ;; The cost in terms of modular exponentiations to evaluate the bilinear map.
  (define (ECC-cost) 8)

  ;; The cost of performing a fixed-base exponentation given precomputation.
  ;; The parameter is the number of bits in the exponent.
  (define (logb-fixedbase expsize)
    (let ((b (expt 2 logb)))
      (- (+ (* (/ (- b 1) (* b logb)) expsize) b) 3)))

  ;; Precomputation needed for wbig
  (define (w-simultaneous-wbig-precomp)
    (expt 2 wbig))

  ;; The cost of wsmall-wise simultaneous exponentiation.
  ;; The parameter is the number of bits in the exponent.
  (define (w-simultaneous-small expsize)
    (/ (- (+ (* 2 expsize) (expt 2 wsmall)) 4)
       wsmall))

  ;; The cost of wsmall-wise simultaneous exponentiation.
  ;; The parameter is the number of bits in the exponent.
  (define (w-simultaneous-big-withoutprecomp expsize)
    (/ (- (* 2 expsize) 4)
       wbig))

  ;; The cost of a proof of a shuffle of lists.
  ;; This value is rather arbitrarily chosen.
  (define (proof-of-shuffle) (* 8 N secp))

  ;; The cost of the proof of a shuffle of the rows in a matrix.
  (define (matrix-reencryption-proof)
    (if (> isbgn 0)
        (+ (* N N (w-simultaneous-small secpc))
           (proof-of-shuffle))
        (+ (* 9 N N (w-simultaneous-small secpc))
           (* 9 (proof-of-shuffle)))))

  ;; The cost of a single proof of double re-encryption.
  (define (double-reencryption)
    (* secpc (+ (* 9 (logb-fixedbase (+ (* 3 secp) (* 2 secpr))))
(* 4 (logb-fixedbase (+ secp (* 2 secpr))))
                (* 2 secp 1.5 9))))

  ;; Translate the number of input multiplications to the
  ;; corresponding cost in terms of exponentiations
  (define (mults-to-exps mults)
    (round (/ mults (* 1.5 secp))))

  ;; The cost of sampling and obfuscating a shuffle.
  ;; In other words the cost of computing a random encrypted
  ;; "permutation matrix".
  (define (sample-obfuscate-exps)
    (mults-to-exps
     (if (> isbgn 0)
         (* N N (logb-fixedbase secp) (ECC-cost))
         (* (+ (* 9 N N) (* 4 N))
            (logb-fixedbase (+ secp secpr))))))

  ;; The cost of proving the correctness of a matrix.
  (define (prove-exps)
    (mults-to-exps
     (if (> isbgn 0)
         (* (matrix-reencryption-proof) (ECC-cost))
         (+ (matrix-reencryption-proof)
            (* N (double-reencryption))))))

  ;; Cost of precomputation for wbig-simultaneous exponentiation
  (define (precompute-paillier-exps)
    (mults-to-exps (/ (* N N (w-simultaneous-wbig-precomp)) wbig)))

  ;; The cost of performing homomorphic matrix multiplication.
  (define (eval-exps)
    (mults-to-exps
     (if (> isbgn 0)
         (* N N (+ (bmap-cost) 1) (ECC-cost))
         (* 9 N N (w-simultaneous-big-withoutprecomp (* 2 secp))))))

  (define (display-result)
    (newline)
    (if (> isbgn 0)
        (display "BGN: ")
        (display "PAI: "))
    (display "secp=")
    (display secp)
    (display ", secpr=")
    (display secpr)
    (display ", secpc=")
    (display secpc)
    (display ", logb=")
    (display logb)
    (display ", wsmall=")
    (display wsmall)
    (display ", wbig=")
    (display wbig)
    (display ", bgood=")
    (display (round (logb-fixedbase (* 2 secp))))
    (display ", N=")
    (display N)
    (newline)

    (display "Sample and Obfuscate ")
    (display (sample-obfuscate-exps))
    (display "   ")
    (display-as-time (sample-obfuscate-exps))
    (newline)
    (display "Prove                ")
    (display (prove-exps))
    (display "   ")
    (display-as-time (prove-exps))
    (newline)
    (cond ((= isbgn 0)
           (display "Precomp. Eval        ")
           (display (precompute-paillier-exps))
           (display "   ")
           (display-as-time (precompute-paillier-exps))
           (newline)))
    (display "Evaluate             ")
    (display (eval-exps))
    (display "   ")
    (display-as-time (eval-exps)))

  (display-result)
  '()
  )

