MPF (Mental Poker Framework) A new family of practical and secure Mental Poker protocols Tesis de Licenciatura

Autor: Sergio Demian Lerner
Director: Dr. Hugo D. Scolnik

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Noviembre de 2010

Agradecimientos

A mi esposa Alush, que me alentó siempre a seguir mi camino.
A mi hijo Ariel, por su infinita impaciencia.
A mis padres, por enseñarme a aprender.
A mi director, Dr. Hugo Scolnik, por darme la oportunidad y ayudarme.
A los profesores de Exactas, que dan lo mejor de sí para engrandecer día a día la facultad.
A mis amigos y en especial a Diego Bursztyn, por el puntapié inicial.

Por último a Bruce Schneier, autor del primer libro sobre criptografía que cayó en mis manos.

Table of Contents

Abstract 6
List of Figures 7
List of Tables 7
1. Introduction 8
  1.1. Background 9
  1.2. Mental Protocol Requirements 9
    1.2.1. Basic Requirements 9
    1.2.2. Extended Requirements 11
    1.2.3. Unreal Requirements 12
  1.3. Measuring Performance 13
  1.4. Delayed Verifications 13
  1.5. Non-interactive vs. Interactive Proofs 13
  1.6. External on-line Auditing 14
  1.7. Hash Chains 14
  1.8. Theoretical Security in Mental Poker Protocols 14
  1.9. Kinds of Proofs 14
2. MPF base concepts 16
  2.1. MPF base protocols 18
  2.2. MPF instantiation 18
  2.3. Verification Protocols 18
  2.4. Unified Verification protocol (UniVP) 20
  2.5. Ad-hoc Verification protocols and malleability 22
  2.6. MPF compared to SRA 23
  2.7. MPF compared to Barnett-Smart 23
  2.8. Keys Lifetime 23
  2.9. Open Cards Lifetime 23
  2.10. Encoding of Open-Cards 24
    2.10.1. VRF-based CO-PRNGP 24
    2.10.2. Hash-based CO-PRNGP 24
  2.11. MPF Rounds 26
  2.12. Free Cards 27
  2.13. Card Keys 28
  2.14. Card Dealing 28
  2.15. Master Card Keys 29
  2.16. Unverified Computations 29
  2.17. Card Transfers 29
  2.18. Suicide Cheaters 29
  2.19. VSM-VPUM trick 30
  2.20. VSM-L-OL, The Fastest Dealing Protocol 30
  2.21. Card Deal Preparation Phase 31
  2.22. Card showdowns 31
  2.23. Advanced Card Operations 31
  2.24. Abrupt Drop-out tolerance 31
  2.25. Duplicated Cards 33
  2.26. Some known CGCs 34
3. MPF formal definition 35
  3.1. Definitions 35
  3.2. Types 35
  3.3. Private Data Structures 35
  3.4. Miscellaneous Operations 36
  3.5. Operations on Cards 37
  3.6. Introduction to MPF base protocols 37
    3.6.1. VSM-L-OL 38
    3.6.2. VSM-VL 39
    3.6.3. VSM-VPUM 40
    3.6.4. VSM-VL-VUM 41

3.7. Card Protocols 42
3.7.1. Create-Deck (CO-PRNGP) 43
3.7.2. Create-Deck (Locking) 44
3.7.3. Shuffle-Deck 44
3.7.4. Prepare-Cards-To-Deal (for VSM-L-OL) 44
3.7.5. Prepare-Cards-To-Deal (for VSM-VL) 45
3.7.6. Prepare-Cards-To-Deal (for VSM-VPUM) 45
3.7.7. Prepare-Cards-To-Deal (for VSM-VL-VUM) 45
3.7.8. Verified-Unmasking-Round 46
3.7.9. Verified-ShuffleMasking-Round 46
3.7.10. Locking-Round 47
3.7.11. Shuffle-Hand(1) 47
3.7.12. Shuffle-Hand(2) 48
3.7.13. Multiple-Cards-Deal (for VSM-VPUM) 49
3.7.14. Single-Card-Deal (for VSM-L-OL, VSM-VL and VSM-VL-VUM) 49
3.7.15. Card-Key-Verification (for VSM-VL-VUM) 50
3.7.16. Card-Key-Verification (for VSM-VL and VSM-L-OL) 50
3.7.17. Build-Representative 50
3.7.18. Show-Cards(1) 51
3.7.19. Show-Cards(2) 51
3.7.20. Reshuffle-Deck (for VSM-L-OL, VSM-VL and VSM-VL-VUM) 52
3.7.21. Change-Hand-Cards-Key 52
3.7.22. Return-Cards-To-Deck (for VSM-L-OL, VSM-VL and VSM-VL-VUM) 53
3.7.23. Private-Cards-Transfer 53
3.7.24. Abrupt-Drop-out-Recovery (for VSM-VL and VSM-VL-VUM) 54
3.7.25. Create-Recovery-Sets 54
3.7.26. Put-Card-On-Table 55
3.7.27. Verified-ShuffleRemasking-Round 55
3.7.28. End-Of-Game (only for VSM-L-OL) 56
4. Sample VSM-VL Run for Texas Hold'em 57
5. Pre-defined UniVPIs 58
5.1. Protocol FI-UniVP 59
5.2. Protocol FIG-UniVP 62
5.3. Protocol I-UniVPI 63
5.4. Protocol NI-UniVPI 64
5.5. Protocol CO-VP 65
6. Security of MPF 66
6.1. Example Attacks 67
6.1.1. Tracking attack 67
6.1.2. Sandwich tracking attack against CO-VP 67
7. Detailed Comparison of Mental Poker Protocols 68
8. PHMP (MPF VSM-VL with Pohlig-Hellman as CGC) 69
8.1. Definition of E 69
8.2. Cipher Parameters Creation 69
8.3. Finding fixed generators 70
8.4. Card Creation 70
8.5. Using quadratic residue cards 70
8.5.1. Finding a random generator of the QR subgroup 70
8.5.2. Key Creation 70
8.6. Using quadratic non-residue cards 70
8.6.1. Finding a generator of (Z/pZ)* 70
8.6.2. Key Creation 71
8.7. Additional checks 71
8.8. Ad-hoc verification protocols 71
8.8.1. HMVP 72
8.8.2. Schnorr's Id Protocol based LVP (S-LVP) 73
8.8.3. Chaum-Pedersen based UMVP (CP-UMVP) 73
8.8.4. CO-SMVP 74
8.9. Numeric Example 75
9. Performance 79
10. Open Questions & Conclusions 81

11. Bibliography 82

Abstract

Mental Poker (MP) protocols allow multiple parties to securely play a game remotely with a virtual deck of cards using cryptography. A Mental Poker protocol can also be used to execute more complex protocols, such as a secure multi-party boolean circuit evaluation and e-voting. In this thesis we propose a new family of practical and secure mental poker protocols, which includes four competing protocols. For comparison, we have implemented and tested one protocol based on the Pohlig-Hellman symmetric cipher and also we have theoretically evaluated the performance of an elliptic-curve based implementation. Both resulting protocols are efficient enough to achieve real-time performance to play Texas Hold-em over the Internet, for up to ten players, using personal computers and keys of 1024 bits long (PH) and 160 bits long (ECC). To the best of our knowledge, this is the first protocol that achieves this milestone.

List of Figures

Figure 1: Players and Agents....8
Figure 2: Hash-based CO-PRNGP....25
Figure 3: Shuffle-Masking round....26
Figure 4: Locking round....27
Figure 5: Undo round....27
Figure 6: Key share disclosure method to deal a card....28
Figure 7: Partial undo method to deal a card....29
Figure 8: Sample execution of Abrupt Drop-out-recovery protocol....33
Figure 9: VSM-L-OL protocol....38
Figure 10: VSM-VL protocol....39
Figure 11: VSM-VL shuffle between agents....40
Figure 12: VSM-VPUM protocol....40
Figure 13: VSM-VL-VUM protocol....41
Figure 14: Execution of all the rounds under the VSM-VL-VUM protocol....41
Figure 15: Protocol FI-UniVP....59

List of Tables

Table 1: Meaning of Letters in Protocol Names....18
Table 2: Differences between verification Protocols....20
Table 3: UniVP Arguments....21
Table 4: Argument modifiers reference....42
Table 5: Card Protocols Quick Reference....43
Table 6: External Functions....58
Table 7: Operations in U, S, R and T boxes....60
Table 8: Comparison of Mental Poker Protocols....68
Table 9: Alternatives for predefined UniVPs Depending on the Type of Verification....71
Table 10: Simulation Scenario....79
Table 11: Comparison of Protocol Times....80

1. Introduction

In this thesis we propose a new family of Mental Poker protocols. The motivation behind the development of such protocols can be explained with an example. Suppose a group of people around the world decide to play a poker game. Each of the players has a computer connected to the Internet and all communications between computers are authenticated and encrypted. Also suppose that all computers are secure and tamper-proof. The problem is that they are not willing to trust an arbiter to deal the cards, because they don´t have any trustworthy person in common. How can they manage to securely play the game?

Mental Poker (MP) protocols solve this and many other related problems, allowing multiple parties to securely handle a deck of virtual cards over a secure peer-to-peer or broadcast medium. The protocol can be a card game such as poker, or a multi-party computation that relies on virtual cards, such as a secure e-voting or secure circuit evaluation schemes. Because of this diversity of applications, there is no uniform nomenclature on the research community. For example, a secure voting ballot may support similar operations and so be homomorphic to a poker card. We'll use the card game nomenclature. We assume “card” and “information token” are synonymous, and also the terms “party” and “player”.
To execute a mental poker protocol, players are required to do computations. In order to achieve state of the art security, computations involve large numbers, are very hard to do by hand, and almost impossible to do mentally, so each player relies on a programmed computer device that compute on his behalf. This device not only does a player own computations, but validates the other players computations to detect cheating. We'll call each device an “agent” in the protocol. Fig. 1 depicts the configuration.

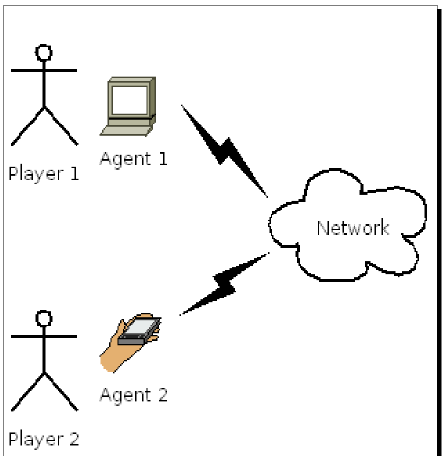

Figure 1: Players and Agents

Mental Poker protocols can be divided in two main groups: those that require a trusted third party (TTP) and protocols that do not (TTP-Free). In the former there is a third party who draws and knows the cards in each player's hand, so it should be trusted and impartial. From the point of view of a poker player, an e-gaming site that uses an MP protocol with TTP poses no additional security than any a site secured by standard methods. In TTP-free protocols, only each player knows his hand, and not even the site operator is able to see players cards. In the context of a voting scheme, a protocol with TTP means that a central office or the collusion of government offices can recover enough information to trace each vote to its voter, and so only a TTP-free protocol can provide true anonymity to individual voters.

In this thesis we define MPF, a framework that allows the creation of practical and secure TTP-free mental poker protocols. MPF defines classes and secure operations that rely on a user-provided commutative group cryptosystem. The framework also provides four base protocols, and other user-selectable options for time/security trade-offs. We also describe a specific protocol of the family over the Pohlig-Hellman symmetric cipher (PHMP). PHMP is efficient enough to archive real-time performance to play Texas Hold-em over the Internet, for up to ten players, using personal computers and keys of 1024 bits long. Compared with previously proposed protocols computation and communication bandwidth requirements of our proposal are far lower.

1.1. Background

Shamir, Rivest and Adleman proposed the first TTP-Free protocol [SRA81] that achieved some of the properties desired for card games, but forced the players to reveal their hands and their strategy at the end of the game. In [Cr86] a standard set of requirements for a MP protocol was establish. If a protocol satisfied these requirements, it would be as secure as a “real” card game. [Cr86] also presents the first protocol that satisfies them. However the protocol is not practical, since an implementation is reported to take 8 hours to shuffle a poker deck [E94]. Obviously the requirement that the protocol should be real-time or at least practical, was not listed.

New protocols were later developed [KKO90][BS03][CDRB03][CR05]. With the exception of [BS03], we've found these protocols use very complex cryptographic constructions such as ANDOS (Anonymous Disclosure of Secrets) and they present many drawbacks. They are difficult to formally verify, to extend and to implement. The main reason is that the papers are mathematical oriented and not system engineer oriented. Protocol descriptions lack abstract layers, abstract data types, and a simple set of operations on cards. Too much is left to the implementor. This was also a motivation to create a coherent and simple protocol that can be studied both from the mathematical and the practical points of view. Recently, and in parallel with the development of MPF, a new protocol has been published [WW09] that is based on the same security assertions than MPF.

In addition to the MP requirements defined in [Cr86], in [CDRB03] it is noted that these requirements are not enough for protocols to withstand real-world scenarios and define a new requirement: drop-out tolerance. Drop-out tolerance is, as we shall see, an unavoidable requirement for a successful on-line system. In this thesis, we define four additional requirements for a protocol to withstand real-world scenarios, two of them which are generally considered, but not always stated, and two additional requirements to allow playing more complex card games.

1.2. Mental Protocol Requirements

We present three sets of requirements. A protocol that satisfies the **basic requirements** is theoretically sound. A protocol that satisfies the **extended requirements** can be used in practice with current home computer technology. A protocol that satisfies the **unreal requirements** allows players to play games which are practically or theoretically impossible to play with a real card deck without a TTP.

1.2.1. Basic Requirements

These are the requirements for a Mental Poker protocol defined in [Cr86] :

R1. Uniqueness of cards: no player alone can duplicate cards. Duplicates should be detected and the cheater identified.

R2. Uniform random distribution of cards: no player can predict or change the outcome of the card shuffle.

R3. Cheating detection with a very high probability: The protocol must detect any attempt to cheat. Cheating is usually detected in a protocol stage in which each player show the others a proof of the correctness of the computations he has done. Each player must prove he has act honestly while performing computations. A player who cannot prove honesty is a cheater. Because honesty proofs are generally the most computational expensive operation in MP protocols, some protocols allow to set a certain threshold which determines the probability of a cheating party go undetected.

R4. Complete confidentiality of cards: No information of the cards in the deck or the cards in player's hand should leak. In practice, we only require that the leakage is small enough to be practically infeasible to guess the opponent's cards or cards in the deck (obtaining statistically significant information must be a computationally hard problem).

R5. Minimal effect of coalitions: Players should no be able to collude and work together to change the outcome of the shuffling, nor to exchange cards privately, nor cheat undetected or chat and put the blame on a third player.

R6. Complete confidentiality of strategy: Players who decide to show their hand should have guaranteed that no strategic information will leak, e.g. the time when each shown card was dealt. Also, players who opt no to show their cards shouldn't be required to disclose any information.

R7. No requirement for a trusted third party: In real-life scenarios, card games are played for money, so players would probably not rely on a trusted third party who knows the cards during play. A third party could be bribed, or his computer system could be hacked.

The two following requirements, were proposed in [CSD05], along with the first abrupt drop-out tolerant protocol.

R8. Polite Drop-out tolerance: If a player decides to quit the game politely, the remaining players should be able to keep playing. They should be able to return the to deck the cards that were in the hand of the player who dropped out, or put them apart.

R9. Abrupt Drop-out tolerance: If a player abruptly quits the game (e.g. by closing his connection) the remaining players should be able to keep playing. Cards that were in the hand of the player having dropped out should be returned to the deck.

Perfect abrupt drop out tolerance cannot theoretically be achieved without a third party. There are two theoretical limitations. The first limitation comes from collusion (Drop-Out-Collusion problem) and the second, because common knowledge cannot be gained in a distributed system with unreliable messages (Semi-open card problem). **Drop-out collusion** is the situation when a player quits the game to force the other players to put back his hand cards in the deck, and then transmit any private knowledge to a colluding player who is still on-line. **Semi-open cards** are cards that cannot be proven to have been opened and seen by a certain player nor have remained unseen in the deck. We show an example situation where both problem arises, during a card deal. Suppose that a group of players is playing a card game over the Internet. The players want to deal a card C to a player Marvin. Let's define a hypothetical protocol to do it:

1. Each player, with the exception of Marvin, publish a key share for the card C.
2. Then Marvin joins all the key shares, including her own private key share, and reconstructs the whole key K.
3. Marvin decrypts the card C wit the key K.

Suppose that Mallory is a player that colludes with Marvin, and instead of publishing his key share (step 1), he interrupts his Internet connection, and privately sends his key share to Marvin. Also he privately sends the values of his own cards. All the remaining players conclude that Mallory is not longer available and try to recover from a drop out. Because they do not trust Marvin more than Mallory, they have no way of knowing if Marvin has received the message from Mallory so they don't know if Marvin has been able to decrypt the card C or if he hasn't. In the former case, they should keep playing without dealing an additional card to Marvin. In the later, they should deal a new card to Marvin, which seem to be the wisest thing to do. Suppose they do decide to deal a new card D to Marvin, and card C is kept on the deck. Now, Marvin has some knowledge that no other player has: that the card C is in the deck (Semi-open card problem), along with the cards that Mallory had (Drop-out-collusion problem).
If each card is dealt separately (and not in groups), then the Semi-open card problem can be limited to a single card. The drop-out-collusion problem cannot be theoretically solved without violating the property R5. A k-out-of-n threshold scheme can be used to exclude a quitting players cards from the deck, but then k players would be able to collude to see any player cards during the game. In most card games players are given more than one card, so the drop-out-collusion problem is much worse than the semi-open card problem, and the the later can be left unsolved.
The semi-open card problem is similar to the Two-General Problem, which is an example that shows that common knowledge cannot be gained in a distributed system with unreliable messages. Nevertheless, there is a solution to the semi-open card problem which involves a third party with minimum trust: each player, except Marvin, sends his key share to the TTP, and only after (n-1) key shares have been received, the TTP forwards the key shares to Marvin. The TTF can gain no information from the key shares, and the semi-open problem only arises if the TTP is colluding with Marvin.

Some authors have considered the possibility of a new player entering an on-going game. We state that, though it may be possible, it is insecure, because there is no way that a new player can be sure the dealt cards were really random (the deck is not marked). For these reasons we will not consider late-enter protocols.

We'll discuss some consequences of the requirement R7 (absence of trusted third party or TTP-free). When MP protocols are implemented and used on real networks, by real people, protocols become both part of and build on other protocols. Then there are always other third parties (TPs) which are required to take part, either on-line or off-line. For example, every remote accessible non-broadcast protocol needs a network connection provider (like ISPs) to forward and transmit messages. Every game that players interact with new opponents requires the services of a Certification Authority to validate identities. Every game that has timeouts requires an external time-stamping service (to assure accurate times on messages). Every game whose outcome has some monetary significance must account for a logging service (to keep a log of all the game actions to allow ruling on a dispute), auditing entities (to rule on a dispute), on-line banks (to hold the bets and pay the pot), payment gateways (to transfer money from the player to the on-line bank), and so on. Any of these third parties can eventually act maliciously and make a player loose a game, or, in the context of a voting protocol, impede an individual from voting. As examples, an ISP can simply close a players connection, and a time-stamping service can provide forged or delayed timestamps for a player and mislead the others to act as if that player had timed-out and so refuse to accept his actions. Although the fact that third parties powers can be sometimes lowered by more clever protocols or technical solutions, some cannot, due to theoretical limitations and the possibility of collusion between trusted parties, as we'll describe in the following sections. One may ask what is the benefit of TTP-free protocols if trusted parties are required anyway in other protocol layers. Since in TTP-Free MP protocols no party knows other private cards, a malicious TP has a lot less information to decide when is should be wise to attack a player. He can't attack a player when he is about to win because he's having a good hand. Still, a TP in collusion with a player can attack a surely winning player when his partner is about to loose because he has engaged in a high stake game having very bad cards.

Mental poker protocols do not generally provide authenticity and privacy for the messages exchanged. The protocols must be run over a secure (authenticated) communication channel. Also the communication channel must be reliable.