;; Compute for both BGN and Paillier
(performance 1 1024 50 50 6 5 18 0.012 350)
(performance 0 1024 50 50 6 5 18 0.012 2000)
```

6.10 UC Proof of Mixnet

The proof proceeds as do most proofs of security in the UC-framework. First, we define an ideal adversary $\mathcal{S}$ that runs the real adversary $\mathcal{A}$ as a black-box. Then, we show that if the environment can distinguish, with non-negligible probability, the ideal model run with the ideal adversary from the real model run with the real adversary, then we can break one of the security properties assumed of the underlying cryptosystem.

The Ideal Adversary. The ideal adversary simulates the view of the real model to the real adversary. Denote by $I_{\mathcal{M}}$ and $I_{\mathcal{P}}$ the indices of the corrupted mix-servers and senders correspondingly. The ideal adversary corrupts the corresponding dummy parties. Then it simulates the real model as follows.

Links Between Corrupted Parties and the Environment. The ideal adversary simulates the simulated environment $\mathcal{Z}'$ such that it appears as if $\mathcal{A}$ is communicating directly with $\mathcal{Z}$. Similarly, it simulates the corrupted dummy party $\tilde{\mathcal{M}}_j$ (or $\tilde{\mathcal{P}}_i$) for $j \in I_{\mathcal{M}}$ (or $i \in I_{\mathcal{P}}$) such that it appears as if $\mathcal{M}_j$ (or $\mathcal{P}_i$) is directly communicating with $\mathcal{Z}$. This is done by simply forwarding messages.

Simulation of Honest Senders. The ideal adversary clearly does not know the messages submitted by honest senders $\{\mathcal{P}_i\}_{i \notin I_{\mathcal{P}}}$ to $\mathcal{F}_{\text{MN}}$ in the ideal model. Thus, it must use some placeholders in its simulation and then make sure that this is not noticed. It simply uses placeholder messages with value 0. $\mathcal{S}$ effectively simulates honest senders $\mathcal{P}_i$ as if they had input 0, performing every other action honestly, including posting to $\mathcal{F}_{\text{BB}}$.

More precisely, when $\mathcal{S}$ receives $(\mathcal{S}, \mathcal{P}_i, \texttt{Input}, f)$ – indication that the ideal $\mathcal{P}_i$ has sent a message—it simulates $\mathcal{P}_i$ on input 0. $\mathcal{S}$ honestly performs encryption of this message 0 and posts it to its own simulation of $\mathcal{F}_{\text{BB}}$, which generates message $(\texttt{Input}, f', \mathcal{P}_i, c_i)$. $\mathcal{S}$ intercepts this index f' and stores the index correspondence (f, f'), and continues the simulation.

$\mathcal{F}_{\text{BB}}$ otherwise passes the $\texttt{AcceptInput}$ messages correctly from the simulated real world to the ideal world and back, pausing simulation until the ideal functionality lets it continue.

Extraction From Corrupt Senders. When a corrupt sender submits a message in the simulated real model, then the ideal adversary must instruct the corresponding corrupted dummy sender to submit the same message.

When some $\mathcal{M}_j$ with $j \notin I_{\mathcal{M}}$ receives $(\mathcal{M}_j, \texttt{Verifier}, \mathcal{P}_\gamma, (pk, c_\gamma), 1)$ from $\mathcal{F}_{\text{ZK}}^{\mathcal{R}_{kp}}$ in the simulation of Step 2b, the simulation is interrupted and the message m_γ encrypted in c_γ is extracted from the simulation of $\mathcal{F}_{\text{ZK}}^{\mathcal{R}_{kp}}$. Then $\tilde{\mathcal{P}}_\gamma$ is instructed to submit m_γ to $\mathcal{F}_{\text{MN}}$. When $\mathcal{S}$ receives $(\tilde{\mathcal{P}}_\gamma, \texttt{Input}, f)$ it hands $(\texttt{AcceptInput}, f)$ to $\mathcal{F}_{\text{MN}}$ and waits until it receives $(\tilde{\mathcal{P}}_i, \texttt{Send})$ from $\mathcal{F}_{\text{MN}}$. Then the simulation of $\mathcal{M}_j$ is continued.

Simulation of Honest Mix-Servers. When an honest mix-server signals that it wishes to start the mixing process, the corresponding simulated mix-server must do the same.

When $\mathcal{S}$ receives $(\tilde{\mathcal{M}}_j, \texttt{Input}, f)$ from $\mathcal{F}_{\text{MN}}$, with $j \notin I_{\mathcal{M}}$, it gives the simulated mix-server $\mathcal{M}_j$ the input $\texttt{Run}$. When $\mathcal{F}_{\text{BB}}$ is about to output $(\mathcal{A}, \texttt{Input}, f', \mathcal{M}_j, \texttt{Run})$ the simulation is interrupted and (f, f') stored before the simulation is continued. When $\mathcal{F}_{\text{BB}}$ receives $(\mathcal{A}, \texttt{AcceptInput}, f')$ the simulation of $\mathcal{F}_{\text{BB}}$ is interrupted and $\mathcal{S}$ hands $(\texttt{AcceptInput}, f)$ to $\mathcal{F}_{\text{MN}}$. When it returns the simulation is continued. Note that it normally returns $(\tilde{\mathcal{M}}_j, \texttt{Run})$ or $(\tilde{\mathcal{M}}_j, \texttt{Output}, L')$, but the empty message can be returned if the accept instruction is ignored.

Extraction From Corrupt Mix-Servers. When a corrupted mix-server signals that it wishes to start the mixing process in the simulated real model, the corresponding corrupted dummy mix-server must do the same.

When $\mathcal{F}_{\mathrm{BB}}$ is about to hand $(\mathcal{A}, \mathtt{AcceptInput}, f')$ to $\mathcal{C}_{\mathcal{I}}$ and $(\mathcal{M}_j, \mathtt{Run})$ has been stored in the database D_1 of $\mathcal{F}_{\mathrm{BB}}$, the simulation is interrupted. Then $\mathcal{S}$ instructs $\tilde{\mathcal{M}}_j$ to hand $\mathtt{Run}$ to $\mathcal{F}_{\mathrm{MN}}$ and waits until it receives $(\tilde{\mathcal{M}}_j, \mathtt{Input}, f)$ from $\mathcal{F}_{\mathrm{MN}}$. Then it hands $(\mathtt{AcceptInput}, f)$ to $\mathcal{F}_{\mathrm{MN}}$. When it returns the simulation of $\mathcal{F}_{\mathrm{BB}}$ is continued.

Simulation of Decryption. Note that when the decryption step is simulated by $\mathcal{F}_{\mathrm{KG}}$, $\mathcal{S}$ already knows the output L' of $\mathcal{F}_{\mathrm{MN}}$. Simulation proceeds slightly differently in the BGN and Paillier cases, but in both cases a list $(m'_1, \ldots, m'_N)$ is formed by padding L' with zeros until it has size N.

BGN CASE. Choose $\pi_{sim} \in \Sigma_N$ randomly and use $(m'_{\pi_{sim}(1)}, \ldots, m'_{\pi_{sim}(N)})$ in the simulation of $\mathcal{F}_{\mathrm{KG}}$.

PAILLIER CASE. In this case the key generator is called twice to decrypt the outer and inner layer of the ciphertexts respectively. Choose $\pi_{sim} \in \Sigma_N$ and $\bar{r}_i \in \mathbf{Z}_n^*$ randomly and compute $c'' = (c''_1, \ldots, c''_N) = (\mathcal{E}^{\mathsf{pai}}_{n,2}(m'_{\pi_{sim}(1)}, \bar{r}_1), \ldots, \mathcal{E}^{\mathsf{pai}}_{n,2}(m'_{\pi_{sim}(N)}, \bar{r}_N))$. In the first call use c'' in the simulation of $\mathcal{F}_{\mathrm{KG}}$ and in the second call use $(m'_{\pi_{sim}(1)}, \ldots, m'_{\pi_{sim}(N)})$.