1.2.2. Extended Requirements

Although a protocol which satisfies all the previously described requirements is theoretically correct, most of the previously known protocols cannot be used in real-time with current personal computer technology due to poor protocol performance. And because the MP concept is about secrecy, we can't expect people to hire CPU cycles from a remote facility to do the job, at least with current development on cryptographically secure remote computing (fully homomorphic encryption). Then all computations must be done locally.

R10. Real-world comparable performance: The protocol must allow real-time play, satisfying users expectation. This includes bandwidth usage, memory usage and CPU usage.

When we compare the performance of different MP protocols, an important characteristic is how are the CPU intensive calculations and data transfers distributed over the protocol lifespan. Most users may not tolerate high delays during the interactive part of the game, but would not bother to wait during shuffles or after showdowns. Other users, planning to play multiple games against the same players, would be glad if the CPU-intensive calculations of future games could be pre-processed in background during current play. A good protocol should allow both fast interactive play and shuffle pre-computation. Delayed verifications can help, but bring other new issues, such as the danger of suicide cheating, as we'll see in requirement R13.

To compare equally-capable protocols we'll need to rule out those which only allow two players to play. Multi-player MP protocols are far more complex than two-player protocols and must be carefully designed to prevent coalitions. Also for some games that allow card transfers, the number of players can be higher than the number of available cards.

R11. Variable number of players: The protocol must admit an unlimited number of players.

Finally we'll introduce some new properties which are required to allow some other card games (rather than only Poker) to be played.

R12. Card transfer actions: Return card to deck, Deck reshuffle, Face-down discarding, Private card transfer between players (with the other players consent)

Some of the protocols offer partial support for these operations. In some cases the protocol authors don't account for the procedures to do so, but the protocol itself can be easily extended. In other cases, some actions are impossible due to shortcomings in the protocols design. For example, [Cr86] does not allow a player to return a card to the deck or the deck to be re-shuffled.

R13. Protection against suicide cheaters: Cheating, even if discovered, cannot be used as a medium to reach other goals.

Suicide cheaters are malicious players willing to cheat even if they are sooner or later detected, in order to gain some information about other players way of playing (strategy) or generate a change in other players mood.

Possible suicide actions are:

a) Change you own cards or other players cards when dealing.
b) See other players cards when they should be private.

In some games, strategic information is of extreme importance. A protocol is fully protected against suicide cheating if it allows the players to detect cheating and the cheating parties without revealing private keys, private cards, nor any strategic information, including strategic information regarding how a player would react to an event that would not occur without cheating. This implies that cheating must be detected immediately after the moment cheating occurs. If suicide cheating protection is not required, verification operations can be delayed to gain performance and responsiveness.

R14. The deck can have indistinguishable duplicated cards

Some unusual applications and games require more than one copy of a card. For those applications decks must support indistinguishable copies of the same card value, without revealing that such a duplicate exists in the deck. There are Secure Multi-party Computation protocols for computing arbitrary boolean functions [dBo90, NR98] with a deck of cards that requires indistinguishable card duplicates. The AND protocol in [St01] requires them. Also some games like Pinoche or Bezique require duplicate cards. MPF support having duplicate cards as long as the cards are created at the beginning of the game (see special protocol), but duplicate cards cannot be transferred between players without leaving trace of which clone is being passed. This does not restricts the use of MPF for the AND protocol, but it does for some card games. Also, MPF does not support the creation of duplicated cards on-the-fly without leaving use traces. Protocol [BS03] fully complies with property R14.

It should be noted that in some games players are required to perform a cyclic shuffle of the deck (also known as “cutting” or “splitting” the deck). Although in practice most games require that players “cut” the deck, this action is only used as a means of avoiding the cheating of the dealer. In practice, properties R1, R2 and R3 assure that no cheating can occur during dealing. Nevertheless the AND protocol requires precisely cyclic shuffles. Cyclic shuffles can always be obtained by the procedure described in [BS03] using normal shuffles and a cut-and-choose protocol, although a faster protocol could be desirable.

1.2.3. Unreal Requirements

Some suggested requirements can be called “unreal”, because cannot be obtained with a real deck of cards (at least not without a trusted third party) but could be useful for some unusual applications.

R15. Players can jointly create duplicated cards on-the-fly, even if face down.

The AND protocol requires the duplication of face down cards when evaluating arbitrary boolean circuits, although it's shown that the duplication can also be obtained by using cyclic shuffles.

R16. Players can test if a card in the deck belongs to a certain set.

In some games is necessary that players follow suit, even if the card is not immediately revealed. Then property R16 can be useful to check this before the game continues and avoid suicide cheating. Another simpler property is to allow players to prove that an owned card belongs to certain set. We can do it using the basic properties. If a player wants to prove that a card x in his hand (the set X) also belongs to the set Y, he shuffle-masks the set X and then opens the card that is the result of the masking of x.

R17. Players can test if two cards belongs to the same set from a partition of the deck.

If the deck is partitioned in sets, then this property allows to verify that two cards belong to the same set, without knowing which is it. There are situations when we want to follow a set of rules over card relations. All players should be able to verify the rules are being correctly applied but only the player making the relation knows which rule in the set is being applied. This is very useful in e-cash protocols where, for example one rule may state a one hundred dollars digital bill can only be converted into another one with the same value, and another rule may state the same for ten dollars bills. In this case R17 allows to destroy trace information but preserve monetary value.

R18. Players can jointly test if a card is higher than some other card.

This property solves the Two-Millionaire problem, and is more difficult that just checking if a card value is higher than a certain value, which can be solved by choosing the set of higher card values and using the property R16.

R19. Players can jointly increment/decrement the value of a card.

We haven't found a situation this property is required, but still sounds nice to have. A game that makes use of this properties is yet to be invented.

1.3. Measuring Performance

A key property used for protocol comparison [CR05] is the computational and communications requirements for a given configuration of players, cards and security threshold. No other resource, such as computer memory, seems to be relevant to performance in MP protocols.

1.4. Delayed Verifications

Because most the most CPU-demanding operation is the verification of the shuffle [CR05], we can achieve real-word comparable performance by almost any MP protocol, including [KKO97], [CDRB03] or [BS05] by delaying the cheating detection sub-protocols until the end of the game, once a week, or any other time in the future. We can also spread some computations and communications over the idle cycles of the CPU. But In low-trust/high-stakes scenarios, immediate cheating detection is required. The two main reasons are:

a. Protect the players against suicide cheating.
b. Avoid blocking the money in the pot.

Money won must be blocked from further transactions until the verifications are over. If not, then money obtained by cheating in games that were not yet verified could be bet in following games, creating chains of games that would be required to be rolled-back when a cheater is detected, which would become a legal nightmare.

1.5. Non-interactive vs. Interactive Proofs

Each player must verify the correctness of other players private operations. In a game with n players, a player proof that an operation is correct generally requires (n-1) interactive proofs, one for each remaining player. Non-interactive proofs are generally longer, but the same proof can be reused and sent to each of the remaining players. As the number of players increase, non-interactive proofs become less expensive than interactive proofs both in CPU and bandwidth usage. The turning point depends on the protocol and security threshold, but generally it's around n=20 for most common card games.

1.6. External on-line Auditing

An external on-line auditing party is generally required when not only the result but all the events in the game itself are important. We'll show one example of such requirement. Suppose there is a tournament and n players playing will be playing a series of games together. Suppose that 2 points are given to the winner of each game, 1 to each player involved in a draw, and 0 points to a looser. Suppose also that 5 extra points are given to a winner of a game that has a poker of aces. Then a group of players may collude and gain some advantage over the remaining honest players by choosing fake cards for each game, letting each player win a game with poker of aces and providing fraudulent proofs to an off-line auditing authority. Even without faking cards, players can just exchange private key information on an external channel, and decide which cards will be dealt to force a specific player get a poker of aces. To avoid this situations, and auditor party may need to take part in the protocol as another card shuffler and take part in verification protocols as another verifier.

1.7. Hash Chains

A **cryptographic hash function** is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the (**cryptographic**) **hash value**, such that an accidental or intentional change to the data will change the hash value. The data to be encoded is often called the "message", and the hash value is sometimes called the **message digest** or simply **digest** [B96]**.**

A **Hash Chain** is a sequence of messages where each message includes a message digest of the previous message on the chain.

If a broadcast medium is used for communication Messages protocol messages can be linked in a hash chain, where each message includes the digest of the previous message sent. In this way, the last messages serves as an authentication code for the whole protocol transcript. An example of a hash chain is the Distributed Notarization Chain (DNC) [CR03].

If peer-to-peer connections are allowed, then a **Hash DAG** (Direct Acyclic Graph) can be used. In a hash DAG, each message sent by a player may reference more than one previous message. Each message carries the hash digests of all previous messages received or sent by that player that are terminal nodes of the DAG that is constructed by the message reference relation, as known by the sender.

1.8. Theoretical Security in Mental Poker Protocols

SRA proved that for a shuffle to be correct and fair, you cannot get perfect security. The best one can have is computational security. MPF attempts to provide a coherent and user selectable level of computational security.

1.9. Kinds of Proofs

Informally, a Zero-knowledge Proof (ZNP) is a protocol that allows a party (the prover) to convince another party (the verifier) that he can perform some secret actions or that he has some secret information, without giving any clue of what these actions are or this information is. A **Perfect Zero-knowledge Proof** (**PZNP**) is a protocol in which the verifier cannot theoretically recover any information regarding the secret from the protocol transcript. A **Computational Zero-knowledge Proof** (**CZNP**) is a protocol in which the verifier cannot computationally recover any information regarding the secret from the protocol transcript. A **Zero-knowledge Argument** (**ZNA**) is a protocol where the prover can only cheat the verifier by performing an intractable computation. A **Computational Zero-knowledge Argument (CZNA)** is a protocol that is secure only if both parties are computationally bounded. Although perfect zero knowledge proofs provide perfect secrecy, there is no point that the verification protocols provide grater security than other operations in the protocol, like encryptions. An attacker generally tries to crack the weakest link of the chain. If the other operations, like encryptions, are secure under computational assertions, then the verifications can also be computational zero-knowledge arguments, without decreasing the overall security. Because perfect ZNP protocols are generally expensive in communication and computation requirements, MPF offers, in addition to perfect zero-knowledge protocols, computational zero-knowledge arguments that rely on the same security assumptions as the ciphers used along the protocol. In most games parties take symmetric roles (both proving and verifying operations), so there is no point in having a verification protocol that can withstand a computationally unbounded prover and a computationally bounded adversary or vice-versa. As stated before, MPF relies only on computational security.

Verification protocols in [KKO97], [BS03] and [CDR03] rely on cut-and-choose perfect-zero-knowledge protocols, which are slow because of the need of iterated rounds to achieve a certain threshold of certainty. To gain performance MPF also provides fixed round proof protocols. PHMP uses an iterated protocol, but with with fewer rounds. For example, to verify a shuffle of a complete deck of poker cards, the classical interactive cut-and-choose protocol requires 20 iterations to achieve 2^{-20} cheating probability, while the proposed PHMP verification protocol requires only 4 iterations.

2. MPF base concepts

MPF is a framework which defines abstract classes, operations and protocols that allow generation of new MP protocols satisfying some or all the required properties. Like SRA, MPF is defined for an ideal Commutative Group Cipher (CGC). A CGC is a commutative cipher which also provides group operation on keys. Examples of commutative ciphers that are not CGC can be found in [BM07].

Formally, a cipher F is a tuple (E,D) where
E is a function KxM -> C, noted $E_k(m)$ and
D is a function KxC -> M, noted $D_k(c)$
Where:
K is the Key space
M is the plaintext space
C is the ciphertext space

And for any $m \in M$, $D_k(E_k(m)) = m$.

To be a CGC, a cipher F must have these other properties:

1. C=M
2. There exists an operator "*" for which G=< K, * > is a commutative group.
3. For all $x \in \mathrm{M}$, and any $k \in K$, $E_k(E_q(x)) = E_{(k*q)}(x)$
4. For all $x \in \mathrm{M}$, and any $k \in K$, $D_k(x) = E_j(x)$ for $j=k^{-1}$
5. For all $x \in \mathrm{M}$, $E_1(x) = x$
6. For each valid pair of keys $k,r \in K$, obtaining k from (k*r) must be computationally infeasible or theoretically impossible.

A a **ciphertext-only attack (COA)** or **known ciphertext attack** is an attack model for cryptanalysis where the attacker is assumed to have access only to a set of ciphertexts. The attack is completely successful if the corresponding plaintexts can be deduced, or even better, the key. The ability to obtain any information at all about the underlying plaintext is still considered a success. A **known-plaintext attack (KPA)** is an attack model for cryptanalysis where the attacker has samples of both the plaintext and its encrypted version (ciphertext), or the attacker has a set of ciphertexts, has knowledge of the plaintext statistical distribution and can guess with high probability plaintext/ciphertext pairs. A **chosen-plaintext attack (CPA)** is an attack model for cryptanalysis which presumes that the attacker has the capability to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts. The goal of the attack is to gain some further information which reduces the security of the encryption scheme or, in the worst case, to reveal the secret key [B96] .

MPF can be proven secure in the random oracle model (ROM) relying on these external security assumptions on the CGC:

1. The underlying CGC is secure against ciphertext-only attack (COA) where the ability to obtain any information about the underlying plaintext is considered a successful attack.
2. The underlying CGC is secure against known plaintext attacks (KPA).
3. The underlying CGC is secure against chosen plaintext attacks (CPA) (see note)
4. The underlying CGC is non-malleable, for any suitable definition a malleability that excludes malleability imposed by commutativity.
5. The CGC is deterministic [K97].
6. The CGC provides historical security [W06] if card transfers between players are required and there is a need to hide the card transfer history.

Standard formal security definitions, such as semantic security or non-malleability do not fully capture adversarial abilities when using a commutative cryptosystem. Definitions such as cascadable semantic security [W06] cannot be applied, since we rely on a symmetric deterministic cipher. The concept of historical security can be easily adapted, and we'll require so if card transfers are required.

Assumption 3 is only required for the VSM-L-OL base protocol (see section "2.1. MPF base protocols") or if the verification protocols will be executed in parallel. It is required in VSM-L-OL because VSM-L-OL has a round without verification. It is required when doing parallel verifications because the output from a player calculation could be sent as input to another player before the verification protocol finished, which opens a window of time for a CPA attack.

Assumption 4 can be removed if the verification protocols are modified to withstand malleability. We'll show that this is possible imposing only a slight penalty in performance.

To proof the security of MPF, two additional assumptions are required (COUC and CUI). Nevertheless, these are not external assumptions. The protocols in MPF guarantee these properties hold on start and keep holding as postconditions of the protocols.

Computational Uniqueness of Open Cards (CUOC)

Given a set of cards X, a **conversion key** is a key that can encrypt a card in the set X into a different card in the set X. A set of cards X is **computationally unique** (**CU**) if no proper subset of the players can compute a conversion key. Finding such a conversion key should be as hard as breaking the CGC. The deck, before being shuffled, consist of open cards. The **CUOC property** states that the initial deck is CU. This property is less stringent that the one stated in [WW09], where the requirement is that the card values are indistinguishable from independent uniform random variable, as a means to avoid conversion keys.

Computational Uniqueness Invariant (CUI)

MPF sub-protocols processes lists of cards (inputs) and generates new lists of cards (outputs). There are two kinds of protocols: action protocols and verification protocols. Action protocols can be verified and unverified. Verification protocols verify action protocols. Every action protocol has the precondition that each input list of cards is CU. Every verified action protocol guarantees that the output list of cards remain CU. Every action protocol satisfies this **Computational Uniqueness Invariant (CUI).** The CUI can be achieved if all players can track the source and destination of each card value that undergoes encryption. And this is done by the verification protocols executed as sub-protocols.

Because each card list used as input of a protocol is a subset of a list that was the output of another protocol (with the exception of the first protocol which creates the deck), a verified action protocol that receives an input set of cards that has been proven to be CU will produce a new set of cards that is proven CU. The CU condition spreads from the CUOC up to every card list used through verified action protocols. Verification protocols may require some input card lists to be CU and can guarantee that another input list is CU, expanding the number of CU card lists. Verification protocols are the bridge that provides action protocols CU guarantees for their outputs. MPF allows the parallelization of sub-protocols. In a situation where a card list whose verification for CU is still pending on an unfinished protocol, and the card list is supplied as input as another protocol, then the failure of the former implies nullity of the later.

One way to assure the CUOC is by choosing open cards values at random, however the card values chosen must be valid plaintexts for the CGC. Computing a conversion key would be as hard as a KPA on the CGC. We give two computational protocols for creating open card values that assure CUOC.

We must note that the CUOC requires that the number of open cards to be orders of magnitude smaller than the key space, to keep the probability of finding a conversion key low.

Types of Protocols

This thesis defines and uses different kinds of protocols, which are detailed below, and summarized as follows:

MPF framework: MPF is a framework which defines four base protocols. Each base protocol is an abstract Mental Poker Protocol, and each represents a different way to accomplish a similar objective, with slight security and performance variations. Note that without specifying the CGC, an MPF base protocol is not by itself a complete Mental Poker protocol: the implementor must still choose a CGC from the ones provided or choose a new CGC and replacements of the verification protocols if the CGC is malleable.

Base protocols: As stated, each base protocol, plus a CGC, is a complete Mental Poker protocol.

Card protocols: Card protocols operate on cards. For example dealing, opening, showing or transferring a card.

Verification protocols: MPF require players to privately operate on cards, and these operations need to be publicly verified to avoid cheating. Verification protocols provide a guarantee that private operations are indeed correct.

PHMP: PHMP is a concrete and fully specified Mental Poker protocol for the Pohlig-Hellman cipher.

ECMP: ECMP is a concrete and fully specified Mental Poker protocol for an elliptic curve cipher.

2.1. MPF base protocols

MPF offers four base protocols: VSM-L-OL, VSM-VL, VSM-VPUM and VSM-VL-VUM. The names refer to the different rounds that each base protocol applies to cards. An explanation of the meaning of each letter in the protocol name is provided in Table 1.

Letters	**Meaning**
V	Verified
L	Locking round
O	Open
SM	Shuffle-Masking round
P	Partial
UM	Unmasking round

Table 1: Meaning of Letters in Protocol Names

Each one of them implements the standard properties described and some of them the extended requirements.

2.2. MPF instantiation

To create a specific MP protocol in MPF, the following steps are carried out:

1. Choose an appropriate CGC
2. Choose one of the MPF base protocol
3. Choose the verification protocols from the ones offered by MPF (see section "2.3. Verification Protocols")
4. If desired, create ad-hoc verification protocols targeted to the specific CGC.

2.3. Verification Protocols

MPF defines seven kinds of verification protocols:

Locking Verification Protocol (LVP)

LVP is a protocol that allows a party (the prover) to prove to other parties (the verifiers) that a list of ciphertexts is the encryption of a lists of plaintexts (without any permutation), without revealing the keys. Each plaintext can be encrypted with a different key.

Shuffle-Masking Verification Protocol (SMVP)

SMVP is a protocol that allows a party (the prover) to prove to other parties (the verifiers) that a list of ciphertexts is the result of the encryption of a list of possibly permuted plaintexts with a single key, without revealing the key nor the permutation function. The protocol can also generate a *representative*. A representative is a pair plaintext/ciphertext which is produced and broadcast at a certain point in a protocol and is referenced afterwards to verify a similar operation or an inverse operation.

Undo Verification Protocol (UVP)

UVP is a protocol that allows a party (the prover) to prove to other parties (the verifiers) that a list of plaintexts is the result of the decryption of a corresponding list of ciphertexts with a single key, without revealing the key. Also the protocol can prove that the key is equal to a previously used key or a product of previously used keys. To reference a previously used key the protocol requires a representative previously generated. To execute a UVP for product keys, multiple representatives (one for each key) must be transformed into a new single product representative using the sub-protocol Build-Representative. The full UVP protocol is not required in MPF. Nevertheless representatives are required to enable some advanced protocols such as Return-Cards-To-Deck or Card-Key-Verification (which is used to protect the protocol against suicide cheaters).