Reaching a Contradiction. Consider any real adversary $\mathcal{A}$ and environment $\mathcal{Z}$. We show that if $\mathcal{Z}$ can distinguish the ideal model run with $\mathcal{S}$ from the real model run with $\mathcal{A}$, then we can break the indistinguishability of the underlying cryptosystem.

Denote by T_{ideal} a simulation of the ideal model run with $\mathcal{S}$ and $\mathcal{Z}$ and denote by T_{real} a simulation of the real model run with Adv and $\mathcal{Z}$. Denote by T_l the simulation of T_0 except for the following modifications of simulation honest senders $\mathcal{P}_i$ for $i \notin I_{\mathcal{P}}$ and $i < l$ and $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$.

1. $\mathcal{P}_i$ submits $(\mathtt{Prover}, (pk, c), \perp)$ to $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$, i.e., it does not submit any witness. Instead, the simulation of $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$ is modified to behave as if it received a witness from these senders.
2. Instead of giving $\mathcal{P}_i$ the zero input, $\mathcal{S}$ peeks into the ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{\mathcal{R}_{kp}}$ and uses the message m_i submitted by the corresponding honest dummy sender $\tilde{\mathcal{P}}_i$.

Claim 1 $|\Pr[T_0 = 1] - \Pr[T_N = 1]|$ *is negligible.*

Proof. This follows by a standard hybrid argument. We need only observe that the secret key of the cryptosystem is not used by $\mathcal{S}$ in its simulation. More precisely, define Adv_l to be the polynomial indistinguishability adversary for the cryptosystem that takes a public key pk as input and simulates T_l, except that if $l \notin I_{\mathcal{P}}$ it interrupts the simulation when $\mathcal{P}_l$ is about to compute it submission ciphertext. Then it hands $(M_0, M_1) = (m_l, 0)$ to the experiment, where m_l is the message that $\tilde{\mathcal{P}}_l$ handed to $\mathcal{F}_{\mathrm{MN}}$. It is given a challenge ciphertext $c_l = \mathcal{E}_{pk}(M_b)$ for a randomly chosen $b \in \{0,1\}$ which it uses in the continued simulation.

By inspection we see that $\Pr[\mathsf{Exp}^{\mathsf{ind}-b}_{\mathcal{CS},\mathsf{Adv}_l}(\kappa) = 1] = \Pr[T_{l-b} = 1]$. The polynomial indistinguishability of the cryptosystem then implies that $|\Pr[T_l = 1] - \Pr[T_{l+1} = 1]|$ is negligible for $l = 1, \ldots, N$. The triangle inequality and the fact that N is polynomially bounded then implies that $|\Pr[T_0 = 1] - \Pr[T_N = 1]|$ is negligible as claimed. □

Informally speaking we have now plugged back the correct messages in the simulation. The problem is that decryption is still simulated incorrectly. In other words T_N is still not identically distributed T_{real}. To prove that the distributions are indistinguishable we need to sample the latter distribution without using the secret key of the cryptosystem. We do this using the knowledge extractors of the proofs of knowledge of correct obfuscation. One of the honest mix-servers must also simulate its proof of correct re-encryption and permutation of a matrix of ciphertexts.

Denote by B_b the simulation of T_N or T_{real} (depending on if $b = 0$ or not) except that in the simulation, if a corrupt mix-server $\mathcal{M}_j$ with $j \in I_{\mathcal{M}}$ succeeds in Protocol 6-1 or Protocol 6-2 then the knowledge extractor of the protocol is invoked to extract the witness. In the Paillier case we assume that in the same way the knowledge extractors of corrupt mix-servers are invoked in Protocol 6-3. Denote the maximal knowledge error of Protocol 6-1 and Protocol 6-2 by $\epsilon = \epsilon(\kappa)$, and recall that the knowledge error of a proof of knowledge does not depend on the adversary, but is a parameter of the protocol itself. Denote then by $t(\kappa)/(\delta - \epsilon)$, where $t(\kappa)$ is some polynomial, the expected running time of the extractor in any of these protocols for a prover with success probability δ, and recall that also $t(\kappa)$ is a parameter of the protocol. In the simulation carried out by B_b the running time of any extractor is restricted to $t(\kappa)/\epsilon$ and if extraction fails it outputs 0.

Claim 2 $|\Pr[B_0 = 1] - \Pr[T_N = 1]|$ *and* $|\Pr[B_1 = 1] - \Pr[T_{real} = 1]|$ *is negligible.*

Proof. First note that due to the soundness of the protocols, the probability that a mix-server succeeds to prove a false statement is negligible. Thus, we assume without loss that all statements for which the extractors are invoked there exists a witness.

Then consider the event E_l that in the lth proof computed by any mix-server $\mathcal{M}_j$ it succeeds with probability less than 2ϵ conditioned on the simulation up to this point, and still succeeds in the actual simulation. We clearly have $\Pr[E_l] < 2\epsilon$. The union bound then implies the claim, since there are at most $2k$ invocations of the protocols in total. □

Claim 3 *B_b runs in expected polynomial time.*

Proof. This follows, since at any instance where an extractor is invoked for a prover on some statement, if the prover succeeds with probability $\delta > 2\epsilon$, then the extractor runs in expected time at most $2t(\kappa)/\delta$ (although it may not output a witness at all), and otherwise the running time is bounded by $t(\kappa)/\epsilon$. Thus, in total this part of the simulation runs in expected time $2t(\kappa)$. As the extractors are only invoked polynomially many times, the complete simulation runs in expected polynomial time. □

Observe that B_1 can be sampled even without the secret key, since all the random permutations and all random exponents are extracted. More precisely, since the list of original inputs and how they are permuted (and inner-layer re-encrypted in the Paillier case), is known by the simulator it can simulate decryption perfectly. Assume from now on that this is done.

Denote by B'_b the simulation of B_b except for the following modification. Denote by $\mathcal{M}_l$ some fixed mix-server with $l \notin I_{\mathcal{M}}$. Instead of letting it execute the prover of Protocol 6-1, or Protocol 6-2 the honest verifier zero-knowledge simulator guaranteed to exist by Proposition 6-3 and Proposition 6-4 respectively is invoked by programming the coin-flipping functionality $\mathcal{F}_{\mathrm{CF}}$.

Claim 4 $|\Pr[B_b = 1] - \Pr[B'_b = 1]|$ *is negligible.*

Proof. This follows directly from the honest verifier zero-knowledge property of Protocol 6-1 and Protocol 6-2. □