Re-Locking Verification Protocol (RLVP)

RLVP is a locking verification protocol of a single encryption with a single representative. This verification protocol is used in the card protocol Build-Representative to achieve protection against suicide cheaters for the base protocols for VSM-VL and VSM-L-OL. The base protocol VSM-VPUM does not require this verification protocol.

Re-Shuffle-Masking Verification Protocol (RSMVP)

RSMVP is a protocol that allows a party (the prover) to prove to other parties (the verifiers) that a list of ciphertexts is the result of the encryption of a list of possibly permuted plaintexts with a single key, without revealing the key nor the permutation function. The key used must be equal to a masking key used before, referenced using a representative. This is actually a UVP with a representative, but using an encryption function operation instead of a decryption operation. It is only used in the card protocol Verified-ShuffleRemasking-Round, which is a sub-protocol of Return-Cards-To-Deck.

Shuffle-Locking Verification Protocol (SLVP)

SLVP is a protocol that allows a party (the prover) to prove to other parties (the verifiers) that a list of ciphertexts is the result of the encryption of a list of possibly permuted plaintexts, without revealing the keys (each encryption can be done using a different key). Shuffle-Locking verification protocols are used by players to shuffle their own hand cards to protect strategic information, such as the transfer history of the cards, and avoid card tracking (card protocol Shuffle-Hand(2)). Also this protocol is used in the card protocol Show-Cards(2), which is an alternate protocol for showing cards.

Unmasking Verification Protocol (UMVP)

UMVP is an UVP which verifies encryptions with a single key, and no permutation, using only one representative (generally taken from a single shuffle-masking round). This verification protocol is used in the card protocol Verified-Unmasking-Round which is a sub-protocol of Multiple-Cards-Deal, used to deal cards for the base protocol VSM-VPUM only. Table 2 illustrates certain differences between verification protocols.

Encryption type	**Permuted**	**Not Permuted**
Same key	SMVP [can generate a representative] RSMVP [requires a representative]	UVP UMVP [requires a representative] RLVP [requires a representative] (*)
Different Keys	SLVP	LVP [generates many representatives] (**)

Table 2: Differences between verification Protocols

(*) Because RLVP verifies a single encryption, the distinction between Permuted/Not permuted does not apply.
(**) Each LVP plaintext/ciphertext pair is itself a representative

The foregoing protocols share certain features; accordingly, MPF implements these protocols as calls to a more general protocol, the Unified Verification Protocol or UniVP. Nevertheless, the implementor can replace an specific verification protocol with an alternate version, based upon performance or security considerations. For example, PHMP perform certain replacements.

2.4. Unified Verification protocol (UniVP)

The Unified Verification Protocol (UniVP) is the core sub-protocol for all verification protocols. Following are the signature definitions, and in Table 3 description of UniVP protocol arguments. See section 3.7 ("Card Protocols") and Table 4, for a detailed description of the notation used for protocol arguments.

```
Signature: UniVP (
        private in L :Key-List,
        private in T :Permutation,
        public  in X :Card-List,
        public  in Y :Card-List,
        public  in p :Player,
        public  in Permuted :Boolean,
        public  in SameKey  :Boolean,
        public  in RX :Card-List ,
        public  in RY : Card-List)
```

L	L is a list of keys. Each key has been used to encrypt the element in the list X with the same index. If L has only one element, then the same keys is used for all elements in X.
T	The permutation applied to X before encryption into Y.
X	X is the list of plaintexts. If Permuted = true, the X must be CU.
Y	Y is the list of possibly permuted ciphertexts
p	p is the player who proves to operation is correct
Permuted	if true, allows the cards on the set Y to be permuted, and verifies the correctness of the permutation.
SameKey	If true, then the verifier must prove all encryptions are done using the same key.
RX	RX is a list of plaintexts used as input or output representatives If RX is empty, then no representatives are being used. If (SameKey=false) then RX must be empty.
RY	The ciphertexts corresponding to each element in RX.

Table 3: UniVP Arguments

Note that RX is the list of required (input) representatives and also the list of produced (output) representatives, because no protocol can have both input and output representatives.

All the verification protocols can be constructed as macro calls to the UniVP:

Locking Verification Protocol (LVP)

The protocol signature is :

LVP(Key-List L, Card-List X, Card-List Y, Player p)

Definition:

LVP(L,X,Y,p) = UniVP(L , Identity, X,Y,p,false,false, [ ], [ ])

Shuffle-Masking Verification Protocol (SMVP)

The protocol signature is :

SMVP (Key m, Permutation T, Card-List X, Card-List Y, Player p, Card-List RX, Card-List RY)

Definition:

SMVP (m,T, X,Y,p,RX,RY) = UniVP([m], T, X,Y,p, true, true, RX, RY)

Unmasking Verification Protocol (UMVP)

The protocol signature is :

UMVP (Key m, Card-List X, Card-List Y, Player p, Card rx, Card ry)

Definition:

UMVP(m,X,Y,p,RX,RY) = UniVP([m], Identity, X,Y,p,false, true, [ry],[rx])

Re-shuffle-masking Verification Protocol (RSMVP)

The protocol signature is :

RSMVP (Key m, Permutation T, Card-List X, Card-List Y, Player p, Card rx, Card ry)

Definition:

RSMVP(m,T, X,Y,p,rx, ry) = UniVP([m], T, X,Y,p, true, true,[rx],[ry])

Re-Locking Verification Protocol (RLVP)

The protocol signature is :

RLVP(Key l, Card x, Card y, Player p, Card rx, Card ry)

Definition:

RLVP(l,x,y,p,rx,ry) = UniVP([l] , Identity, [x],[y],p,false,false, [rx], [ry])

Shuffle-Locking Verification Protocol (SLVP)

The protocol signature is :

SLVP (Key-List L, Permutation T, Card-List X, Card-List Y, Player p)

Definition:

SLVP(L,T, X,Y,p) = UniVP(M, T, X,Y,p, true,false, [ ],[ ])

LPV, UMVP and SLVP preserve the CU condition of the cards lists. If X is CU then a successful execution of the protocol guarantees that Y is CU, but does not impose that X must be CU as a precondition. SMVP requires X to be CU as a precondition, and a successful execution of the protocol guarantees that Y is CU.

MPF defines three UniVPs:

- An interactive cut-and-choose UniVP that provides a Perfect-Zero-Knowledge Proof (I-UniVP).
- A non-interactive UniVP that provides a computational zero knowledge argument (NI-UniVP). This protocol is obtained from I-UniVP, applying Fiat-Shamir transformation [FLS90].
- A fast interactive UniVP, which provides a computational-zero-knowledge argument (FI-UniVP).

The implementor is free to choose the UniVP that suits his needs.

2.5. Ad-hoc Verification protocols and malleability

An encryption algorithm has the property of **malleability** (or it is **malleable**) if it is possible for an adversary to transform a ciphertext into another ciphertext which decrypts to a related plaintext. That is, given an encryption of a plaintext *m*, it is possible to generate another ciphertext which decrypts to *f*(*m*), for a known function *f*, without necessarily knowing or learning *m*. An **Homomorphic cipher** is a malleable cipher that translates an operation on the plaintext and to another (possibly different) operation on the ciphertext. Let E be the homomorphic cipher encryption function, then given two plaintexts x_1,x_2, E(x_1) * E(x_2) = E($x_1 \circ x_2$), for two (possible different) operators * and °.

A user selected CGC for MPF may be homomorphic or malleable in some other way. These are undesired properties, and reduce the security of MPF. Standard MPF (without replacements) is only secure under a non-malleable CGC. Nevertheless all protocols with the exception of some verification protocols withstand malleability present in the CGC. Verifications protocols such as the ones derived from I-UniVP and NI-UniVP withstand an homomorphic CGC. They also withstand dishonest verifiers. We´ll show a variation of FI-UniVP that is immune to homomorphic properties of the CGC. For other kinds of malleability, the security may have to be re-proven. If this proof fails, malleability in the CGC may render the UniVP completely insecure or insecure under a CPA. MPF may still be used if the provided UniVPs are replaced with ad-hoc protocols, specially targeted for the specified CGC to withstand malleability attacks. Also, there are other reasons to choose specific verification protocols, such as to increase the performance or to rely on other widely analyzed protocols, whose security has been pre-established.

2.6. MPF compared to SRA

MPF includes at least four MP protocols, depending on the base protocol. Four variants include VSM-L-OL, VSM-VL, VSM-VPUM and VSM-VL-VUM. Each one of them represents a different balance of performance and security. We believe VSM-VL-VUM and VSM-VL are the most secure, although we were unable to break any of the protocols. The core of MPF can be viewed as a generalization and optimization of the repaired SRA protocol.
The are at least three differences between MPF and SRA protocol:

1. In MPF each player encrypts each card with a different key, where SRA protocol uses the same key for all cards
2. In MPF, each codified card is guaranteed to be computationally unique, where the SRA protocol does not pose any restriction in the codification of the cards, nor a padding scheme.
3. MPF cannot suffer from information leakage problems (e.g. quadratic residuosity) of the card codifications because it poses restrictions on the quality of the CGC by definition.

2.7. MPF compared to Barnett-Smart

The main differences between Barnett-Smart protocol [BS03] and MPF are:

1. MPF uses a deterministic cipher. On the contrary, Barnett-Smart uses a probabilistic cipher (either ElGamal or Paillier's system).
2. Possibly because of 1, Barnett-Smart does not have a Abrupt Drop-out recovery protocol.
3. Barnett-Smart uses an iterated cut-and-choose protocol to verify encryptions. MPF uses a fixed round protocol. PHMP uses a variable-round protocol, but requires less iterations.

2.8. Keys Lifetime

All encryption keys are chosen for each game, and are disposed afterwards. This ensures that the information leakage due to the computational nature of MPF security is kept to a minimum. Other cipher internal parameters (such as a common modulus) can be either fixed for long periods or generated for each game, depending on the computational cost of creating a new set of parameters. If some parameters are fixed, then they must be standardized and publicly scrutinized for trap-doors.

2.9. Open Cards Lifetime

Before shuffling, real cards values are specially encoded in bit-strings known as open cards or O-Cards. Open cards are chosen such that no O-Card is an encryption of another O-Card with a known key (CUOC condition). Also, all card values used during a game are *rooted* in one of the open cards, meaning that each card value can be traced back to an open card with a series of encryptions with player secret keys. (although this doesn't mean that this trace is publicly available)
No player can bring his own deck of O-cards to play, because there is no easy way he can prove he doesn't know the conversion keys. Nevertheless a player could bring a deck created by a publicly verifiable method (such as taking digits of π as card values). Also, he could forward a deck created by a trusted authority (a TTP). This last scheme allows a marked deck to be used so a central authority (such as a game operator)

can track collusion of players by looking at the game transcript, but without interfering with the protocol on-line.

For maximum protection, and to protect the players from information leakage, we recommend that the deck is created by the players for each new game. If this represents a performance problem, the deck could be reused in multiple games, as long as the players are the same. Also, if a new player joins a game, the previous deck can be re-randomized (along with a proof of correctness) by that new player only.

2.10. Encoding of Open-Cards

A **pseudo-random number generator** (**PRNG**), is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. A **cryptographically secure pseudo-random number generator** (**CSPRNG**) is a pseudo-random number generator (PRNG) with properties that make it suitable for use in cryptography. One of the properties is that with the only knowledge of the numbers already generated, the numbers to be generated next should still be computationally indistinguishable from random.

To obtain a deck with CUOC property, open card values will be obtained by a protocol P that satisfy these requirements:

- The output of P is a list of open card values.
- Each party must take part in P (either by doing some computations or by providing a seed-share to some function)
- No proper subset of the parties should be able to repeatedly and privately evaluate the output of P in any intermediate stage of the protocol and use that information to modify their own computations or their seed-shares that contribute to the output of P.
- The output of P should be pseudo-random as long as one of the parties is honest and chooses random values when required.

We'll call such protocol a Collaborative Pseudo-Random Number Generator Protocol (CO-PRNGP).
A CO-PRNGP be realized by using at least one of this three computational methods:

1. Verifiable pseudo-Random functions (VRFs), Distributed Pseudo-Random functions (DPRFs) or Distributed Verifiable Random Functions (DVRFs) [MRV99][Lys02].

2. A Hash-based approach: guarantees that no proper subset of the players can force the output values to be biased or chosen, under the random oracle model.

3. A locking round (see section “2.11. MPF Rounds”) which guarantees all open cards are rooted back to a single initial card but assures the conversion key is shared among the players in such a way that no proper subset of the players can recover the key.

2.10.1. VRF-based CO-PRNGP

To build a VRF based CO-PRNGP each player i creates a personal VRF f_i and publishes the public parameters. A public variable d is used to count the number of decks created. Each time a new deck is required, each player publishes $x_i=f_i(d)$ along with a proof that verifies the correctness of the computation. All outputs x_i are xor-ed to create a seed for a public CSPRNG, whose output is used to build the o-card values.

2.10.2. Hash-based CO-PRNGP

A **commitment scheme** allows one party to commit to a value while keeping it hidden, with the ability to reveal the committed value later. Commitments are used to bind a party to a value so that it cannot change it afterwards.

To build a hash-based CO-PRNGP, we split the protocol into two stages. In the first stage each party commits to an input, and then all inputs are revealed. In a second stage a special cryptographically secure pseudo-random number generator (CSPRNG) is used to compute pseudo-random sequence required. This generator takes a variable number of inputs as seeds. Each party should take care that the input seed provided is unpredictable by the other players. It's recommended to use true physical random bits. If unavailable, then a suitable replacement, such as a CSPRNG with a large pool should be used (as in Yarrow or Fortuna) to compute the seeds. Better PRNGs can be constructed from one-way functions [GGM86, HILL99] and by well studied number-theoretic assumption (such as the Blum Blum Shub PRNG). A popular such construction is due to Naor and Reingold [NR97] and is based upon the Decisional Diffie-Hellman (DDH) assumption. To approximate a CSPRNG with a variable number of seeds, we concatenate all the seeds in a fixed order to form a message and then hash the message, using a cryptographically secure hash function. The resulting hash is used as the CSPRNG seed.

With reference to Fig. 2, the hash-based method is diagrammed.

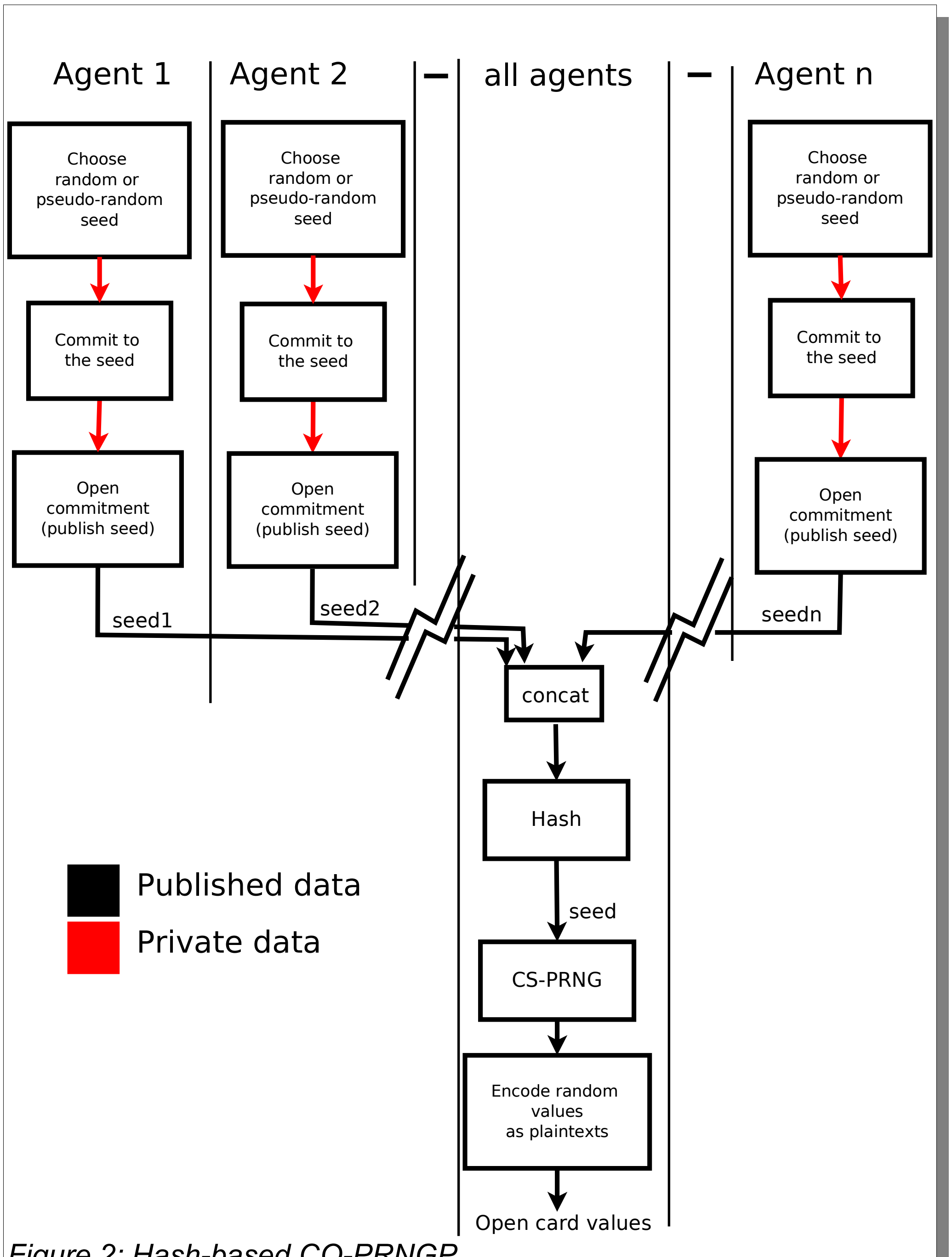

Figure 2: Hash-based CO-PRNGP

2.11. MPF Rounds

In MPF most of the steps are done in rounds. A round can be of at one of three kinds: Shuffle-Masking (SM), Locking (L), and Undo (U). Also, rounds can be verified (by MVP, LVP or UVP) or unverified. Shuffle-Masking rounds always precedes locking and undo rounds, with the exception of a locking round performed to create cards encodings. Masking rounds are always verified. Rounds can be complete (all players publish their output) or partial (the last player to compute does not reveal the result). Masking rounds are always complete rounds. For simplicity, MPF does not directly use the undo round but defines two other rounds: UnLocking (UL) and Unmasking (UM). Unlocking is an undo round of a single locking round. Unmasking is an undo round of a single shuffle-masking round.

Rounds Description

- Shuffle-Masking Round:
  In a Shuffle-Masking round (SM), shown in Fig. 3, each player encrypts (with a single key) and permutes all the input cards. The output of a masking round are a set of masked cards or M-Cards. Masking rounds are always verified.

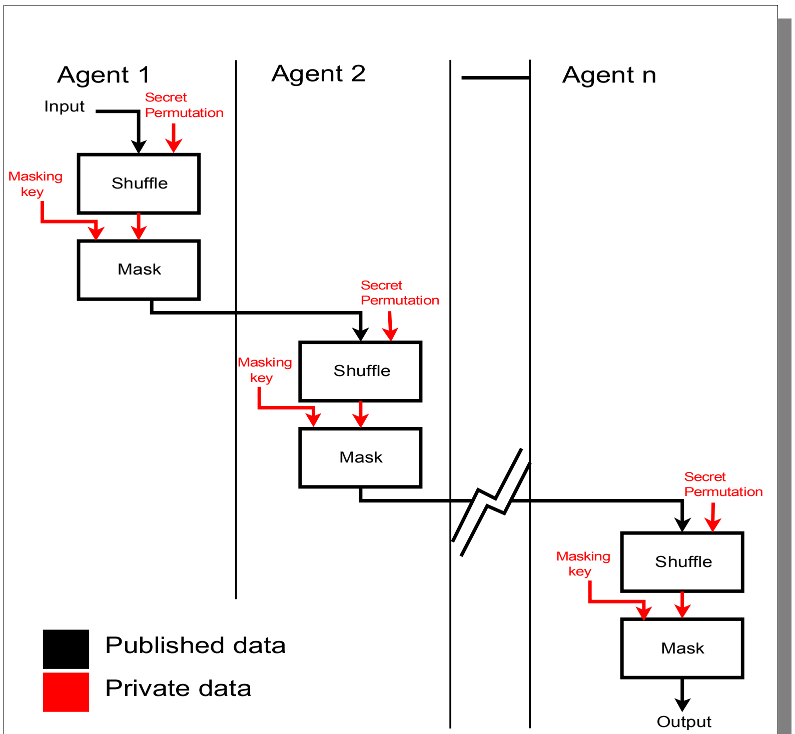

Figure 3: Shuffle-Masking round

- **Locking Round:**
  In a locking round (L), shown in fig. 4, each player encrypts each input M-card with a different private key. Masked and locked cards are called ML-Cards (no matter how many locking/rounds have undergo).