BGN CASE. Simply write B''_0 instead of B'_0 to allow a single proof for the two cases (we are taking care about some special features of the Paillier case below).

PAILLIER CASE. In this case we need to take care of the fact that the inner re-encryption factors used by the simulator to simulate decryption are independently chosen from the true re-encryption factors generated in Protocol 6-3.

Denote by B''_0 the simulation of B'_0 except that the former uses the re-encryption factors extracted from the executions of Protocol 6-2.

Claim 5 (Paillier Case) $|\Pr[B'_0 = 1] - \Pr[B''_0 = 1]|$ *is negligible.*

Proof. Denote by Adv_{ind} the polynomial indistinguishability adversary that takes n as input and simulates B'_0 except that in Protocol 6-3 $\mathcal{M}_l$ computes its output c_j as follows. It generates two random lists $r_i^{(0)}, r_i^{(1)} \in (\mathbf{Z}_n^*)^N$ and hands these to the experiment. The experiment returns $c_j = \mathcal{E}_{n^3}^{\mathsf{pai}}(r_i^{(b)})$ for a random $b \in \{0,1\}$, which is used in the continued simulation. Then it defines $\bar{r}_i = r_i^{(1)} \prod_{j=l+1}^{k} r_{j,i}$, where $r_{j,i}$ are the values extracted by the knowledge extractors or chosen by simulated honest mix-servers. Note that if $b = 0$, the the simulation is identically distributed to B'_0 and otherwise statistically close distributed to B''_0.

The standard extension of polynomial indistinguishability to polynomial length lists of ciphertexts now implies the claim. □

At this point we have reduced the difference between B''_0 and B''_1 to how decryption is simulated. In the former decryption is simulated incorrectly by simply outputting the correct messages in some randomly chosen order, whereas in the latter the correspondence between individual ciphertexts and output messages is preserved.

Claim 6 $|\Pr[B''_0 = 1] - \Pr[B''_1 = 1]|$ *is negligible.*

Proof. Consider the following polynomial indistinguishability adversary Adv_{ind} to the obfuscation of the decryption/re-encryption shuffle. It accepts a public key pk as input. Then it simulates B_0'' until some fixed simulated honest mix-server $\mathcal{M}_l$ with $l \notin I_{\mathcal{M}}$ is about to produce its output in the mixnet.

The adversary Adv_{ind} chooses $\pi^{(0)}, \pi^{(1)} \in \Sigma_N$ randomly and defines $\pi_{sim} = \pi^{(1)}\pi_{l+1} \cdots \pi_k$, and $\pi_l = \pi^{(0)}$. Due to the group properties of Σ_N, this does not change the distribution of π_{sim} in either B_0'' or B_1''.

BGN CASE. The adversary Adv_{ind} hands $(DS^{\mathsf{bgn}}_{\pi^{(0)}}, DS^{\mathsf{bgn}}_{\pi^{(1)}})$ to the experiment and waits until it returns an obfuscation $\mathcal{O}(pk, sk, DS^{\mathsf{bgn}}_{\pi^{(b)}})$ for a random b. It extracts the encrypted permutation matrix $\tilde{\Lambda}^{\pi^{(b)}}$ from the obfuscated shuffle and uses it as $\mathcal{M}_l$'s output.

PAILLIER CASE. Denote by $r = (r_1, \ldots, r_N)$ the joint randomization values such that $\mathcal{D}^{\mathsf{pai}}_{p,3}(C_0)$ is the diagonal matrix with diagonal $(r_i^n \bmod n^2)$. These factors can be computed, from the witnesses extracted from the corrupt mix-servers in Protocol 6-2. The adversary Adv_{ind} hands $(RS^{\mathsf{pai}}_{\pi^{(0)},r}, RS^{\mathsf{pai}}_{\pi^{(1)},r})$ to the experiment and waits until it returns an obfuscation $\mathcal{O}(pk, sk, RS^{\mathsf{pai}}_{\pi^{(b)}})$ for a random b. It extracts the encrypted re-encryption and permutation matrix $\tilde{\Lambda}^{\pi^{(b)}}$ from the obfuscated shuffle and uses it as $\mathcal{M}_l$'s output.

We conclude that $\mathsf{Exp}^{\mathsf{oind}-b}_{RS^{\mathsf{pai}}_N, \mathsf{Adv}_{ind}}(\kappa)$ is identically distributed to B_b''. Then the claim follows from the polynomial indistinguishability of the cryptosystem, since an expected polynomial time distinguisher can be turned into a strict polynomial time distinguisher using Markov's inequality in the standard way. □

Conclusion of Proof of Proposition. To conclude the proof we simply note that Claim 1-6 implies that $|\Pr[T_{ideal}(\mathsf{Adv}) = 1] - \Pr[T_{real}(\mathsf{Adv}) = 1]|$ is negligible for any real adversary Adv.

6.11 Alternate Modeling using Obfuscation

We modeled our constructions in the public-key obfuscation model of Ostrovsky and Skeith [132], in large part because the indistinguishability property of this model maps quite well to the indistinguishability of our obfuscated shuffles: an adversary should be unable to determine any information about the permutation encoded in an obfuscated shuffle. In this section, we very briefly note that we could have modeled our construction in the obfuscation model of Barak et al. [12] and Goldwasser and Tauman-Kalai [78].

Deterministic Mixing. To perform obfuscation in the model of Barak et al. or Tauman-Kalai and Goldwasser, one must first define the exact functionality that will be obfuscated. In our case, the functionality must capture the shuffling actions of a mixnet, but also an additional property of determinism: the shuffle permutation is fixed ahead of time once and for all, and, given a particular sequence of input ciphertexts, the obfuscated functionality always produces the same exact set of outputs. We can call this a *deterministic mixing* functionality.

This functionality can be realized if we assume that key generation for the cryptosystem will be performed as part of the sampling of the functionality family. Then, a simple program can, given the public and secret keys (we are not worried about security yet, only whether this functionality can be realized to begin with) and a fixed permutation of size N:

1. decrypt the input ciphertexts

2. use a PRF (pseudo-random function) on the sequence of input ciphertexts to generate randomization factors

3. shuffle the decrypted plaintexts according to the fixed permutation, and encrypt them with the newly generated randomization factors.

4. when queried, provide either the permutation size N or the public key pk

A more advanced implementation could be achieved without the secret key, using a cryptosystem that supports reencryption. Again, the randomization values would be generated using a PRF on the sequence of input ciphertexts.

Finally, a last implementation can simply do exactly what we suggest in this work, in particular in the BGN case: generate a mixing matrix using only the permutation and some randomization factors, and perform the matrix multiplication to mix. In the "black-box" version of this implementation of the functionality, the matrix would remain "inside the black box," of course.