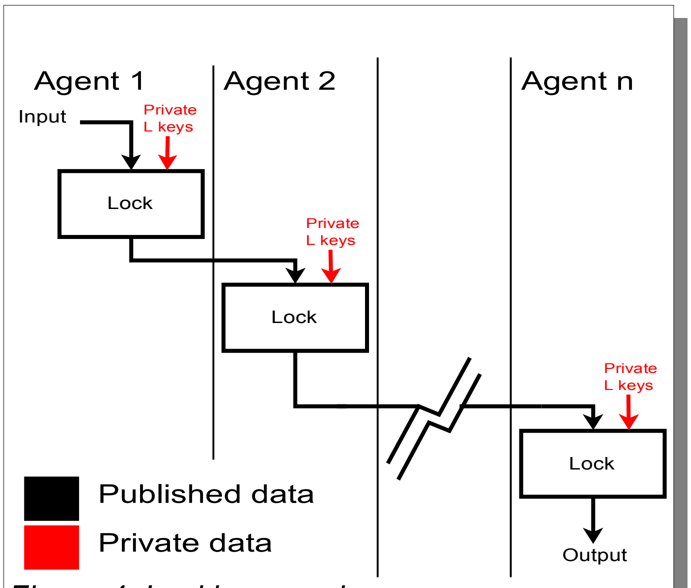

Figure 4: Locking round

- In a undo round (U), shown in fig. 5, each player undoes the encryptions previously done to each input card in a set of rounds S. The protocol specifies the set S. The implementation must track which keys apply to which cards, to undo locking rounds.

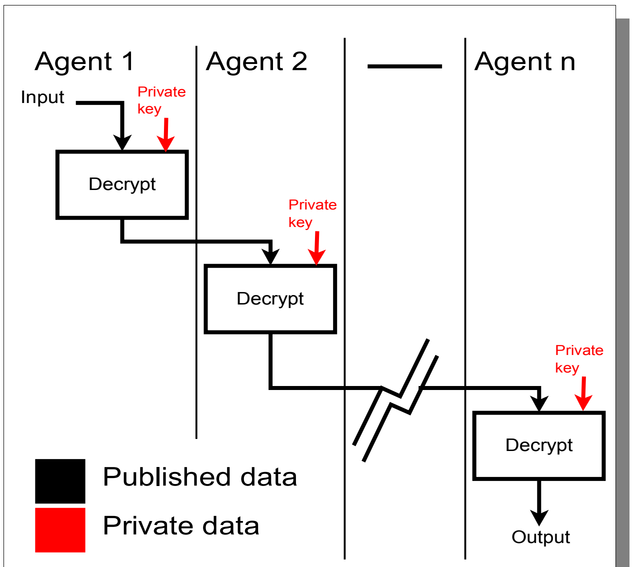

Figure 5: Undo round

Undo rounds can be complete (all players publish their output) or partial (the player receiving the card undo last, and the output of the last player computation is not revealed).

In VSM-L-OL, there are 3 rounds (VSM, L_1, OL_2) The last round is a “Open” round. An open round is a round which is verified, but only at the end of the game. In VSM-VL, there are only two rounds (VSM, VL) . In VSM-VPUM, 2 rounds (VSM, VPUM), where VPUM is a verified partial unmasking round. In VSM-VL-VUM, 3 rounds (VSM, VL, VUM).

2.12. Free Cards

When a card is dealt, an encrypted card is taken from the deck. This is called a a Free Card or F-Card. Each player takes note of who is the holder of each F-Card, and this information is kept in a table within each players computer memory (the Card-Holder table). To allow card exchanges, some times F-Card bit strings

are disposed and new ones generated. Therefore each player must keep a dynamic mapping for F-Card holders.
Any player can shuffle his own hand cards to erase any trace between the F-Cards that were given to him and a new set of F-Cards. During the shuffle, each other player disposes the old F-Cards in the Card-Holder table and adds the new F-Cards. There are two types of hand shuffles:

a. Single key shuffle (using a protocol similar to a single player shuffle-masking operation)
b. Multi-key shuffle (using a protocol similar to a single player locking operation).

A single key shuffle is verified by an SMVP. A multi key shuffle is verified by an SLVP.
The reason to have two different protocols is that an SLVP can be considerably faster than SMVP, depending on the number of players, the number of cards to shuffle and the UniVP selected.

2.13. Card Keys

The key that results from the product of a masking key and all locking keys applied to a single player to a certain card is called the Card Key. When a card is dealt, the card ends up encrypted with each players card key. Generally, the player receiving a card keeps his card key secret, while all the others must publish them. Multiple encryptions allow players to publish a card key without revealing the masking key. Because each card key can be decomposed as a key pairs or triplets in as many or more ways as keys in the key-space, the masking key remains secure even of the card key is published.

2.14. Card Dealing

There are two methods to deal a card to a player:

a) Key share disclosure method

The diagram of Fig. 6 illustrates how agents interact to deal a card to player *x* under the Key Share Disclosure method. To open an L-card/ML-Card, all the players except the one receiving the card publish the Card Key, which is is the product (in G) of all locking and masking keys which hasn't been undone. The player receiving the card computes a master key which is the product of the published card keys with his secret card key (calculated in the same way). The original L-card or ML-card becomes a new F-card for that player, and the card is taken out from the deck. The F-card is privately decrypted by the player to get an open card.

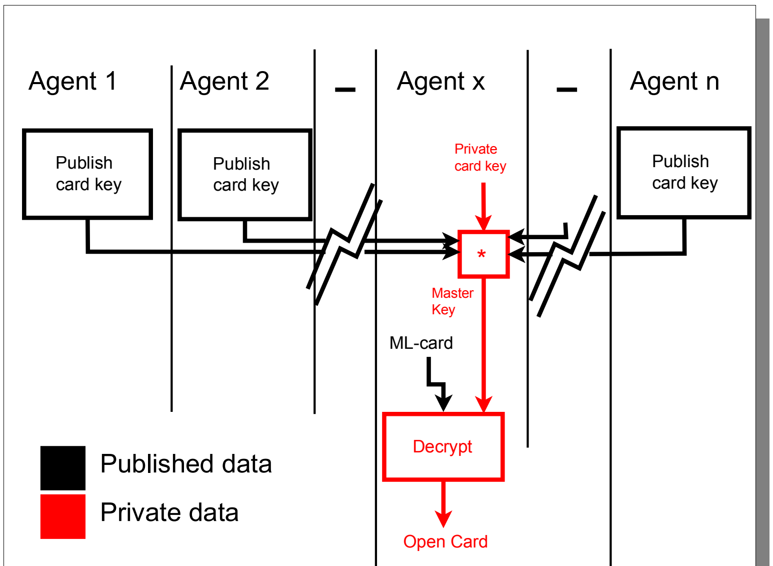

Figure 6: Key share disclosure method to deal a card

b) Partial undo method

The diagram of Fig. 7 illustrates how agents interact to deal a card to player *x* under the Partial Undo method. Players do a partial undo round for the set of rounds that remain to be undone for the cards to be dealt. The player receiving the card must be the last in the round. The input cards to the last player in the round are new hand cards for that player. (new F-cards). All undo operations must be verified.

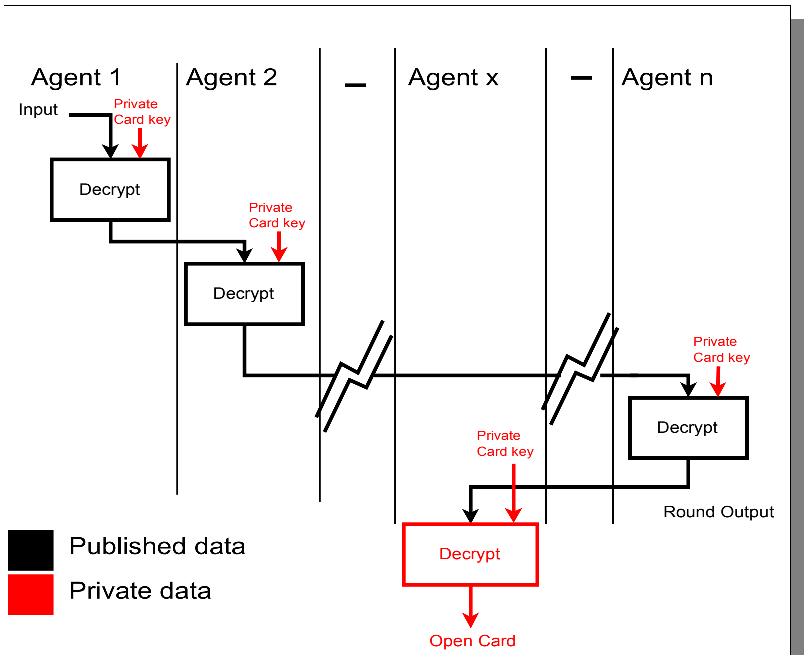

Figure 7: Partial undo method to deal a card

2.15. Master Card Keys

The set of all card keys for a given F-Card are again multiplied by the player receiving the card, creating a Master Card Key for that card. Only the player who is receiving the card can compute the master key, because he is keeping his own card key secret. When a showdown takes place, he can show the master card key to proof its authenticity. Because the underlying cipher is resistant to chosen plaintext attack, a player has no way to compute a valid master key that decrypts a fully encrypted card to an O-Card without going through the dealing protocol.

2.16. Unverified Computations

In VSM-VL, VSM-VPUM and VSM-VL-VUM, all rounds are verified so cheating can not occur due to adulterated computations. VSM-L-OL verifies the shuffle-masking round immediately, as in the other protocol verified rounds, but does not immediately verify the following locking rounds. The locking rounds are verified at the end of the game. Adulterated computations can occur during the locking rounds, so suicide cheating is possible. Nevertheless this fact does not decrease its core security and its ability to identify the cheater.

2.17. Card Transfers

If a player wants to give a card to another player, he publicly sends the F-Card and privately sends the master key. All players must log who is the new holder of the F-Card, because only the holder of an F-Card can lately show its associated open card.

2.18. Suicide Cheaters

Because of the partial unmasking round in VSM-VPUM, this protocol trivially provides protection against suicide cheaters: every operation can be immediately verified.

On the contrary, in VSM-L-OL, VSM-VL-VUM and VSM-VL, card keys are published during deal. These card keys cannot be directly verified. The card holder can detect cheating if the master key obtained cannot correctly decrypt the free card into an open card. If the card holder suspects cheating, then a Card-Key-Verification protocol must be executed. The Card-Key-Verification protocol is just an UVP for the shuffle-masking and locking rounds (with product keys). The card key verification protocol can detect who is cheating without asking for a full private keys disclosure.

The protocol VSM-L-OL does not resist a suicide cheater willing to change his own cards, in the general case.

2.19. VSM-VPUM trick

In VSM-VPUM there is no locking round during shuffling phase. Shuffling phase consist of only a shuffle-masking round. A partial unmasking round is done when a card need to be dealt, and all players except the card holder sequentially unmask the card until they compute the free card which is the last published output of the round. The holding player masking key for that card becomes the free card decryption key (Master key). Because masking keys cannot be published without disclosing all private information, these free cards should be tagged by the holder as "private key". To show a "private key" free card, the card holder can do one of two things:

1. Use an LVP from the O-Card to the F-Card or
2. Re-encrypt the F-Card into a new F-Card and publish the new F-Card. This operation must be verified by a LVP. Then the new free card master key can be published.

Also note that after a hand shuffle, "private key" cards are disposed and new "non-private key" cards are created, so a hand shuffle before showing cards may suffice.
The only drawback in VSM-VPUM is that the unmasking round cannot be pre-computed, while in VSM-VL-VUM it can.

2.20. VSM-L-OL, The Fastest Dealing Protocol

VSM-L-OL is a protocol specifically optimized for static games.

A static game is a game where:

a. There are no card transfers
b. There no need to hide the dealing order of a card when it's shown.
c. The game ends with a showdown of some of the player's hand cards.

An example of a static game is Texas hold'em.

We'll say that a player outputs an adulterated card value x, if x cannot be obtained by the normal execution of the steps in the protocol in the MPF security model. In VSM-L-OL, the locking rounds are not immediately verified. Any player can provide an adulterated value as if it were a result of his computations during the locking rounds. Any adulterated card value will be detected during the game when a player tries to decrypt the card and it decrypts to an invalid O-Card. A player receiving an invalid card must request all players to execute a card key verification protocol and then the cheater will be inevitably detected (or the player having raised a false alarm). A cheater may go undetected in the game if:

1. The adulterated card was never dealt
2. The adulterated card was dealt to the same player having cheated or a colluding player, and that player kept the adulterated card in his hand or transferred it to a colluding player, but the card was not shown during the game.

In either case, the adulteration cannot give the malicious players any advantage (on the contrary) and the adulteration cannot have any measurable consequence for the rest of the players.

Assuming the verification operations are the most time consuming, VSM-L-OL can achieve the lowest possible latency from the start of the game to the showdown on a static game: only one verified encryption per card.
And as in VSM-VL and VSM-VL-VUM, shuffling can be pre-processed so that each dealt card requires only that players publish the card key, with no computing taking place.
The VSM-L-OL base protocol is protected against suicide cheaters for static games. For other kinds of games, the protocol is not protected from suicide cheaters willing to change their own cards and possibly transfer the cards to other players. The cheating will be detected at the end of the game.

2.21. Card Deal Preparation Phase

In VSM-L-OL, VSM-VL and VSM-VL-VUM, dealing can be pre-processed so that each card dealt requires only that a subset of the players publish their keys, with only one decryption taking place. Theses protocols include a “Card Deal Preparation Phase”. Each card to be dealt requires a preparation phase. All cards or a subset of cards can be prepared just after a shuffle and some others can be delayed until new cards needed to be drawn.

There are situations where this can be a great advantage:

a. If a certain set of cards in the deck will always be dealt, then those cards can be prepared along the shuffling protocol. This can improve the network bandwidth use because the preparation of the cards can be done simultaneously for each player, reducing the number of packets transferred. Afterwards dealing is a constant time operation for the prepared set of cards.

b. if players know in advance they are playing multiple games together, multiple shuffles (including shuffle-masking and locking rounds) can be precomputed.

2.22. Card showdowns

To show a card is to reveal the open-card value associated, and to prove that the card was legally dealt. A player can show a card by two methods:

a) Publishing the master key used to decrypt the F-card to an open card (but only if the key is not also a masking key).

b) Using an LVP or UVP to prove that an F-card can be decrypted to an specific open card, without revealing the key.

2.23. Advanced Card Operations

VSM-L-OL, VSM-VL and VSM-VL-VUM provide advanced card operations, such as to put a card back into the deck and re-shuffle the deck. A set of free cards can be put back in the deck by a a sequence of operations. First the deck is re-masked. Then the master-key for cards to return to the deck is changed to the new masking key (all cards share a single key again). Afterwards those cards are re-masked by the remaining players, with the last masking key. Then the cards are appended to the deck. Finally, the deck is re-shuffled to destroy any trace of the transferred cards. An alternative is to use a variation of the abrupt-drop-out protocol, with no player leaving the game, but without claiming ownership of the cards to return to the deck.

2.24. Abrupt Drop-out tolerance

VSM-VL and VSM-VL-VUM provide abrupt drop-out tolerance. To recover from a drop-out, players execute a recovery protocol. The recovery protocol works as follows: players shuffle-mask the original deck in a round with the a new mask key and creates a new deck. Each player also creates a personal recovery set, which is a shuffle-masked version of the deck, with the same mask key used for the new deck. Then each player vetoes the cards that are in his hand from the new deck. This is done by proving how each hand card can be encrypted to a card in the personal recovery set. The set of vetoed cards in the personal recovery is the

personal veto set. Each personal veto set is then shuffle-masked by the remaining players, with the same masking key used to create the new deck. A global shuffle-masked veto set is constructed by appending all the shuffle-masked veto sets. The global veto set is then excluded from the new deck. This leaves in the deck only the cards that were never dealt and those which were in the quitting player's hand. The protocol time complexity is O((d+c)*n) (for non-interactive verifications) where d is the number of cards dealt, c is the number of cards in the deck and n is the number of players. The protocol requires the calculation of recovery sets. Recovery sets can be precomputed after shuffling, but after a player drops-out, a new recovery set must be calculated.
The computations of the protocol can be reordered in a binary tree structure to provide a slight performance gain when the number of cards are distributed in a non-uniform way between the players.
Note that when using multiple dealing decks, an abrupt recovery protocol does not reestablish the same distributions of cards into those decks. Cards must be split into those decks again, so any information a player may have gained regarding cards that where present in certain decks is lost.

With reference to Fig. 8, a sample execution of Abrupt Drop-out-recovery protocol is illustrated, where two players, through their respective agents, act to recover from an abrupt drop out of a third player, where player 1 has two cards in his hand, player 2 has only one card in his hand, player 3 had 1 card in his hand before dropping out, and there is a single card left in deck. After drop-out recovery, deck contains two cards (a card that was in the deck before and card that was in player 3’s hand).

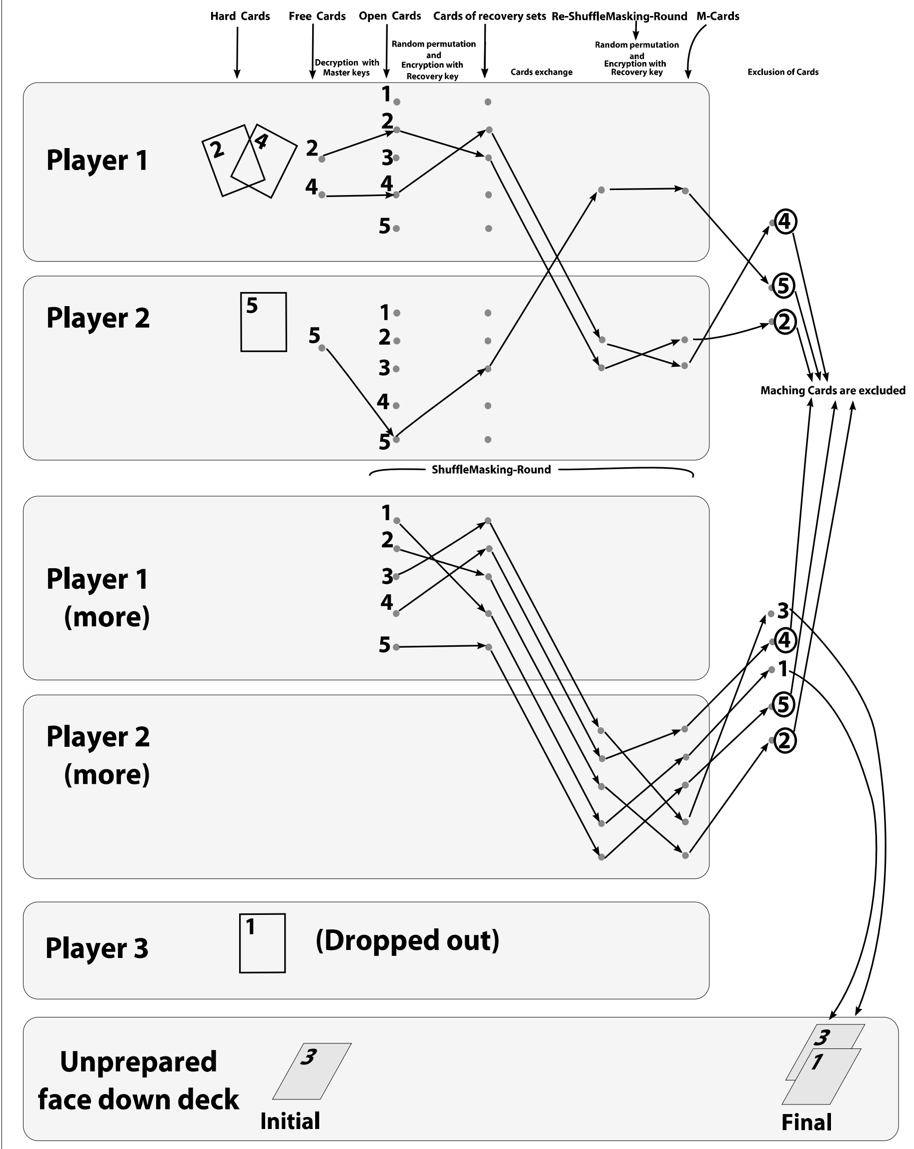

Figure 8: Sample execution of Abrupt Drop-out-recovery protocol

2.25. Duplicated Cards

To use duplicated cards, players follow this protocol:

- At the beginning of the game, for each card value v which has duplicates, players create a deck Dk_v of distinct cards which represented the same value (v).