Security Property: Back to Indistinguishability! Given the above functionality definition of deterministic mixing, it is clear that we can obfuscate this functionality: simply output the matrix in question and let others perform the mixing. In fact, it is almost as if wrapping the matrix inside a black box functionality served only to prove the obfuscation property to begin with! Of course, the semantic security of the BGN cryptosystem guarantees computational obfuscation: a simulator can simply re-create a new encrypted matrix and pass it to the adversary. If the adversary can distinguish the two encrypted permutation matrices, then it has broken the cryptosystem's semantic security.

What is interesting here is that this secure obfuscation does not, in any way, imply that we have built a secure mixing functionality. The only thing we have shown is that some functionality—deterministic mixing—was properly obfuscated. We still have to show that deterministic mixing is secure to begin with. How can we define this security? It is likely that we must return to an indistinguishability definition!

In other words, proving obfuscation in this generic setting requires us to recreate the tools defined by Ostrovsky and Skeith for public-key obfuscation. This explains why we picked the model of Ostrovsky and Skeith to begin with.

6.12 Conclusion

It is surprising that a functionality as powerful as a shuffle can be public-key obfuscated in any useful way. It is even more surprising that this can be achieved using the Paillier cryptosystem which, in contrast to the BGN cryptosystem, was not specifically designed to have the kind of "homomorphic" properties we exploit. One intriguing question is whether other useful "homomorphic" properties have been overlooked in existing cryptosystems.

From a practical point of view we stress that, although the performance of our mixnet is much worse than that of known constructions, it exhibits a property which no previous construction has: a relatively small group of mix-servers can prepare obfuscated shuffles for voting precincts. The precincts can compute the shuffling without any private key and produce ciphertexts ready for decryption.

Bibliography

[1] Masayuki Abe. Universally verifiable MIX with verification work independent of the number of MIX servers. In Nyberg [126], pages 437–447.

[2] Masayuki Abe. Mix-networks on permutation networks. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *ASIACRYPT*, volume 1716 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 1999.

[3] ACM. *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing, 1990.* ACM, 1990.

[4] ACM. *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada.* ACM, 1994.

[5] ACM. *PODC 2001, Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, August 26-29, 2001, Newport, Rhode Island, USA.* ACM, 2001.

[6] Ben Adida and C. Andrew Neff. Ballot Casting Assurance. In *EVT '06, Proceedings of the First Usenix/ACCURATE Electronic Voting Technology Workshop, August 1st 2006, Vancouver, BC,Canada.*, 2006. Available online at http://www.usenix.org/events/evt06/tech/.

[7] Ben Adida and Douglas Wikström. How to Shuffle in Public. Cryptology ePrint Archive, Report 2005/394, 2006. http://eprint.iacr.org/2005/394.

[8] Alan A. Reiter. Hong Kong residents asked to photograph voting for pro-Beijing candidates, August 2004. http://www.wirelessmoment.com/2004/08/hong_kong_resid.html.

[9] Andrew Gumbel. *Steal This Vote: Dirty Elections and the Rotten History of Democracy in America.* Nation Books, July 2005.

[10] Ari Shapiro. Absentee Ballots Go Missing in Florida's Broward County, October 2004. http://www.npr.org/templates/story/story.php?storyId=4131522.

[11] Avi Rubin. An Election Day clouded by doubt, October 2004. http://avirubin.com/vote/op-ed.html.

[12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Kilian [108], pages 1–18.

[13] Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Guillaume Poupard. Practical multi-candidate election system. In *PODC* [5], pages 274–283.

[14] Donald Beaver. Foundations of secure interactive computing. In Feigenbaum [63], pages 377–391.

[15] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Nyberg [126], pages 236–250.

[16] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Santis [154], pages 92–111.

[17] Ben Adida and C. Andrew Neff. Assisted Human Interactive Proofs: A Formal Treatment of Secret Voter Receipts, 2006. In preparation.

[18] Ben Adida and Ronald L. Rivest. Scratch & Vote: Self-Contained Paper-Based Cryptographic Voting. In Roger Dingledine and Ting Yu, editors, *ACM Workshop on Privacy in the Electonic Society*. ACM, 2006. To Appear.

[19] Josh Benaloh and Moti Yung. Distributing the power of government to enhance the power of voters. In *PODC*, pages 52–62. ACM, 1986.

[20] Josh Cohen Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *STOC* [4], pages 544–553.

[21] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic Primitives Based on Hard Learning Problems. In Stinson [165], pages 278–291.

[22] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC*, pages 103–112. ACM, 1988.

[23] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–342. Springer, 2005.

[24] Dan Boneh, editor. *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*. Springer, 2003.

[25] Dan Boneh and Matthew K. Franklin. Efficient Generation of Shared RSA Keys (Extended Abstract). In Kaliski [103], pages 425–439.

[26] Dan Boneh and Philippe Golle. Almost entirely correct mixing with applications to voting. In Vijayalakshmi Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 68–77. ACM, 2002.

[27] Stefan Brands. Untraceable off-line cash in wallets with observers (extended abstract). In Stinson [165], pages 302–318.

[28] C. Andrew Neff. Codebooks. Unpublished Manuscript.

[29] C. Andrew Neff. Practical High Certainty Intent Verification for Encrypted Votes. `http://votehere.com/vhti/documentation/vsv-2.0.3638.pdf`, last viewed on August 30th, 2006.

[30] C. Andrew Neff. Coerced Randomization, April 2006. Private conversations with and unpublished manuscript by Andy Neff.

[31] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In Kaliski [103], pages 455–469.

[32] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA, Proceedings*, pages 136–145. IEEE Computer Society, 2001. (Full version at Cryptology ePrint Archive, Report 2000/067, `http://eprint.iacr.org`, October, 2001.).

[33] Ran Canetti and Rosario Gennaro. Incoercible multiparty computation (extended abstract). In *37th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1996), 1996, Proceedings*, pages 504–513. IEEE Computer Society, 1996.

[34] CBS News. How to Spend a Billion Dollars, May 2004. `http://www.cbsnews.com/stories/2004/05/12/politics/main617059.shtml`.

[35] CBS News. Switzerland Tries Internet Voting, September 2004. `http://www.cbsnews.com/stories/2004/09/25/world/main645615.shtml`.

[36] Charles Stewart III and Julie Brogan. Voting in Massachusetts, August 2003. `http://www.vote.caltech.edu/media/documents/VotinginMass.pdf`.

[37] D. Chaum and T. Pedersen. Wallet Databases with Observers. In Ernest F. Brickell, editor, *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1993.

[38] David Chaum. Punchscan. `http://punchscan.org` viewed on August 13th, 2006.

[39] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[40] David Chaum. Secret-Ballot Receipts: True Voter-Verifiable Elections. *IEEE Security and Privacy*, 02(1):38–47, 2004.

[41] David Chaum, Peter Y. A. Ryan, and Steve Schneider. A practical voter-verifiable election scheme. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 118–139. Springer, 2005.

[42] Richard Clayton. Improving onion notation. In Roger Dingledine, editor, *Privacy Enhancing Technologies*, volume 2760 of *Lecture Notes in Computer Science*, pages 81–87. Springer, 2003.

[43] Josh D. Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme. In *FOCS*, pages 372–382. IEEE Computer Society, 1985.

[44] Ronald Cramer, Ivan Damgard, and Berry Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In Desmedt [52], pages 174–187.

[45] Ronald Cramer, Matthew Franklin, Berry Schoenmakers, and Moti Yung. Multi-authority Secret-Ballot Elections with Linear Work. In Ueli M. Maurer, editor, *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1996.

[46] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A Secure and Optimally Efficient Multi-Authority Election Scheme. In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 481–490. Springer, 1997.

[47] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In Feigenbaum [63], pages 445–456.

[48] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[49] Ivan Damgård and Maciej Koprowski. Practical Threshold RSA Signatures without a Trusted Dealer. In Pfitzmann [137], pages 152–165.

[50] David Jefferson, Aviel D. Rubin, Barbara Simons, David Wagner. A Security Analysis of the Secure Electronic Registration and Voting Experiment (SERVE). `http://www.servesecurityreport.org/`.

[51] David Wagner. Written Testimony Before the Committee on Science and Committee on House Administration U.S. House of Representatives, July 2006. `http://www.house.gov/science/hearings/full06/July%2019/Wagner.pdf`.

[52] Yvo Desmedt, editor. *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*. Springer, 1994.

[53] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1990.

[54] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures (extended abstract). In Feigenbaum [63], pages 457–469.

[55] Yvo Desmedt and Kaoru Kurosawa. How to break a practical mix and design a new one. In Preneel [139], pages 557–572.

[56] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[57] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[58] Douglas Jones. Parallel Testing: A menu of options, August 2004. `http://www.cs.uiowa.edu/~jones/voting/miamiparallel.pdf`.

[59] Stefan Droste. New results on visual cryptography. In Koblitz [109], pages 401–415.

[60] Quang Viet Duong and Kaoru Kurosawa. Almost ideal contrast visual cryptography with reversing. In Okamoto [131], pages 353–365.

[61] Election Data Services. New Study Shows 50 Million Voters Will Use Electronic Voting Systems, 32 Million Still with Punch Cards in 2004, February 2004. `http://www.electiondataservices.com/EDSInc_VEstudy2004.pdf`.

[62] Uriel Feige and Adi Shamir. Witness indistinguishable and witness hiding protocols. In *STOC* [3], pages 416–426.

[63] Joan Feigenbaum, editor. *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*. Springer, 1992.

[64] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–437. IEEE Computer Society, 1987.

[65] Amos Fiat and Adi Shamir. How to prove yourself. practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–189. Springer, 1987.

[66] Kevin Fisher, Richard Carback, and Alan Sherman. Punchscan: Introduction and System Definition of a High-Integrity Election System. In Peter A. Ryan, editor, *Proceedings of the IAVoSS Workshop On Trustworthy Elections (WOTE'06)*, Cambridge, UK, June 2006.

[67] Florida Department of State. Official Results of the November 7, 2000 General Election, 2000. `http://election.dos.state.fl.us/elections/resultsarchive/SummaryRpt.asp?ElectionDate=11/7/2000&Race=PRE&DATAMODE=`.

[68] Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. Sharing decryption in the context of voting or lotteries. In Yair Frankel, editor, *Financial Cryptography*, volume 1962 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2001.

[69] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed RSA-key generation. In *STOC*, pages 663–672. ACM, 1994.

[70] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In Kilian [108], pages 368–387.

[71] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Trans. Inform. Theory*, 31:469–472, 1985.

[72] Rosario Gennaro. Achieving independence efficiently and securely. In *PODC*, pages 130–136. ACM, 1995.

[73] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust and Efficient Sharing of RSA Functions. In Koblitz [109], pages 157–172.

[74] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In Stern [164], pages 295–310.

[75] Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, Cambridge, UK, 2001.

[76] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.

[77] Shafi Goldwasser and Yael Tauman Kalai. On the (In)security of the Fiat-Shamir Paradigm. In *FOCS*, pages 102–. IEEE Computer Society, 2003.

[78] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS*, pages 553–562. IEEE Computer Society, 2005.

[79] Shafi Goldwasser and Leonid A. Levin. Fair computation of general functions in presence of immoral majority. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1991.

[80] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *STOC '82: Proceedings of the Fourteenth Annual ACM symposium on Theory of Computing*, pages 365–377. ACM, 1982.

[81] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[82] Philippe Golle, Markus Jakobsson, Ari Juels, and Paul F. Syverson. Universal re-encryption for mixnets. In Okamoto [131], pages 163–178.