- To prove ownership of a duplicated card c without revealing any trace, a player creates a set R by a verified-shuffle-masking of the deck Dk_v with a new private masking key. Then proves that c can be encrypted to the card in R whose open-card matches c.

Note that duplicate cards cannot be transferred to another player without revealing the exact open-card that the clone matches, thus leaving a trace of its use.

2.26. Some known CGCs

There are many choices to select a CGC:

- Pohlig-Hellman symmetric cipher over:
    - The subgroup of kth residues modulo a prime p, where $(p - 1) / k$ is also a large prime (a Schnorr group). For the case of $k = 2$, this corresponds to the group of quadratic residues modulo a safe prime.
    - The set of quadratic nonresidues modulo a safe prime.
- Exponentiation over any Galois field where DDH assumption holds.
- Elliptic curve Pohlig-Hellman.
- RSA or other factoring based cryptosystems. Exponentiation over the cyclic group of order $(p - 1)(q - 1)$, where p and q are safe primes.

Note that the listed cryptosystems are malleable, so modified protocols, as shown in section 7, must be used.

3. MPF formal definition

We'll define abstract classes and operations a MP protocol based on MPF should implement.

3.1. Definitions

- **N**: The number of players
- **C**: The number of cards in the deck.
- **H(x)**: a binary string that is a cryptographic hash of the message x.
- **CSPRNG**: A cryptographically secure pseudo random number generator.
- **$E_k(x)$**: The symmetric encryption of plain-text x with key k.
- **$D_k(y)$**: The symmetric decryption of cipher-text y with key k.

3.2. Types

- **Block:** A binary value suitable for encryption/decryption with the CGC.
- **Plain-text**: a Block that is provided for encryption.
- **Cipher-text**: a Block that is the result of an encryption.
- **Key**: a binary value suitable to use as a key for the CGC.
- **Card:** A card is a Block which represents a real card either encrypted or by a known mapping.
- **Open Card or O-Card**: An open card is an public bit string which uniquely maps to a real card of a deck. Each open card must be distinct.
- **Group Encrypted Card:** A group encrypted card is a card encrypted by a key shared between some players, but each player has only a fragment of the group key. The decryption process requires the same players to participate. Because the underlying cryptosystem is commutative, group encrypted cards are obtained by sequentially encrypting the card by each player with each players secret key. A card encrypted by only one player is already a group-encrypted card.
- **Masked Card or M-Card**: A masked card is a group encrypted card that was obtained sequentially encrypting with each player masking key. Each player has one masking key used for all encrypted cards.
- **Locked Card or L-Card**: A locked Card is a group encrypted card, encrypted sequentially with the locking keys. Each player must pick a locking key for each card of the deck.
- **ML-Card**: A Locked and Masked card.
- **Complete M/L/ML Card**: A complete M/L/ML card is a M/L/ML-card where all the players in the game have taken part on the encryption process.
- **Free Card or F-Card:** An card which has been dealt to a player.
- **Master key:** A key that decrypts a free card into an open card.
- **Card Key:** The product of all locking keys and masking keys used to encrypt a card specific card. A card key represents the share of the master key for a specific card that a player has.
- **A deck**: a list of Cards, either Open, Masked or Locked.
- **A hand:** a set of locked/masked/ML cards received by a player, whose values are public (F-cards) and whose associated open cards are known to the player holding the cards.
- **Representative**: A pair (p, c) for which p is any value and c is p encrypted with the key k. Representatives are used for the verification of decryption rounds and ML keys.
  - Definition: Representative = record { p,c : Card } , where $E_K(p)=c$
- **Player**: An integer value that uniquely identifies each player.
- **Round Number**: An integer value that uniquely identifies each round.

3.3. Private Data Structures

- **Card-Holder**: a data structure that maps every F-Card to a player or the main deck (if the card is not mapped).
  - Definition: Card-Holder **: (**Map[F-Card] → Player)

- **Card-Trace**: a data structure that maps a round number and a round output card to the index of the

card in the round input output/list. It's only used in VSM-L-OL at the end of game.
  - Definition: Card-Trace**:** **(**Map[Round number, F-Card] → Index)

- **Master-Key**: a data structure that maps encrypted cards to the key which is able to decrypt them to open cards. Normally, maps ML-cards to the unlocking and unmasking composite key.
  - Definition: Master-Key : (Map[F-Card] → Key)

- **Mask-Key**: a variable which holds the last used masking key.
  - Definition: Mask-Key :Key

- **Recovery-Set-Key**: a variable which holds the last recovery-set masking key.
  - Definition: Recovery-Set-Key :Key

- **Recovery-Set**: a structure which identifies each players recovery set.
  - Definition: Recovery-Set : (Map[Player] → Card-List)

- **Lock-Key**: a data structure which maps a protocol round number and a card index to locking keys.
  - Definition: Lock-Key : (Map[Round number,Integer] → key)

- **Card-Key**: a data structure which maps ML-Cards to card keys (product of all masking and locking keys previously used to encrypt this card for a single player)
  - Definition: Card-Key : (Map[ML-Card] → key)

- **Main-Deck:** A Variable which holds the main deck of cards.

- **Mask-Representative**: a data structure which holds the representative of the last shuffle-masking round for each player.
  - Definition: Mask-Representative : (Map[Player] → Representative)

- **Recovery-Mask-Representative**: a data structure which holds the representative of the recovery shuffle-masking round for each player.
  - Definition: Recovery-Mask-Representative : (Map[Player] → Representative)

- **Lock-Representative**: a data structure which holds the representative of the a locking round number, for a certain index, for each player
  - Definition: Lock-Representative : (Map[Round number, Player, Integer] → Representative)

- **Open-Deck**: a list of cards which contain all the open-cards generated.

3.4. Miscellaneous Operations

- **RandomNumber**(x :Integer) → bit string
  - Returns a random bitstring of bit-length x.
- **RandomPermutation**(n :Integer) → Permutation
  - Returns a permutation of the integers from 1 to n.
- **RandomCardValue()** → Card
  - Returns a random bitstring suitable as a plaintext for the underlying CGC.
- **RandomKey()** → Key
  - Returns a random bitstring suitable as a key for the underlying CGC.
- **CreateCard**(binary string x) → **Card**
  - Creates a valid card that contains the binary string x or a cryptographic hash of x, if x is too long to fit into the card.
- **Product**(collection: expression :Key) → Key: Multiply all key values that result from evaluating each expression from the collection given.
- **Union**(collection: expression :Card) → Card-List: Returns a list which is the union of the lists that result from the evaluation of each expression from the collection given.
- **(exclusion operator)** X : Card-List **–** Y :Card-List → Card-List : Returns a card list of all the elements in X that are not included in the list Y.

3.5. Operations on Cards

The card operations transform one bit string representing a card (either Open, Masked or Locked) into other bit string representing other type of card. Card operations are private and involve one participant.

- **LockCard**(c :Card, k : Key) → Locked-Card
  - Lock a card with the player key k.
  - LockCard(c,k) = E_k(c)
- **EncryptCards(**X:Card-List, k: Key) → Masked Card-list
  - Encrypts with the key k each card of X.
  - for each x = X[s] , EncryptCards(X,k)[s] = E_k(x)
- **DecryptCards(**X :Card-List , k :Key) → Card-list
  - Decrypts with the key k each card of X.
  - for each x = X[s] , DecryptCards(X,k)[s] = D_k(x)
- **PermuteCards(**X :Card-List , F :Permutation) → Card-list
  - Permutes the cards in X with the permutation function F.
  - for each s, PermuteCards(X,F)[s] = X[F(s)]
- **ShuffleMaskCards**(X :Card-List , k :Key , F :Permutation) → Masked Card-list
  - Encrypts with the key k each card of X, and then applies permutation F on the resulting list.
  - for each s, ShuffleMaskCards(X,k,F)[s] = E_k(X[F(s)]))
- **UnLockCard**(c :L-Card , k :Key) → Card
  - Unlocks a card c with the key k.
  - UnLockCard(c,k) = D_k(c)
- **UnMaskCard**(c :M-Card , k :key) → Card
  - Unmask a card with the key k.
  - UnMaskCard(c,k) = D_k(c)
- **OpenCard**(c :F-Card, k :Key) → Card
  - Decrypt a card with the key master key k.
  - OpenCard(c,k) = D_k(c)

Note that UnLockCard, UnMaskCard and OpenCard implement the same operation, and are defined separately as a mean to guarantee a precondition on the classes of cards accepted as input (Masked, Locked or Free).

3.6. Introduction to MPF base protocols

We'll first show the four MPF base protocol graphically, along with the pros and cons for each protocol. Base protocols specify the way cards are shuffled, dealt and opened.

A MPF game has up to 5 main stages:

1. **Shuffle**: All cards are shuffled.
2. **Deal Card Preparation**: A subset of the cards can be prepared to be dealt very quickly. This stage can be executed along the shuffle stage, delayed until a card needs to be dealt or a mixture of both.
3. **Deal Card**: A card is dealt to a specific known player.
4. **Prove Ownership**: A card is opened and the player who was holding the card proves it is legitimate.
5. **End of Game**: Some additional verifications takes place to assure an honest game has taken place.

Stages 2-4 can be repeated or interleaved with other card operations, such as card transfers.

3.6.1. VSM-L-OL

Fig. 9 illustrates the VSM-L-OL protocol, discussed below.

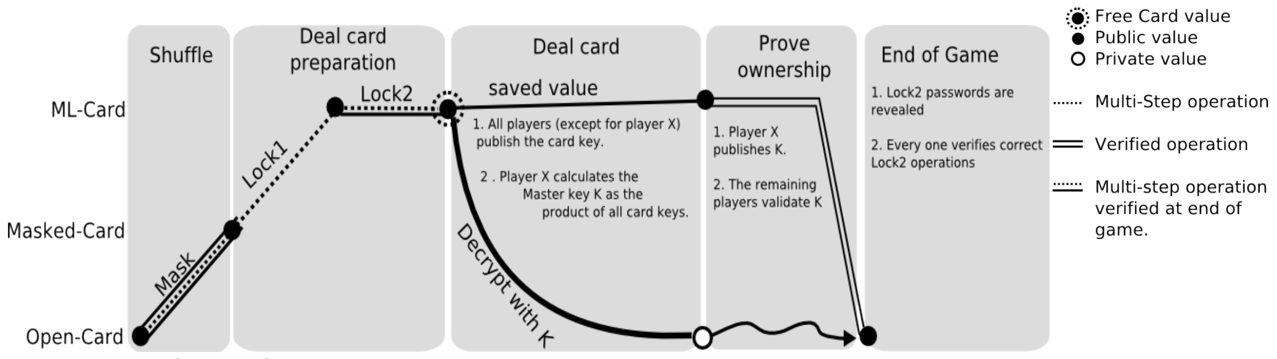
Shuffle
Deal card preparation
Deal card
Prove ownership
End of Game
ML-Card
Masked-Card
Open-Card
Mask
Lock1
Lock2
saved value
1. All players (except for player X) publish the card key.
2 . Player X calculates the Master key K as the product of all card keys.
Decrypt with K
1. Player X publishes K.
2. The remaining players validate K
1. Lock2 passwords are revealed
2. Every one verifies correct Lock2 operations
Free Card value
Public value
Private value
Multi-Step operation
Verified operation
Multi-step operation verified at end of game.

Figure 9: VSM-L-OL protocol

Pros:

- 1 verification round required
- Fastest variant for static games
- Provides advanced card operations
- All free cards are equally treated.
- Protected against suicide cheating for static games.

Cons:

- For non-static games, the protocol is not protected against suicide cheating.
- Is not abrupt drop-out tolerant.

Security Proof (simplified)

Any player can provide adulterated values as results for his calculations on Lock1 or Lock2 rounds without being immediately detected. Because Lock2 keys are revealed at the end of the game, cheating during Lock2 round is a suicide cheat. The only way to cheat is to do it during the Lock1 round or during a card deal.
Let's suppose there is a cheating group (which is a proper subset of all the players) consisting of at least the player Mallory and possibly Marvin. Suppose no one tries to cheat at Lock2 round but player Mallory tries to cheat during a Lock1 round providing a chosen card value c as output, instead of the re-encryption of the value x being received. Finally, after all re-encryptions of Lock1 and Lock2, the value c is going to be dealt, and converted into a free card y.

First suppose the card is dealt to a player Marvin taking part on the cheat. He won't complain on any invalid value obtained during the calculation of the master key. He will always accept the card. Later the player will try to successfully show the card. To do it, the player must posses a valid master key.
To obtain a master key k, Marvin must solve the equation:

$D_k(y) = x$
$y = E_w(c)$

Knowing:

q = Product(for all i: q_i), where q_i is the card key of player i.
w = Product(for a subset of the players i: w_i), where w_i is the lock2 key for player i.

So $D_{k*(w^{-1})}(c) = x$

Because of CUOC, the only way to obtain a valid open card x is that c is itself an encryption/decryption of the same open card x. Let $c = E_s(x)$ for some s. The value s must be fixed before the Lock2 keys have been used.

So $D_{k*(w^{-1})*(s^{-1})}(x) = x$

The only way to solve the equation is taking k = w*s. But L cannot be derived from q, because q values contain Lock2 values multiplied by the unknown keys for the Lock1 and shuffle-masking rounds. The only way to solve the equation is to do a KPA on the CGC, which is computationally infeasible under the MPF model.

Mallory cannot tamper with the dealing protocol, because the protocol security relies on the unknown Lock2 keys used to encrypt the card to be dealt. Any attempt to cheat during the dealing protocol will result in an invalid open card received, and the cheater will be caught.

3.6.2. VSM-VL

Fig. 10 illustrates the VSM-VL protocol, as further described below:

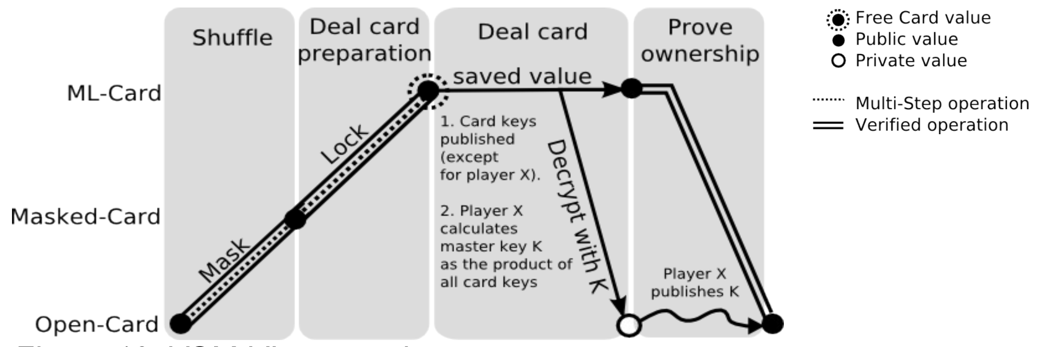

Figure 10: VSM-VL protocol

Pros:

- Provides advanced card operations
- All free cards are equally treated
- Protected against suicide cheating.
- Abrupt drop-out tolerant.

Security Proof (simplified)

1. All computations are verified.
2. It's infeasible to compute the key m or l from a key q such as q=m*l and a known message encrypted with m and later with l, under the MPF model.

Fig. 11 illustrates a VSM-VL shuffle between agents.

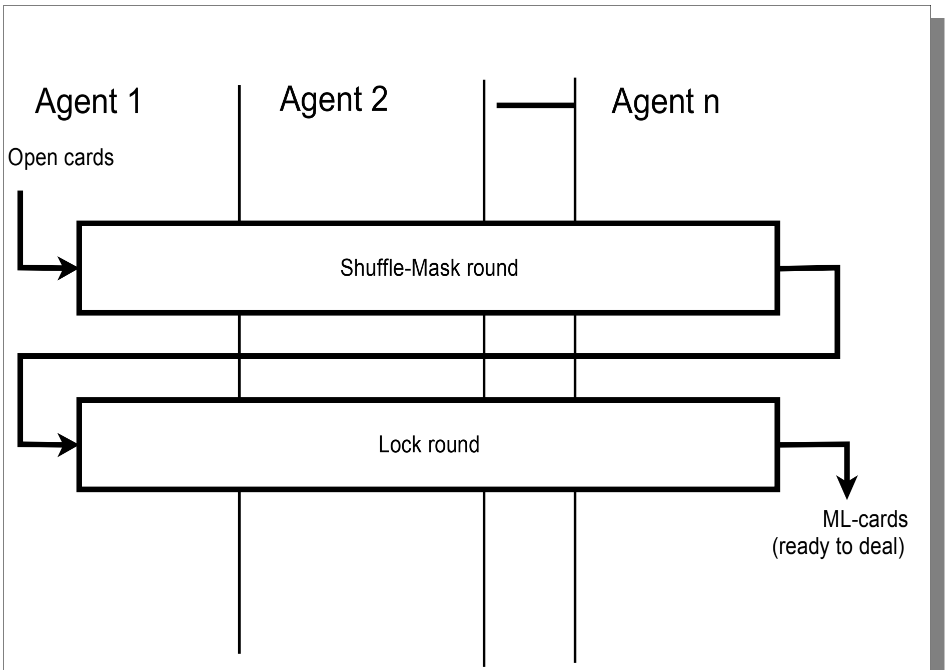

Figure 11: VSM-VL shuffle between agents

3.6.3. VSM-VPUM

Fig. 12 illustrates the VSM-VPUM protocol, as further described below.

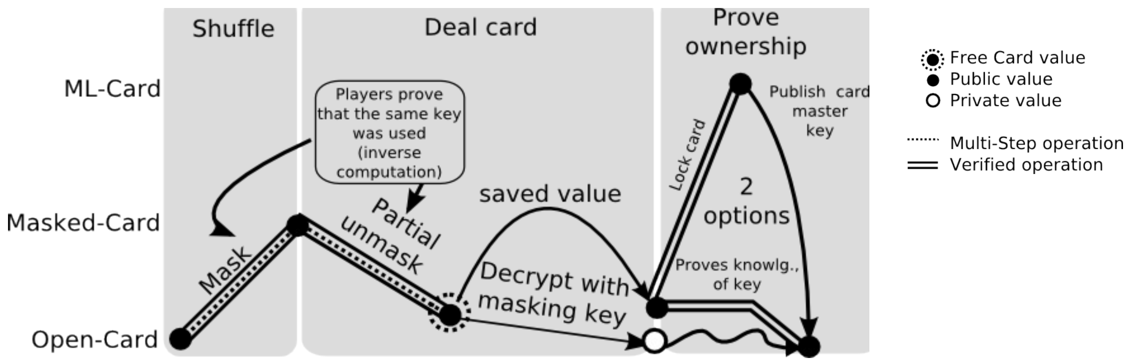

Figure 12: VSM-VPUM protocol

Pros:

- Protected against suicide cheating.

Cons:

- 2 verification rounds required
- Initial free cards must be carefully treated in a special way (because they are encrypted with a “private key”)
- The unmasking round cannot be pre-calculated.
- Does not provide advanced card operations

Security Proof (simplified)

1. All computations are verified.
2. If a card is locked afterwards, it's infeasible to an opponent to compute the key m or l from a key q such as q=m*l and a known message encrypted with m and later with l, under the MPF model.
3. If a card is opened by proving knowledge of m using a LVP, it's infeasible to an opponent to compute m.

3.6.4. VSM-VL-VUM

Fig. 13 illustrates the VSM-VL-VUM protocol, as further described below.

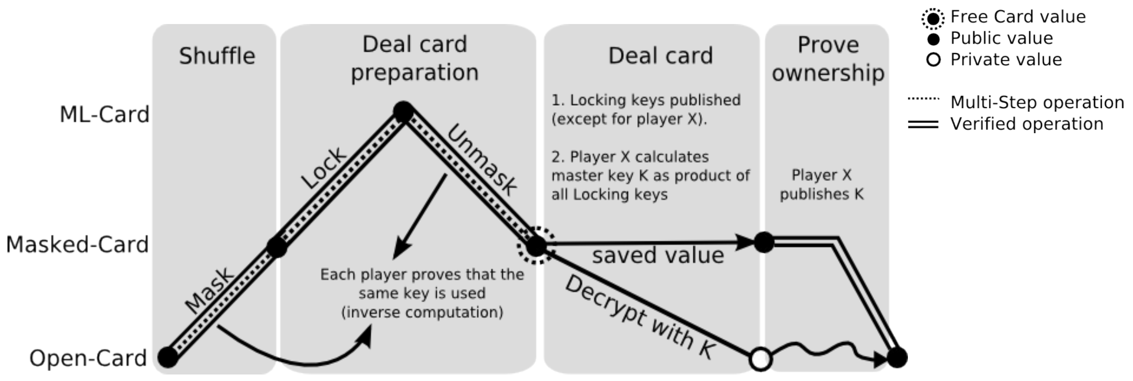

Figure 13: VSM-VL-VUM protocol

Pros:

- All free cards are equally treated
- Provides advanced card operations
- Protected against suicide cheating
- Abrupt drop-out tolerant.

Cons:

- 3 verification rounds required

Security Proof (simplified)

1. All computations are verified.
2. it's infeasible to an opponent to compute the key I from a known message encrypted with I (a KPA attack), under the MPF model.

Fig. 14 illustrates the execution of all the rounds between agents under the VSM-VL-VUM protocol.

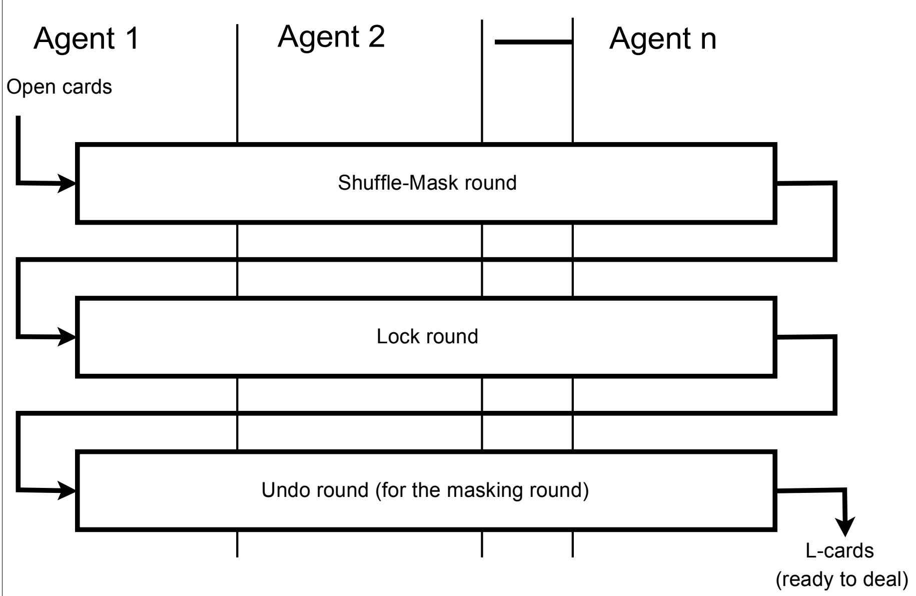

Figure 14: Execution of all the rounds under the VSM-VL-VUM protocol

3.7. Card Protocols

Protocols involve more than one participant. We'll define some basic sub-protocols which constitute the building blocks of the base protocols. We'll say that a player “posses”, “has” or “holds” an open card x (or a free card y) if the player has a mapping (y → k) in his Master-Key table and $D_k(y)=x$.
For simplicity of the definitions, we'll treat card lists as sets when needed, allowing set operations (membership test, inclusion and exclusion) on lists, taking into account that no card list can have duplicates due to the CUOC assumption. Because the deck has very few cards compared to the encryptor plaintext size, and the number of computations is bounded on the number of cards, players and card transfers, the probability that duplicates appear spontaneously during computations is negligible. Anyway, players can check the lists after each shuffle to ensure no duplicates exists and redo the step, changing any random value used, if a duplicate is found.
Protocols have arguments, which are described in the protocol signature. Arguments can be in (input) or out (output). Also, arguments are public (meaning the argument is known by all players) or private (meaning the argument is only known to a certain player, and unless broadcast, remain private during the protocol). Arguments can also be multi-private, meaning that each player receives a private copy of the argument. To specify which copy is referred in the protocol steps, a subscript with the player number is used. Public arguments values are checked by all players and they cannot differ. Every player must supply exactly the same argument. Public arguments are meta-arguments and do not need to be really transferred, but can be broadcast to assert all players are willing to do the same operation.
Sub-protocol executions are requested and expected unconditionally by all players, with the exception of the Card-Key-Verification protocol, which is conditional.
Values can be transferred privately from one player to another or broadcast to all the remaining players. Values transmitted always are referred with an underscored variable name. Underscored variables are meta-variables created for the only purpose of tracking values as they are transferred through the network. Meta-variables can change their value after they have been broadcast, and is assumed that all players can perform the same change at the same time. Because of this notation, protocols described do not make use of “receive x” commands. Public values are automatically received by other players in meta-variables with the same name as the ones sent. The input and output arguments of sub-protocols can be meta-variables, so are passed as reference, which means that the actual values are not directly stored but refer to some values previously broadcast or calculated by all players. Private variables can also be passed to output multi-private arguments. In such a case each player stores the result privately in his own local variable. An explanation of the meaning of each letter argument modifier is provided in table 4. A quick reference of all card protocols is shown in table 5.

public	Argument is know to all players
private	Argument is know only to a single player
multi-private	Every player gets or sets a private copy of the argument.
in	The argument is an input to the protocol
out	The argument is an output from the protocol

Table 4: Argument modifiers reference

Deck creation

3.7.1. Create-Deck (CO-PRNGP)
3.7.2. Create-Deck (Locking)

Deck shuffle and preparation

3.7.3. Shuffle-Deck
3.7.4. Prepare-Cards-To-Deal (for VSM-L-OL)
3.7.5. Prepare-Cards-To-Deal (for VSM-VL)
3.7.6. Prepare-Cards-To-Deal (for VSM-VPUM)
3.7.7. Prepare-Cards-To-Deal (for VSM-VL-VUM)
3.7.8. Verified-Unmasking-Round
3.7.9. Verified-ShuffleMasking-Round
3.7.10. Locking-Round

Card deal

3.7.13. Multiple-Cards-Deal (for VSM-VPUM)
3.7.14. Single-Card-Deal (for VSM-L-OL, VSM-VL and VSM-VL-VUM)

Shuffle hand cards

3.7.11. Shuffle-Hand(1)
3.7.12. Shuffle-Hand(2)

Protect against suicide cheaters

3.7.15. Card-Key-Verification (for VSM-VL-VUM)
3.7.16. Card-Key-Verification (for VSM-VL and VSM-L-OL)
3.7.17. Build-Representative

Show cards

3.7.18. Show-Cards(1)
3.7.19. Show-Cards(2)

Deck reshuffle

3.7.20. Reshuffle-Deck (for VSM-L-OL, VSM-VL and VSM-VL-VUM)
3.7.21. Change-Hand-Cards-Key

Card transfers

3.7.22. Return-Cards-To-Deck (for VSM-L-OL, VSM-VL and VSM-VL-VUM)
3.7.23. Private-Cards-Transfer
3.7.26. Put-Card-On-Table
3.7.27. Verified-ShuffleRemasking-Round

Abrupt-Drop-out resistance

3.7.24. Abrupt-Drop-out-Recovery (for VSM-VL and VSM-VL-VUM)
3.7.25. Create-Recovery-Sets

Game finalization

3.7.28. End-Of-Game (only for VSM-L-OL)

Table 5: Card Protocols Quick Reference

3.7.1. Create-Deck (CO-PRNGP)

Create a fresh deck of unique cards by a CO-PRNGP.

Signature: Create-Deck

1) Each player i :
    1.1) Chooses a random number r_i := RandomNumber(csprng-seed-bit-length)
    1.2) Computes cr_i := $H(r_i)$

1.3) Broadcasts $\underline{cr_i}$ (a commitment to r_i)
2) Each player i:
    2.1) Broadcasts $\underline{r_i}$
    2.2) For each j, verifies that $\underline{cr_j}$:= H($\underline{r_j}$)
    2.3) Computes S = H($\underline{r_1}$;$\underline{r_2}$;..;$\underline{r_n}$)
    2.4) Uses S as seed for a common CSPRNG.
    2.5) Use CSPRNG to generate the symmetric algorithm (CGC) common parameters.
    2.6) Uses the CSPRNG to generate c distinct suitable encodings of the real cards in a deck to be used as open cards. The generated cards are saved in the Open-Deck list.
    2.7) Computes dh_i := H(Open-Deck).
3) The first player broadcasts $\underline{dh_1}$.
4) Everybody verifies having computed dh_i equal to $\underline{dh_1}$. If a player detects a mismatch, the protocol aborts.

- **After:**
    - Open-Deck is a set of O-Cards.

3.7.2. Create-Deck (Locking)

Create a fresh deck of unique cards with a locking round.

Signature: Create-Deck

1) Players constructs the card list X containing C copies of a single fixed card value g.
2) Players execute the protocol Locking-Round (X, $\underline{Y}$,1, true)
3) Each player i:
    3.1) Sets Main-Deck :=$\underline{Y}$
4) Every player:
    4.1) Empties the Lock-Key.
    4.2) Empties the Lock-Representative table

- **After:**
    - Open-Deck is a set of O-Cards.

3.7.3. Shuffle-Deck

A deck of cards is mixed by all the players so that anyone is assured nobody can predict the position of an input card on the output deck.

Signature: Shuffle-Deck

- Before:
    - Open-Deck is a list of O-Cards.
    - Each player's Main-Deck list is empty.

1) Players execute the protocol Verified-ShuffleMasking-Round (Open-Deck, $\underline{Y}$).
2) Each player i:
    2.1) Sets Main-Deck :=$\underline{Y}$

- **After:**
    - The Main-Deck is ready for a card preparation for dealing.

3.7.4. Prepare-Cards-To-Deal (for VSM-L-OL)

Some cards are prepared be be dealt. This protocol tag cards from Main-Deck.

Signature: Prepare-Cards-To-Deal (public in $\underline{v}$:Integer)

- Before:
    - Open-Deck is a list of O-Cards.
    - The meta-variable v is the number of cards to prepare.

```
1) Let Z := Copy (Main-Deck, v )
2) Players execute the protocol Locking-Round (Z, L ,1, false)
3) Players execute the protocol Locking-Round (L , Y , 2, false)
4) Each player i:
        4.1) For j from 1 to v :
                4.1.1) Let k := Mask-Key*Lock-Key[1,j]*Lock-Key[2,j]
                4.1.2) Inserts the mapping ( Y[j] → k ) in Card-Key.
                4.1.3) Prepare-Card[ Main-Deck[j] ] := Y[j]
```

- **After:**
    - The Main-Deck is ready for a card dealing protocol.

3.7.5. Prepare-Cards-To-Deal (for VSM-VL)

Some cards are prepared be dealt. This protocol just moves cards from Main-Deck to Prepared-Main-Deck.

Signature: Prepare-Cards-To-Deal (public in v :Integer)

- Before:
    - Main-Deck contains at least v cards.

```
1) Let Z := Copy (Main-Deck, v )
2) Players execute the protocol Locking-Round (Z, Y , 1)
3) Each player i:
        3.1) For j from 1 to v :
                3.1.1) Let k := Mask-Key*Lock-Key[1,j]
                3.1.2) Inserts the mapping ( Y[j] →  k ) in Card-Key.
                3.1.3) Prepare-Card[ Z[j] ] := Y[j]
```

- **After:**
    - The first v cards of the Main-Deck are prepared for dealing.

3.7.6. Prepare-Cards-To-Deal (for VSM-VPUM)

Some cards are prepared be dealt. This protocol just moves cards from Main-Deck to Prepared-Main-Deck.

Signature: Prepare-Cards-To-Deal (public in v :Integer)

- Before:
    - Main-Deck contains at least v cards.

```
1) Let Z := Copy (Main-Deck, v )
2) Each player i:
        2.1) For each z in Z:
                2.1.1) Inserts the mapping ( z →  Mask-Key ) in Card-Key.
                2.1.2) Prepare-Card[ Z[j] ] :=Z[j]
```

- **After:**
    - The Prepared-Main-Deck is ready for a card dealing protocol.

3.7.7. Prepare-Cards-To-Deal (for VSM-VL-VUM)

Some cards are prepared be dealt.

Signature: Prepare-Cards-To-Deal (in v :Integer)

- Before:
    - Main-Deck contains at least v cards.

```
1) Let Z := Copy (Main-Deck, $\underline{v}$ )
2) Players execute the protocol Locking-Round(Z, $\underline{L}$, 1 , true).
3) Players execute the protocol Verified-Unmasking-Round( $\underline{L}$, $\underline{Y}$ , -1)
4) Each player i:
        4.1) For each j from 1 to $\underline{v}$ :
                4.1.1) Inserts the mapping ( $\underline{Y}[j] \rightarrow$ Lock-Key[1, j] ) in Card-Key.
                4.1.2) Prepare-Card[ Z[j] ] := $\underline{Y}[j]$
```

- **After:**
  - The Main-Deck is ready for a card dealing protocol.

3.7.8. Verified-Unmasking-Round

This parametrized protocol implements a verified unmasking round.

Signature: Verified-Unmasking-Round(
    public in $\underline{L}$: ML-Card-List,
    public out $\underline{Z}$: L-Card-List,
    public in <u>skip-player</u> :Integer)

- Before:
  - L is a list of ML-Cards.

```
1) For each player i
        1.1) If i =0 then $\underline{X}_0$ =L else $\underline{X}_i$ =$\underline{Y}_{i-1}$
        1.2) Constructs a card list $Y_i$ := UnmaskCards( $\underline{X}_i$, Mask-Key )
        1.3) Broadcast $\underline{Y}_i$.
        1.4) Let r := Mask-Representative[i].
        1.5) Players execute  UMVP(  Mask-Key , $\underline{X}_i$  , $\underline{Y}_i$  , i ,   r.p , r.c )
2) $\underline{Z}$  = $\underline{Y}_p$ (where p is last player in the round ).
```

- After:
  - Z a list of L-Cards.
  - Z is public.

Note that the UMVP protocols in step 4 can be executed in parallel to increase the performance for multi-threading or multi-core CPUs.

3.7.9. Verified-ShuffleMasking-Round

This protocol implements a verified shuffle-masking round.

Signature: Verified-ShuffleMasking-Round**(**
    public in $\underline{D}$: O-Card-List,
    public out $\underline{Z}$: M-Card-List)

- Before:
  - D is a list of O-Cards.
  - R is a representative

```
1) For each player i, in increasing order:
        1.1) If i=0 then $X_i$ =$\underline{D}$ else $\underline{X}_i$ =$\underline{Y}_{i-1}$
        1.2) Set F :=RandomPermutation(#D).
        1.3) Set m :=RandomKey()..
        1.4) Constructs a card list $Y_i$ := ShuffleMaskCards( $\underline{X}_i$, m , F )
        1.5) Broadcast $\underline{Y}_i$.
        1.6) R.p :=RandomCardValue()
        1.7) R.c :=MaskCard(R.p, m);
        1.8) Broadcasts R
        1.9) Players execute SMVP ( m , $X_i$, $Y_i$ , i ,R )
        1.10) Sets Mask-Key := m.
        1.11) All players set Mask-Representative[i] :=R
2) Z = $Y_p$ (where p is the last player in the round )
```

- After:
  - Z a list of M-Cards.
  - Z is public.
  - Z is a permutation of the cards in X, after masking.
  - The permutation cannot be computed by any proper subset of players.
  - Each player has his masking key saved.

3.7.10. Locking-Round

This parametrized protocol implements a verified shuffle-masking round.

```
Signature: Locking-Round (
        public in L : ML-Card-List,
        public out Z: L-Card-List,
        public in lock-round :integer,
        public in verified :boolean)
```

- Before:
  - L is a list of Cards.

```
1) For each player i, in increasing order:
        1.1) If i=0then $X_i$ =L else $X_i$ =$Y_{i-1}$
        1.2) Chooses a random or pseudo-random key-List K: For j from 1 to #L, set K[j] :=RandomKey()
        1.3) Constructs a card list $Y_i$ := LockCards( X, K )
        1.4) Broadcast $Y_i$.
        1.5) If verified then players execute LVP( K , $X_i$, $Y_i$ , i )
        1.6) If lock-round>0 then for j from 1 to #K:
                1.6.1) set Lock-Key[lock-round , j] := K[j]
        1.7) All players:
        1.7.1) For j from 1 to # $X_i$:
                1.7.1.1) If lock-round>0 then
                        1.7.1.1.1) Lock-Representative[lock-round ,i,j] := { p: $X_i$[j], c: $Y_i$[j] }
3) Z = $Y_p$ , where p is the last player in the round (Z is the round output).
4) All players:
        4.1) For j from 1 to # $X_i$:
                4.1.1) Card-Trace[lock-round, Z[j] ] := j
```

- After:
  - Z a public list of L-Cards.
  - Each player has his locking keys saved.

3.7.11. Shuffle-Hand(1)

This is a single key hand shuffling verified by a SMVP that allows a player to mix a subset of the free cards he is holding. The mapping between the newly generated free cards and the previous free cards remains secret.

Signature: Shuffle-Hand(private in X :Card-List, public in i :Integer)

- Before:
  - X is card list of F-Cards
  - X is public.

- Player i will mix his hand cards.

1) Player i:
    1.1) Broadcast X and i.
2) Each player checks, for each x in X, if Card-Holder[x] = i. If not, then player i is attempting to prove ownership for cards not given to him (cheat) and the protocol aborts.
3) Player i:
    3.1) Creates a new temporary key w (w should not be the identity): Set w :=RandomKey()
    3.2) Set F :=RandomPermutation(#X).
    3.3) Computes Y := ShuffleMaskCards(X, w ,F)
    3.4) Broadcasts Y.
4) Players execute SMVP (W, X,Y, i , [ ], [ ]).
5) Now, player i will change his master keys for the set Y. Player i Inserts, for each j the mapping (Y[j] → w*Master-key[X[F^{-1}(j)]]) in Master-key,
6) Player i removes mappings of the set X from his Master-key map.
7) All the players remove the mappings for the set X from their Card-Holder map.
8) All players insert, for each y in Y, the mappings (y → i) in their Card-Holder map.

- After:
  - Y is a set of ML-Cards
  - Y is public
  - Y is a permutation of the locked and masked cards in X.
  - The player shuffling the cards obtains a new set of locking keys for the newly created set Y.
  - Each player has updated his Card-Holder map.

3.7.12. Shuffle-Hand(2)

This is a multi-key hand shuffling verified by a SLVP that allows a player to mix a subset of the free cards he is holding. The mapping between the newly generated free cards and the previous free cards remains secret.

Signature: Shuffle-Hand(private in X :Card-List, public in i :Integer)

- Before:
  - X is card list of F-Cards
  - X is public.