[83] Philippe Golle and Ari Juels. Parallel mixing. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick Drew McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 220–226. ACM, 2004.

[84] Philippe Golle, Sheng Zhong, Dan Boneh, Markus Jakobsson, and Ari Juels. Optimistic mixing for exit-polls. In Yuliang Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2002.

[85] T. Granlund. GNU Multiple Precision Arithmetic Library (GMP). Software available at `http://swox.com/gmp`, March 2005.

[86] Dimitris Gritzalis, editor. *Secure Electronic Voting.* Kluwer Academic Publishers, 2002.

[87] Jens Groth. A verifiable secret shuffle of homomorphic encryptions. In Yvo Desmedt, editor, *Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2002.

[88] Louis C. Guillou and Jean-Jacques Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In C. G. Günther, editor, *EUROCRYPT*, volume 330 of *Lecture Notes in Computer Science*, pages 123–128. Springer, 1988.

[89] Jonathan Herzog, Moses Liskov, and Silvio Micali. Plaintext awareness via key registration. In Boneh [24], pages 548–564.

[90] Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In Preneel [139], pages 539–556.

[91] Nicholas J. Hopper and Manuel Blum. Secure human identification protocols. In Colin Boyd, editor, *ASIACRYPT*, volume 2248 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2001.

[92] Markus Jakobsson. A practical mix. In Nyberg [126], pages 448–461.

[93] Markus Jakobsson. Flash mixing. In *PODC*, pages 83–89. ACM, 1999.

[94] Markus Jakobsson and Ari Juels. Millimix: Mixing in small batches. Technical Report 99-33, Center for Discrete Mathematics & Theoretical Computer Science (DIMACS), 1999.

[95] Markus Jakobsson and Ari Juels. An optimally robust hybrid mix network. In *PODC* [5], pages 284–292.

[96] Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In Dan Boneh, editor, *USENIX Security Symposium*, pages 339–353. USENIX, 2002.

[97] James Robinson and Jean-Marie Baland. Land and Power: Theory and Evidence, 2005. `http://repositories.cdlib.org/berkeley_econ211/spring2005/16/`.

[98] John Wack. Voter Verified Paper Audit Trail Update, March 2006. `http://vote.nist.gov/032906VVPAT-jpw.pdf`.

[99] Douglas W. Jones. A Brief Illustrated History of Voting, 2001-2003. `http://www.cs.uiowa.edu/~jones/voting/pictures/`.

[100] Joseph P. Harris. *Election Administration in the United States.* Brookings Institute Press, 1934.

[101] Robert F. Kennedy Jr. Was the 2004 Election Stolen?, June 2006. In Rolling Stone Magazine.

[102] Ari Juels and Stephen A. Weis. Authenticating Pervasive Devices with Human Protocols. In Shoup [162], pages 293–308.

[103] Burton S. Kaliski, editor. *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*. Springer, 1997.

[104] Chris Karlof, Naveen Sastry, and David Wagner. Cryptographic voting protocols: A systems perspective. In *USENIX Security Symposium*, pages 33–50. USENIX, 2005.

[105] Jonathan Katz, Steven Myers, and Rafail Ostrovsky. Cryptographic counters and applications to electronic voting. In Pfitzmann [137], pages 78–92.

[106] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 2002.

[107] Aggelos Kiayias and Moti Yung. The vector-ballot e-voting approach. In Ari Juels, editor, *Financial Cryptography*, volume 3110 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2004.

[108] Joe Kilian, editor. *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*. Springer, 2001.

[109] Neal Koblitz, editor. *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996.

[110] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. On the composition of authenticated byzantine agreement. In *STOC*, pages 514–523. ACM, 2002.

[111] Lynn Landes. The Nightmare Scenario Is Here - Computer Voting With No Paper Trail, August 2002. `http://www.ecotalk.org/Dr.RebeccaMercuriComputerVoting.htm`.

[112] Lynn Landes. Scrap the secret ballot - return to open voting, November 2005. `http://www.opednews.com/articles/opedne_lynn_lan_051104_scrap_the__secret__b.htm`.

[113] Marsha Walton. Voting methods under close watch, October 2004. `http://www.cnn.com/2004/TECH/10/26/evote/index.html`.

[114] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[115] Rebecca Mercuri. Voting-machine risks. *Commun. ACM*, 35(11):138, 1992.

[116] Silvio Micali and Phillip Rogaway. Secure computation (abstract). In Feigenbaum [63], pages 392–404.

[117] Michael Shamos. Paper v. Electronic Voting Records - An Assessment, April 2004. `http://www.electiontech.org/downloads/Paper%20vs%20Electronic.pdf`.

[118] Markus Michels and Patrick Horster. Some remarks on a receipt-free and universally verifiable mix-type voting scheme. In Kwangjo Kim and Tsutomu Matsumoto, editors, *ASIACRYPT*, volume 1163 of *Lecture Notes in Computer Science*, pages 125–132. Springer, 1996.

[119] Masashi Mitomo and Kaoru Kurosawa. Attack for flash mix. In Okamoto [130], pages 192–204.

[120] Moni Naor and Benny Pinkas. Visual authentication and identification. In Kaliski [103], pages 322–336.

[121] Moni Naor and Adi Shamir. Visual cryptography. In Santis [154], pages 1–12.

[122] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *STOC* [3], pages 427–437.

[123] National Transportation Safety Board. `http://www.ntsb.gov/`.

[124] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *ACM Conference on Computer and Communications Security*, pages 116–125. ACM, 2001.

[125] Valtteri Niemi and Ari Renvall. How to prevent buying of votes in computer elections. In Josef Pieprzyk and Reihaneh Safavi-Naini, editors, *ASIACRYPT*, volume 917 of *Lecture Notes in Computer Science*, pages 164–170. Springer, 1995.

[126] Kaisa Nyberg, editor. *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*. Springer, 1998.

[127] Wakaha Ogata, Kaoru Kurosawa, Kazue Sako, and Kazunori Takatani. Fault tolerant anonymous channel. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *ICICS*, volume 1334 of *Lecture Notes in Computer Science*, pages 440–444. Springer, 1997.

[128] Miyako Ohkubo and Masayuki Abe. A length-invariant hybrid mix. In Okamoto [130], pages 178–191.

[129] Tatsuaki Okamoto. Receipt-free electronic voting schemes for large scale elections. In Bruce Christianson, Bruno Crispo, T. Mark A. Lomas, and Michael Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 1998.

[130] Tatsuaki 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.

[131] Tatsuaki Okamoto, editor. *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*. Springer, 2004.

[132] Rafail Ostrovsky and William E. Skeith III. Private Searching on Streaming Data. In Shoup [162], pages 223–240.