1) Player i:
    1.1) Broadcast X and i
2) Each player checks if Card-Holder[x] = i, for each x in X. If not, then player i is attempting to prove ownership for cards not given to him (cheat) and the protocol aborts.
3) Player i:
    3.1) Creates a list of random keys W, where W[j] ,is the key the will be used to re-encrypt the card X[j] (W[j] should not be the identity). For j from 1 to #X, set W[j] :=RandomKey()
    3.2) Set F :=RandomPermutation(#X).
    3.3) Computes Y := PermuteCards(LockCards(X, W) , F)
    3.4) Broadcasts Y.
4) Players execute SLVP (W, X,Y, i).
5) Player i (changes his master keys for the set Y):
Inserts, for each j, the mapping (Y[j] → W[j]*Master-key[X[F^{-1}(j)]]) in Master-key,
6) Player i removes mappings for the set X from his Master-key map.
7) All players remove the mappings for the set X from their Card-Holder map.
8) All players insert, for each y in Y, the mappings (y → i) in their Card-Holder map.

- After:
  - Y is a set of ML-Cards
  - Y is public
  - Y is a permutation of the locked and masked cards in X.
  - The player shuffling the cards obtains a new set of locking keys for the newly created set Y.

- Each player has updated his Card-Holder map.

3.7.13. Multiple-Cards-Deal (for VSM-VPUM)

A player is given a set of n free cards, and the corresponding open cards.
This protocol is defined for multiple cards to take advantage of performance gain calling Verified-Unmasking-Round for multiple cards at once.

```
Signature: Multiple-Cards-Deal(
        public in v :Integer,
        public in i :Integer)
```

- Before:
  - The prepared deck has at least n cards available.
  - The player i wants to get n cards from the deck.

```
1) Player i broadcasts v and i.
2) All players:
        2.1) Set Q := Copy(Main-Deck, v).
        2.2) Set X := []
        2.3) For each q in Q, append Prepared-Card[q] to X.
3) Players execute Verified-Unmasking-Round(X , Z, i).
4) Player i:
        4.1) Computes Y := DecryptCards(Z, Mask-Key)
        4.2) Checks that each card in Y is a valid open card. if not then aborts this protocol.
        4.3) For each card y in Y, inserts the mapping ( y → Mask-Key ) in his Master-key map (y is a
        "private key" card).
5) Every player:
        5.1) For each card z in Z, sets Card-Holder[z] := i. (Z is the set of free cards obtained)
        5.2) For each card x in X, deletes the card x from the Main-Deck.
```

- After:
  - The player i receives a list of O-Cards Y (private to that player)

3.7.14. Single-Card-Deal (for VSM-L-OL, VSM-VL and VSM-VL-VUM)

A player receives a free card, where the related open card is revealed only to him.

Signature: Single-Card-Deal(public in i :Integer)

- Before:
  - The prepared main deck has at least one card available.
  - The player i wants to get a card from the prepared main deck.

```
1) Player i broadcasts i.
2) All players:
        2.1) Set x := Prepared-Card [ Main-Deck[1] ].
3) For each player t, such as t <> i, in increasing order:
        3.1) Sets $q_t$ := Card-Key[x]
        3.2) Broadcasts $q_t$
4) Player i:
        4.1) Computes w := $q_1$* .. * $q_n$. (the q values broadcast by the players)
        4.2) Computes y := OpenCard(x,w) = $D_w(x)$
        4.3) Checks that y is a valid open card, if not then aborts this protocol, proclaims a cheating attempt
        and do the following steps:
                4.3.1) Let Q = [$q_1$, .. , $q_n$] and $q_i$ is undefined.
                4.3.2) Execute Card-Key-Verification(i,1, Q) protocol.
5) Insert the mapping ( x → w ) in his Master-key map.
```

6) Every player:
    6.1) Sets Card-Holder[x] := i.
    6.2) Deletes the card x (at index 1) from the Main-Deck.

- After:
    - A player gets an open card z (private to that player).

3.7.15. Card-Key-Verification (for VSM-VL-VUM)

This protocol lets a player i who has received invalid card key values to check who is cheating when dealing a card with the index j of the main deck.

Signature: Card-Key-Verification(

public in i :Integer,

public in j :Integer,

public in Key-List Q)

1) For each player p, such as p <> i do

1.1) Set R :=Lock-Representative[1,p, j]

1.2) Every other player checks that $E_{Q[j]}$ (R.p) = R.c. If not, then player p is cheating.

3.7.16. Card-Key-Verification (for VSM-VL and VSM-L-OL)

This protocol lets a player i who has received invalid card key values to check who is cheating when dealing a card with the index j of the main deck.

Signature: Card-Key-Verification(

public in i :Integer,

public in j :Integer,

public in Key-List Q)

1) For each player p, such as p <> i do

1.1) Set R :=Mask-Representative[p]

1.2) Execute Build-Representative(p, R, R, Lock-Representative[1, p, j], Lock-Key[1,j])

1.3) (Only for VSM-L-OL) Execute Build-Representative(p, R, R, Lock-Representative[2,p, j] , Lock-Key[2, j])

1.4) Every other player checks that $E_{Q[j]}$ (R.p) = R.c. If not, then player p is cheating.

3.7.17. Build-Representative

Create a new representative for the union of representatives A and B, for player i.

Signature: Build-Representative(

public in i :Integer,

public out R :Representative;

public in A :Representative;

public in B :Representative;

private in k : Key)

```
1) All players:
    1.1) Compute x := LockCard(A.c,k)
    1.2) Execute RLVP( k , A.c , x ,i , B.p , B.c )
    1.3) Set R.p :=A.p
    1.4) Set R.c :=x
```

3.7.18. Show-Cards(1)

Allows a player to put a set of open cards on the table, and proof the legitimate possession of the cards. Legitimate possession means the card was previously drawn to a player by a deal protocol, and after any number of card transfers, the cards is now in the prover hand. This protocol only publishes master keys so it's very fast.
The protocol is generally preceded by a Shuffle-Hand protocol for all the cards in the player's hand to avoid leaving any trace that points to cards dealt or previously transferred .

Signature: Show-Cards(
public in i :Integer,
private in Y :Card-List)

- Before:
  - Player i wants to open a list of free cards Y and he has the corresponding master keys.

```
1) Player i:
    1.1) Broadcasts i.
    1.2) For each j , from 1 to #Y:
    1.3) Broadcasts a tuple (Y[j], $k_j$) where $k_j$ =Master-key[Y[j]].
2) Each player j, such as j <>i:
    2.2) For each j , from 1 to #Y:
        2.2.1) Check that Card-holder[Y[j]] = i. If the check fails, then player i is trying to cheat.
        2.2.2) Computes c := OpenCard(Y[j], $k_j$)
        2.2.3) Checks that c is a valid O-Card. If the check fails, then player i is trying to cheat.
```

- After:
  - The open cards associated with Y are published.
  - The player i proof legitimate possession of each shown card.

3.7.19. Show-Cards(2)

This protocol allows a player to put a set of open cards on the table, and proof the legitimate possession of the cards. Legitimate possession means the card was previously drawn to a player by the Card Drawing Protocol, and after any number of card transfers, the cards is now in the prover hand. This is a slower protocol that relies on proving the knowledge of the master keys, without publishing them.

This protocol is only faster than Show-Cards(1) protocol if:

a) All the cards in the player's hand will be shown
b) The cards are not released and will be kept in the player's hand
c) Some of the cards will be privately transferred later.

This saves a further Shuffle-Hand protocol execution before the private transfer.
This protocol is generally preceded by a Shuffle-Hand protocol for all the cards in the player's hand to avoid leaving any trace that points to cards dealt or previously transferred.

Signature: Show-Cards(
public in i :Integer,
private in Y :Card-List)

- Before:
  - Player i wants to open a list of free cards Y and he has the corresponding master keys.

1) Player i broadcasts i and Y.
2) Each player j, such as j <> i do:
    2.1) Check that, for each y in Y, Card-holder[y] = i. If the check fails, then player i is trying to cheat.
3) Player i:
    3.1) Construct the key list K. For each index j from 1 to #Y, K[j] := Master-key[Y[j]]
    3.2) Construct the card list X. For each index j from 1 to #Y, X[j] := OpenCard(Y[j], K[j])
    3.3) Broadcast X
4) Players execute SLVP(K, X,Y, i)

- After:
  - The open cards X (associated with Y) are published.
  - The player i proof legitimate possession of each shown card.

3.7.20. Reshuffle-Deck (for VSM-L-OL, VSM-VL and VSM-VL-VUM)

This protocol allows the players to reshuffle the deck, keeping the permutation function unknown to any proper subset of the players. This protocol is available only for VSM-L-OL, VSM-VL and VSM-VL-VUM.

Signature: Reshuffle-Deck

1) Players execute the protocol Verified-ShuffleMasking-Round (Main-Deck, Y).
2) Each player i:
    2.1) Sets Main-Deck :=Y

- After:
  - A deck Y of ML-cards (public)
  - New set of card keys for all cards.

3.7.21. Change-Hand-Cards-Key

This is a protocol that allows a player to simultaneously change the key of a set of hand cards to match the last Mask-Key. This protocol must be used after a Deck-reshuffle by the player willing to put cards back in the deck.

Signature: Change-Hand-Cards-Key(
public in Card-List X,
public out Card-List Y,
public in i : Integer)

- Before:
  - X is card list of F-Cards
  - X is public.

1) Each player checks, for each x in X, if Card-Holder[x] = i. If not, then player i is attempting to prove ownership for cards not given to him (cheat) and the protocol aborts.
2) Player i:
    2.1) Creates a list of keys W: for each 1 <= j <=#X: set W[j] := Master-Key[X[j]]$^{-1}$ * Mask-Key.
    2.2) Computes Y := LockCards(X, W)
    2.3) Broadcasts Y.
3) Players execute LVP (W, X,Y, i).

4) Player i (changes his master keys for the set Y): Inserts, for each j, the mapping (Y[j] → Mask-Key) in Master-key,
5) Player i removes mappings for the set X from his Master-key map.
6) All players remove the mappings for the set X from their Card-Holder map.
7) All players insert, for each y in Y, the mappings (y → i) in their Card-Holder map.

- After:
  - Y is a set of ML-Cards
  - Y is public
  - The player shuffling the cards obtains a new set of locking keys for the newly created set Y.
  - Each player has updated his Card-Holder map.

3.7.22. Return-Cards-To-Deck (for VSM-L-OL, VSM-VL and VSM-VL-VUM)

A player i take some cards from their hands and put them back on the deck. To remove any track of cards the appended to the deck, a Reshuffle-Deck protocol may be required.
This protocol is available only for VSM-L-OL, VSM-VL and VSM-VL-VUM.

Signature: Return-Cards-To-Deck(
    private in Card-List X,
    private in i :Integer)

- Before:
  - Player i wants to return list of cards X to the deck
  - X is a list of F-cards

1) Player i broadcasts X and i.
2) Execute Reshuffle-Deck (creates a new Mask-Key for every player)
3) Execute Change-Hand-Keys(X,Y) (force the key of the cards to put back on the deck to be the new Mask-Key).
4) Player i, for each y in Y, removes all the mappings from y in Master-Key.
5) Each player j, such as j <> i:
    5.1) For each y in Y, verify that Card-Holder[y] = i. If not, then abort the protocol.
    5.2) For each y in Y, remove the all mappings from y in Card-Holder.
6) Execute Verified-ShuffleRemasking-Round(Y,Z,i)
7) All players:
    7.1) Remove all mappings from the Prepared-Card.
    7.2) Append the set Z to the Main-Deck.

- After:
  - The main deck is appended a new list of cards Z.
  - The player i cannot claim he is holding the cards anymore.

3.7.23. Private-Cards-Transfer

A player i gives some cards to a player j privately, but with the approval of the rest of the players.

Signature: Private-Card-Transfer(
    private in Card-List X,
    private in i :Integer,
    private in j :Integer)

- Before
  - X is the list of F-cards cards in player's i hand to give to player j.

1) Player i:
    1.1) Publishes X , i and j.
    1.2) Constructs a key list Q such as, for each t from 1 to #X : Q[t] := Master-Key[X[t]]
    1.3) Sends privately the list Q to player j.

```
        1.4) For each x in X, removes the mappings from x in Master-Key.
2) Player j:
        2.1) Verifies that, for t from 1 to #X : OpenCard(X[t],Q[t]) is a valid open card. If not, then aborts the
        protocol and broadcasts Q[t] to prove player i is cheating.
3) For each t from 1 to #X : sets Master-Key[X[t]] := Q[t]
4) All players:
        4.1) For each x in X,  verifies that Card-Holder[x] := i. If not, then abort the protocol.
        4.2) For each x in X,  sets Card-Holder[x] := j.
```

- After:
    - The cad list X has been transferred from player i to player j.

3.7.24. Abrupt-Drop-out-Recovery (for VSM-VL and VSM-VL-VUM)

This protocol allows some players to recover from the drop-out of a player i.

Signature: Abrupt-Drop-Out-Recovery(public in i :Integer)

- Before:
    - Player i quits the game

```
1) Private tables are rebuilt:
        1.1) Every active player excludes player i from the game.
        1.2) All private table records that refer to player i are discarded.
        1.3) Players are renamed so player numbers are continuous and leave no gaps.
        1.4) All private table records are changed according to the new player numbers.
2) If not executed before, players execute Create-Recovery-Sets.
3) Each player j:
        3.1) Set Mask-Key :=Recovery-Set-Key
        3.2) Set Mask-Representative :=  Recovery-Mask-Representative
        3.3) Set X = []
        3.4) For each mapping ( z → j ) in Card-Holder do:
                3.4.1) Let s be the index such as $E_{Mask-Key}(D_{master-Key[z]}(z))$=Recovery-Set[j][s]
                3.4.2) Broadcast  (z, s)
                3.4.3) All players compute q := Recovery-Set[j][s]
                3.4.4) Players execute LVP( { z } , {q } , j )
                3.4.5) All players append q to X.
        3.5) Players Execute Verified-Re-ShuffleMasking-Round(X,Y, j)
        3.6) All players:
                3.6.1) Set Hand-Cards[j] := Y
                3.6.2) Check that #Y = number of cards mapped to player j in Card-Holder.
4) All players:
        4.1) Compute Z := Union ( For each player j, Hand-Cards[j] )
        4.2) Check that #Z = number of cards dealt minus the number of cards that were in the hand of the
        quitting player.
5) Let T  be the set of cards that are open and shown in the table.
6) Let R = Open-Deck – T
7) Players execute Verified-ShuffleRemasking-Round(R,Q,-1)
This will execute a complete shuffle-masking round, validating that the previous masking keys are used
again.
8) All players:
        8.1) Set Main-Deck := Q – Z
```

3.7.25. Create-Recovery-Sets

Every player create a set of masked cards from the open cards.

Signature: Create-Recovery-Sets

```
1) Each player i, in increasing order:
    1.1) Set F :=RandomPermutation(#Open-Deck).
    1.2) Set Recovery-Set-Key :=RandomKey(). (creates a new masking key for future use)
    1.3) Constructs a card list $Y_i$ := ShuffleMaskCards( Open-Deck, Recovery-Set-Key, F )
    1.4) Broadcast $\underline{Y_i}$
    1.5) Creates a representative R for the new masking key ,R.p :=RandomCardValue()
    1.6) R.c :=MaskCard(R.p, Recovery-Set-Key);
    1.7) Broadcasts $\underline{R}$
    1.8) Players execute RSMVP (  Recovery-Set-Key, F, Open-Deck , $\underline{Y_i}$ , i , [R.p] , [R.c] )
    1.9) Every player j :
        1.9.1) Set Recovery-Set[i] := $\underline{Y_i}$
        1.9.2) Set Recovery-Mask-Representative[i] :=$\underline{R}$
```

3.7.26. Put-Card-On-Table

This protocol allows a player to put a card on the table. The card put can be face up (if the card was opened before) or face down (if the card was not opened)

Signature: Put-Card-On-Table(private in Card x, private in i :Integer)

- Before:
  - Player i wants to put the card x on the table

```
1) Player i broadcasts $\underline{x}$ and $\underline{i}$
2) Every player:
    2.1) Checks that Card-Holder[$\underline{x}$] = i. If not, abort.
    2.2) Sets  Card-Holder[$\underline{x}$] = 0. (0 is a special value meaning “on the table”)
```

- After:
  - Player i no longer has the card x in his hand.

3.7.27. Verified-ShuffleRemasking-Round

This protocol implements a verified re-shuffle-masking round, skipping a certain player, who has already masked the cards. The masking key is checked to be the one previously used. This protocol is a sub-protocol of Return-Cards-To-Deck**.** If p is not a valid player number, then no player is skipped.

Signature: Verified-ShuffleRemasking-Round**(**
    public in $\underline{D}$: O-Card-List,
    public out $\underline{Z}$: M-Card-List,
    public in p :Integer)

- Before:
  - D is a list of O-Cards.

```
1) Each player i, such as i<>p, in increasing order:
    1.1) If i=0 then X =$\underline{D}$ else $\underline{X_i}$ =$\underline{Y_{i-1}}$
    1.2) Set F :=RandomPermutation($\underline{X_i}$).
    1.3) Constructs a card list $Y_i$ := ShuffleMaskCards( $\underline{X_i}$, Mask-Key , F )
    1.4) Broadcast $\underline{Y_i}$.
    1.5) Players execute RSMVP (  Mask-Key , F , $\underline{X_i}$, $\underline{Y_i}$, i , Mask-Representative[i].p, Mask-
    Representative[i].c)
2) $\underline{Z}$  = $\underline{Y_p}$ , where p is the last player in the round (Z is the round output).
```

- After:
  - Z a list of M-Cards.
  - Z is public.
  - Z is a permutation of the cards in D, after masking.
  - The permutation cannot be computed by any proper subset of players.

3.7.28. End-Of-Game (only for VSM-L-OL)

This protocol ensures the correctness of the second locking round

Signature: End-Of-Game

```
1) Each player i:
    1.1) For each card c such as (c,i) is in Card-Holder table.
        1.1.1) Set card_index :=Card-Trace[ 2, c]
        1.1.2) Let k :=Lock-Key[2,card_index]
        1.1.3) Broadcast c and k
    1.2) Each other player j:
        1.2.1) Check that (c,i) is in Card-Holder table. If not, then player i is cheating.
        1.2.2) Checks that (2,c) is in Card-Trace. If not, then the Card-Trace table is corrupt.
        1.2.3) Set card_index :=Card-Trace[ 2, c].
        1.2.4) Let R := Lock-Representative[2, i, card_index]
        1.2.5) Check that LockCard(R.p, k) = R.c. If not, then player i is cheating.
```

4. Sample VSM-VL Run for Texas Hold'em

Suppose two players want to play Texas Hold'em. This is a log of all protocols executed if there is no attempt to cheat and both players arrive to the showdown. Player 1 plays the game and also act as a dealer.

1.	Create-Deck	
2.	Shuffle-Deck	
3.	Prepare-Cards-To-Deal (8)	(prepared cards for fast deal)
4.	[Pre-flop betting round]	
5.	Single-Card-Deal(1)	(1st card to player 1)
6.	Single-Card-Deal(1)	(2nd card to player 1)
7.	Single-Card-Deal(2)	(1st card to player 2)
8.	Single-Card-Deal(2)	(2nd card to player 2)
9.	Single-Card-Deal(1)	(flop 1 card given to player 1 (the dealer))
10.	Single-Card-Deal(1)	(flop 2 card given to player 1)
11.	Single-Card-Deal(1)	(flop 3 card given to player 1)
12.	Show-Cards(1, { the 3 flop cards })	(flop cards are shown on table)
13.	[Betting round]	
14.	Single-Card-Deal(1)	(the turn card given to player 1)
15.	Show-Cards(1, { the turn card })	(turn card is shown on table)
16.	[Betting round]	
17.	Single-Card-Deal(1)	(the river card given to player 1)
18.	Show-Cards(1, { the river card })	(turn river is shown on table)
19.	Betting round	
20.	[Showdown]	
21.	Show-Cards(1, { all cards player 1 holds })	
22.	Show-Cards(2, { all cards player 2 holds })	

5. Pre-defined UniVPIs

This section describes 3 protocols that implement a UniVP. Protocols are secure in the MPF security model. The **I**-UniVPI protocol provides a Perfect Zero-Knowledge proof (PZNP). The rest provide Computational-Zero-Knowledge arguments of encryption and permutation of a computationally unique set of cards. The last protocol CO-VP is not strictly an UniVP, but can be used in cases where all players execute a locking or masking round and want to verify all others players operations. Table 6 summarizes external functions, for reference.