[133] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Stern [164], pages 223–238.

[134] Choonsik Park, Kazutomo Itoh, and Kaoru Kurosawa. Efficient anonymous channel and all/nothing election scheme. In Tor Helleseth, editor, *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 248–259. Springer, 1994.

[135] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In Donald W. Davies, editor, *EUROCRYPT*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer, 1991.

[136] Birgit Pfitzmann. Breaking efficient anonymous channel. In Santis [154], pages 332–340.

[137] Birgit Pfitzmann, editor. *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science.* Springer, 2001.

[138] Birgit Pfitzmann and Andreas Pfitzmann. How to break the direct rsa-implementation of mixes. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT*, volume 434 of *Lecture Notes in Computer Science*, pages 373–381. Springer, 1990.

[139] Bart Preneel, editor. *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science.* Springer, 2000.

[140] Charles Rackoff and Daniel R. Simon. Cryptographic defense against traffic analysis. In *STOC*, pages 672–681. ACM, 1993.

[141] Ralph C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.

[142] Hugo Krawczyk Ran Canetti and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In Boneh [24], pages 565–582.

[143] Randal C. Archibold. Arizona Ballot Could Become Lottery Ticket, July 2006. `http://www.nytimes.com/2006/07/17/us/17voter.html?ex=1310788800&en=9626060428eeb1ed&ei=5088&partner=rssnyt&emc=rss`.

[144] Brian Randell and Peter Y. A. Ryan. Voting technologies and trust. In Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2006.

[145] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[146] Robert Simeon Fay. Card trick. `http://www.caveofmagic.com/pickcrd2.htm`, visited on August 23rd 2006.

[147] Ronald L. Rivest. Remarks on The Technologies of Electronic Voting, Harvard University's Kennedy School of Government Digital Voting Symposium. `http://theory.lcs.mit.edu/~rivest/2004-06-01%20Harvard%20KSG%20Symposium%20Evoting%20remarks.txt`.

[148] Roy G. Saltman. *The History and Politics of Voting Technology.* Palgrave Macmillan, 2006.

[149] Peter Y.A. Ryan and Steve A. Schneider. Prêt à voter with re-encryption mixes. In *ESORICS*, Lecture Notes in Computer Science. Springer, 2006. To Appear.

[150] Kazue Sako and Joe Kilian. Secure voting using partially compatible homomorphisms. In Desmedt [52], pages 411–424.

[151] Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme - a practical solution to the implementation of a voting booth. In Louis C. Guillou and Jean-Jacques Quisquater, editors, *EUROCRYPT*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 1995.

[152] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[153] San Diego Elections Task Force. Elections Task Force Recommendation: Mail-Only Ballot Election, July 2006. `http://www.sandiego.gov/electionstaskforce/pdf/reports/mail_balloting_report.pdf`.

[154] Alfredo De Santis, editor. *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science.* Springer, 1995.

[155] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *STOC* [4], pages 522–533.

[156] Bruce Schneier. Crypto-gram newsletter, February 2001. `http://www.schneier.com/crypto-gram-0102.html#10`.

[157] Claus P. Schnorr. Efficient identification and signatures for smartcards. In Feigenbaum [63], page 239.

[158] Scratch 'N Win Ballots To Debut In November, July 2006. `http://www.theonion.com/content/node/50640`.

[159] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[160] Daniel Shanks. Class number, a theory of factorization, and genera. In *Number Theory Institute, 1969*, volume 20 of *Proceedings of Symposia in Pure Mathematics*, pages 415–440. American Math Society, 1969.

[161] Victor Shoup. Practical threshold signatures. In Preneel [139], pages 207–220.

[162] Victor Shoup, editor. *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*. Springer, 2005.

[163] Stephanie Chasteen. Electronic voting unreliable without receipt, expert says, February 2004. `http://news-service.stanford.edu/news/2004/february18/aaas-dillsr-218.html`.

[164] Jacques Stern, editor. *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*. Springer, 1999.

[165] Douglas R. Stinson, editor. *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*. Springer, 1994.

[166] Ted Selker. Testimony on voter verification presented to Senate Committee on Rules and Administration, July 2005. `http://rules.senate.gov/hearings/2005/Selker062105.pdf`.

[167] The Barcode Software Center. PDF-417, 2003. `http://www.mecsw.com/specs/pdf417.html`.

[168] Yiannis Tsiounis and Moti Yung. On the security of elgamal based encryption. In Hideki Imai and Yuliang Zheng, editors, *Public Key Cryptography*, volume 1431 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 1998.

[169] Verified Voting Foundation. `http://verifiedvoting.org`.

[170] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: Using Hard AI Problems for Security. In Eli Biham, editor, *EUROCRYPT*, volume 2656 of *Lecture Notes in Computer Science*, pages 294–311. Springer, 2003.

[171] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, 2004.

[172] The Vote By Mail Project. `http://www.votebymailproject.org/`.

[173] Hoeteck Wee. On obfuscating point functions. In Harold N. Gabow and Ronald Fagin, editors, *STOC*, pages 523–532. ACM, 2005.

[174] Wikipedia. 2D Barcode. `http://en.wikipedia.org/wiki/2D_barcode`.

[175] Wikipedia. United States Presidential Election, 2000. `http://en.wikipedia.org/wiki/U.S._presidential_election,_2000`, viewed on June 6th, 2006.

[176] Wikipedia. Ostracon — wikipedia, the free encyclopedia, 2006. [Online; accessed 28-July-2006].

[177] Wikipedia. Silk Air 185, June 2006. `http://en.wikipedia.org/wiki/Silkair_Flight_185`.

[178] Douglas Wikström. Five practical attacks for "optimistic mixing for exit-polls". In Mitsuru Matsui and Robert J. Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2004.

[179] Douglas Wikström. A universally composable mix-net. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 317–335. Springer, 2004.

[180] Douglas Wikström. A sender verifiable mix-net and a new proof of a shuffle. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2005.

[181] Douglas Wikström and Jens Groth. An adaptively secure mix-net without erasures. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 276–287. Springer, 2006.

[182] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.