Function Name	Description
H	H is a function that models a random oracle To implement the protocol, a secure one-way cryptographic hash function is used instead.
EncryptCards(X:Card-List, L :Key-List)	Returns a card list for which each element is an encryption of the element in X and the key in M with the same index. If L has only one element, then the same keys is used for all elements in X.
#X	Returns the number of elements in the list X.
X[i]	Returns the i-th element of the list X.
X + Y	Returns the concatenation of the list X with the list Y.
RandomPermutation(n :Integer)	Returns a random or pseudo-random permutation of the integers from 1 to n.
Random(A :Set)	Returns a random or pseudo-random integer in the set A.
RandomKey()	Returns a random bitstring suitable as a key for the underlying CGC.
Com(X)	A non-interactive commitment to X
Check(X,R,C)	A Check that the commitment c to X is valid, using the additionally revealed information R.
Permute(C,P)	Permutes the card-list C using the permutation P. We extend the function so that if P is a special symbol SORT, then C is lexicographically sorted.

Table 6: External Functions

External Constants

- **s**: the security threshold for interactive proofs (cheating probability).
- $s_{ns:}$ the security threshold for non-interactive proofs (cheating probability).

5.1. Protocol FI-UniVP

With reference to Fig. 15, the FI-UniVP protocol is illustrated, corresponding to a Challenge-Response protocol which uses the commutativity property of the CGC. The verifier challenges the prover to re-do the encryption he is trying to prove on a different card-list. The challenge consist of a re-encryption and a permutation of X using a single key. The basic scheme is the following:

- P boxes are permutations.
- E boxes are encryptions.
- H boxes are hash functions.
- A is a card-list append operation.
- C is a message commitment.
- Depending on the arguments of the protocol, the U , L and P permutations are used or not.
- Dotted lines represent transmitted information.
- Time flows from top to bottom.
- If a check fails, the protocol aborts and no additional information is sent.
- Check 1: Checks that the challenge valid before opening the commitment on the result of the encryption. This stage prevents CPA attacks.
- Check 2: Checks that the committed response sent by the prover is correct. This check prevents the prover from changing the response after he knows the challenge.
- Check 3: The verifier checks that the response to the challenge is correct.

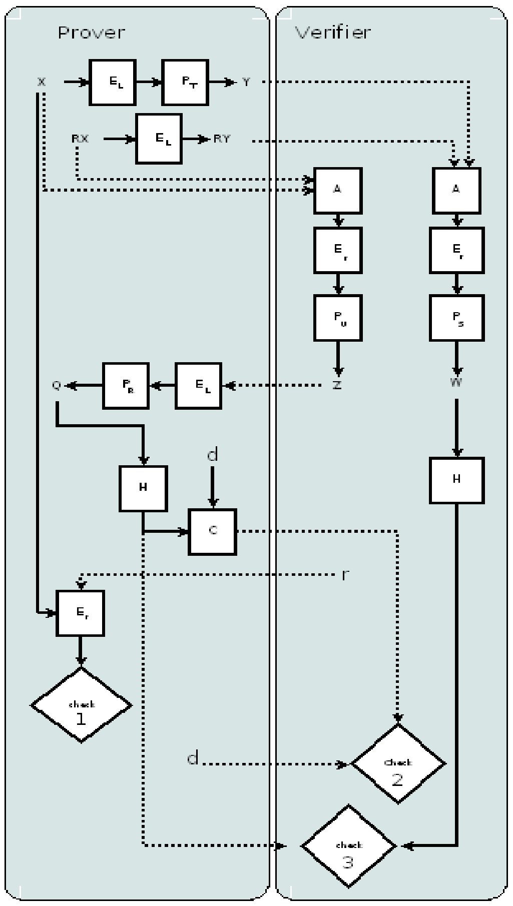

Figure 15: Protocol FI-UniVP

Commitments can be implemented in a number of ways [HIRLL99] [N91] [CS04]. We will use a simple scheme based on a cryptographic one-way hash function. The function must not only be one-way but must hide any information regarding the pre-image of a message digest.

Because we only use the commutativity property, this protocol cannot withstand malleability of the CGC. For a interactive protocol that withstand CGC malleability, see the protocol HMVP.

The permutations L, U and S vary depending on the protocol arguments, the three possibilities are a sort, the repetition of the permutation T, or the identity, as shown in Table 7.

	Permuted	Not Permuted
Samekey, no representatives	SMVP (*)	*Not required*

	U = RandomPermutation
	S = Sort
	R = Sort	

	U = RandomPermutation
	S = U
	R = T	

Samekey, with representatives	SMVP and RSMVP	UVP, UMVP and RLVP

	U = RandomPermutation
	S = Sort
	R = Sort
		U = RandomPermutation
		S = Sort
		R = Sort
		T = Identity

Not samekey	SLVP (*)	LVP

	U = Identity
	S = Sort
	R = Sort
		U = Identity
		S = Identity
		R = Identity
		T = Identity

	U = Identity
	S = Identity
	R = T	

(*) For this cases there are two possibilities.		

Table 7: Operations in U, S, R and T boxes

Proof Watermarking

If parallel runs of these protocols will be used during a game, then an additional security measure must be applied. The identity of the prover must be embedded in the proof in a process similar to a cryptographic watermark. This prevents the proof to be forwarded.

Verifier Nonce Watermarking

An alternate way to avoid parallel runs is that random keys chosen by the verifier are watermarked. The prover must verify the identity embedded in the key before giving away the proof.

Here is the protocol to obtain a random watermarked key r:

1. Choose a random value v.
2. Construct the message m = H(<verifier identity> | v).
3. Choose a valid key r which includes in an unambiguous representation of m. If necessary, pad with extra random bytes.

If decryption keys are not unique, then the step 3 of this method can open a security hole to attacks. A better way would be to supply m as the seed for a CSPRNG, and use the generator to produce the random key r.

<u>Protocol FI-UniVP</u>

Signature: FI-UniVP (private in L :Key-List, private in T :Permutation, public in <u>X</u> :Card-List,
public in <u>Y</u> :Card-List, public in p :Player, public in Permuted :Boolean,
public in SameKey :Boolean, public in <u>RX</u> :Card-List , public in <u>RY</u> : Card-List)

```
1) For each player v in increasing order, such as v <> p do
        1.1) Execute FI-UniVP-TP (M, X +RX, Y + RY , p,  v, Permuted, SameKey )
```

<u>Protocol FI-UniVP-TP</u>

```
Signature: FI-UniVP-TP ( private in L :Key-List, private in T :Permutation,          public  in X :Card-List,
        public  in Y :Card-List, public  in p :Player, public  in v :Player,
        public  in Permuted :Boolean,    public  in SameKey  :Boolean)

1) If (SameKey) then
        1.1) Set  f := 1
        1.2) Repeat until (f< s)
                1.2.1) Set f :=f*(1 / #X)
                1.2.2) Execute FI-UniVP-Core (L, X, Y, p, v, Permuted, SameKey)
2) else
        2.1) Execute FI-UniVP-Core (L, X, Y, p,  v, Permuted, SameKey )
```

<u>Protocol FI-UniVP-Core</u>

```
Signature: FI-UniVP-Core (
        private in L :Key-List,  private in T :Permutation,
        public  in X :Card-List, public  in Y :Card-List, public  in p :Player, public  in v :Player,
        public  in Permuted :Boolean, public  in SameKey  :Boolean)

        ■   L = Length(X) = Length(Y)

1) Player v:
        1.1) r  := RandomKey()
        1.2) Computes Z'  :=EncryptCards(X,r)
        1.3) If SameKey then
                1.3.1) Set U :=RandomPermutation(  #X )
        1.4) else
                1.4.1) Set U :=Identity
        1.5) If Permuted then
                1.5.1) S := SORT
        1.6) else
                1.6.1) S :=U
        1.7) Set Z := Permute(Z', U)
        1.8) Broadcast Z.
        1.9) Computes W' := EncryptCards(Y ,r)
        1.10) Set W := Permute(W',S)
        1.11) Computes $h_w$ := H(<prover identity> | W )
2) Player p:
        2.1) Computes Q'  := EncryptCards(Z, L).
        2.2) If (Permuted) then
                2.2.1) Set R :=SORT
        2.3) else
                2.3.1) Set R := Identity
        2.4) Set  Q  := Permute(Q' , R)
        2.5) Computes $h_q$ := H(<prover identity> | Q )
        2.6) Chooses a random fixed length string d.
        2.7) Computes $h_s$ := H( $h_q$ | d)
        2.8) Broadcasts $h_s$ (a commitment)
3) Player v:
        3.1) Broadcasts r.
4) Player p:
        4.1) If r is watermarked, verifies the identity of the sender. If a mismatch is found, aborts.
        4.2) Let G := EncryptCards(X,r).
        4.3) If not SameKey then
        4.3.1) Checks that Z = G.
    4.4) else
        4.4.1) Checks that Z is a permutation of G
        4.4.2) If not, then player v is cheating, trying to do a chosen plaintext attack.
    4.5) Broadcasts $h_q$ and d
5) Player v:
    5.1) Verifies that $h_w = h_q$ and that $h_s = H(h_q | d)$.
    If they are not equal, then player p is cheating.
```

5.2. Protocol FIG-UniVP

This protocol is similar to FI-UniVP but uses the CGC encryptor to make the verifier commitments. To to it, we'll use the group property of the CGC and not only commutativity.

Protocol FIG-UniVP

```
Signature: FIG-UniVP ( private in L :Key-List, private in T :Permutation, public in X :Card-List,
    public in Y :Card-List, public in p :Player, public in Permuted :Boolean,
    public in SameKey :Boolean, public in RX :Card-List , public in RY : Card-List)

1) For each player v in increasing order, such as v <> p do
    1.1) Execute FIG-UniVP-TP (M, X +RX, Y + RY , p, v, Permuted, SameKey )
```

Protocol FIG-UniVP-TP

```
Signature: TFI-UniVP-TP ( private in L :Key-List, private in T :Permutation, public in X :Card-List,
    public in Y :Card-List, public in p :Player, public in v :Player,
    public in Permuted :Boolean, public in SameKey :Boolean)

1) If (SameKey) then
    1.1) Set f := 1
    1.2) Repeat until (f< s)
        1.2.1) Set f :=f*(1 / #X)
        1.2.2) Execute FIG-UniVP-Core (L, X, Y, p, v, Permuted, SameKey)
2) else
    2.1) Execute FIG-UniVP-Core (L, X, Y, p, v, Permuted, SameKey )
```

Protocol FI-UniVP-Core

```
Signature: FI-UniVP-Core (
    private in L :Key-List, private in T :Permutation,
    public in X :Card-List, public in Y :Card-List, public in p :Player, public in v :Player,
    public in Permuted :Boolean, public in SameKey :Boolean)

    ■ L = Length(X) = Length(Y)

1) Player v:
    1.1) r := RandomKey()
    1.2) Computes Z' :=EncryptCards(X,r)
    1.3) If SameKey then
        1.3.1) Set U :=RandomPermutation( #X )
    1.4) else
        1.4.1) Set U :=Identity
    1.5) If Permuted then
        1.5.1) S := SORT
    1.6) else
```

```
        1.6.1) S :=U
    1.7) Set Z := Permute(Z', U)
    1.8) Broadcast Z.
2) Player p:
    2.1) Chooses a random key g. Let g :=RandomKey()
    2.2) Computes L' as, for each 1 <= i <= #L, L'[i] := L[i] * g
    2.3) Computes Q' := EncryptCards(Z, L').
    2.4) If (Permuted) then
        2.4.1) Set R :=SORT
    2.5) else
        2.5.1) Set R := Identity
    2.6) Set Q := Permute(Q' , R)
    2.7) Computes $h_s$ := H(<prover identity> | Q )
    2.8) Broadcasts $h_s$ (a commitment)
3) Player v:
    3.1) Broadcasts r.
4) Player p:
    4.1) If r is watermarked, verifies the identity of the sender. If a mismatch is found, aborts.
    4.2) Let G := EncryptCards(X,r).
    4.3) If (not SameKey) then
        4.3.1) Checks that Z = G.
    4.4) else
        4.4.1) Checks that Z is a permutation of G. If not, then player v is cheating, trying to do a
        chosen plaintext attack.
    4.5) Broadcasts g
5)Player v:
    5.1) Computes Y' as , for each 1 <= i <= #Y, Y'[i] := Y[i] * g
    5.2) Computes W' := EncryptCards(Y' ,r)
    5.3) Set W := Permute(W',S)
    5.4) Computes $h_w$ := H(<prover identity> | W )
    5.5) Verifies that $h_w$ = $h_s$. If they are not equal, then player p is cheating.
```

5.3. Protocol I-UniVPl

This is a multi-round cut-and-choose protocol. In each round verifier does the following operations: $X \rightarrow E_L \rightarrow P_T \rightarrow Y \rightarrow E_r \rightarrow P_U \rightarrow W$. E is encryption (with single or multiple keys), P is and optional permutation. R is a random key. W is transmitted to the verifier. Afterwards the verifier chooses to be sent the keys that encrypt X into W or the keys that encrypt Y into W (but not both).

Protocol I-UniVP

```
Signature: FI-UniVP ( private in L :Key-List, private in T :Permutation, public in X :Card-List,
    public in Y :Card-List, public in p :Player, public in Permuted :Boolean,
    public in SameKey :Boolean, public in RX :Card-List , public in RY : Card-List)

1) For each player v in increasing order, such as v <> p do
    1.1) Execute TI-UniVP (M, X + RX, Y + RY , p, v, Permuted, SameKey )
```

Protocol TI-UniVP

```
Signature: TI-UniVP ( private in L :Key-List, private in T :Permutation, public in X :Card-List,
    public in Y :Card-List, public in p :Player, public in v :Player,
    public in Permuted :Boolean, public in SameKey :Boolean)

1) Set f := 1
2) Repeat until (f<s)
    2.1) Set f :=f*(1 / 2)
    2.2) Execute CI-UniVP (L, T, X, Y, p, v, Permuted, SameKey)
```

Protocol CI-UniVP

```
Signature: CI-UniVP (   private in L :Key-List,  private in T :Permutation,
        public  in X :Card-List, public  in Y :Card-List, public  in p :Player, public  in v :Player,
        public  in Permuted :Boolean, public  in SameKey  :Boolean)

1) Player p:
        1.1) Set r :=RandomKey()
        1.2) Set W' :=EncryptCards(Y,r)
        1.3) If Permuted then
                1.3.1) Set U :=RandomPermutation(#X)
        1.4) else
                1.4.1) Set U :=Identity
        1.5) Set W :=Permute(W' , U)
        1.6) Broadcasts W
2) Player v:
        2.1) Set c :=Random( { 0,1 } ) (A random integer value similar to a coin flip)
        2.2) Broadcasts c
3) Player p:
        3.1) If (c = 0) then
                3.1.1) If SameKey then
                        3.1.1.1) Set g :=r*L[1]
                        3.1.1.2) Broadcasts g
                3.1.2) else
                        3.1.2.1) Set G :=r*L (scalar product of vector L)
                        3.1.2.2) Broadcasts G
        3.2) else
                3.2.1) Set g :=r
                3.2.2) Broadcasts g
4) Player v:
        4.1) If (c=0) then
                4.1.1) If SameKey then
                        4.1.1.1) Computes Q := EncryptCards(X, g)
                4.1.2) else
                        4.1.2.1) Computes Q := EncryptCards(X, G)
        4.2) else
                4.2.1) Computes Q := EncryptCards(Y, g)
        4.3) If Permuted then
                4.3.1) Check that Q is a permutation of W. If not, then player v is cheating,
        4.4) else
                4.4.1) Check that Q = W.  If not, then player v is cheating,
```

5.4. Protocol NI-UniVPI

This protocol is obtained by modifying the I-UniVP protocol using the Fiat-Shamir transformation.

```
1) Player p:
        1.1) Let F be a binary stream.
        1.2) Save the prover identity into the stream F.
        1.3) Repeat $s_{ni}$ times
                1.3.1) Execute Step 1 of protocol I-UniVP locally (witting the broadcast outputs into the
                stream F)
        1.4) Compute $h_f$ := H (F)
        1.5) For i from 1 to $s_{ni}$ do
                1.5.1) Let c be the i-th bit of $h_f$ (the message digest $h_f$ must be long enough)
                1.5.2) Execute Step 3 of protocol I-UniVP locally (writing the broadcast outputs into a stream
```

T)

1.6) Construct the message (F , T) (the non-interactive proof).

1.7) Broadcasts (F , T)

2) All other players:

2.1) Compute u :=H (F)

2.2) Read the prover identity from the stream F and check it. If not equal, abort.

2.3) For i from 1 to s_{ni} do

2.3.1) Let c be the i-th bit of u.

2.3.2) Read the values G or g (depending on the protocol arguments) from the stream F. If the stored value is of an invalid type (e.g: reads G expecting g) abort.

2.3.3) Execute Step 4 of protocol I-UniVP.

5.5. Protocol CO-VP

This is a cut-and-choose protocol to verify a round that can be a either a locking round or shuffle-masking round. All the players act as provers and verifiers.

1) Players execute a unverified round (Locking or ShuffleMasking) that transforms the card-list X into Y where K_i is the list of key used by player i. Every player has taken part and want to verify the round. This can be either a normal round or a CO-SMVP round.

2) Repeat a number of times until a security threshold has been achieved:

2.1) Each player i chooses a new single key or list of keys R_i (depending on the kind of round)

2.2) Players re-execute the (still unverified) round with card-list Y as input and creates Z as output. This can be either a normal round or a CO-SMVP round.

2.3) Each player commits to a value 0 or 1, similar to a coin-flip.

2.4) All players open the commitments and calculate a value c as the exclusive-or of all committed values.

2.5) If (c=0) then every player i:

2.5.1) Publishes R_i

2.5.2) Check that the operations done by each other player in the last round are correct.

2.6) If (c=1) then every player i:

2.6.1) Calculates S_i as the item-by-item product of K_i and R_i.

2.6.2) Commits to S_i

2.6.3) Waits until all commitments have been done.

2.6.4) Opens the commitment and publishes S_i

2.6.5) After all commitments have been opened, computes S as the product of all S_i values.

2.6.6) Checks that X can be transformed into Z by encrypting with S. The transformation may require a permutation if the round to verify is a Shuffle-Masking round.

6. Security of MPF

The security of protocols that are based on algebraic properties of its building blocks are difficult to prove formally [VSP06]. We won´t attempt such a formalization in this thesis. Nevertheless we´ll analyze the protocol against the most common attacks:

1. Communication channel: Eavesdropping, Impersonation, Message Injection, Man-In-The-Middle, etc.
2. Algebraic properties of the CGC
3. Card Protocols Design
4. Verification Protocols Design

Attacks on the Communication channel

MPF, like other MP protocols, does not provide authenticity and privacy for the messages exchanged. MPF must be run over on secure communication channels, such as SSL. Secret keys used for the secrecy of the communication channel or for authentication should never be reused in MPF, as this can decrease the overall security.

Attacks on the algebraic properties of the CGC

In this thesis UniVPs are not proven secure. Nevertheless, there exists proofs for soundness, completeness and and zero knowledge.

In VSM-VL, VSM-VPUM and VSM-VL-VUM protocols, attacks on the algebraic properties of the CGC are impossible due to the fact that all private computations are verified by executions to UniVP. Chosen plaintext attacks are also avoided by UniVPs.

In VSM-L-OL, the Lock1 round is not explicitly verified. Nevertheless, Lock1 is a round in which every player encrypts each card with a distinct random key, so no information can leak. MPF security assume CPA for VSM-L-OL. so the Lock1 key cannot be recovered by an active attacker.

Attacks on the card protocols design

The most common attack on a protocol design is the replay attack [PS04]. Replay attacks often require interleaved runs of the protocol. MPF prevents replay attacks in two ways:

a. Private keys for each game are randomly chosen out of each player CS-PRNG.
b. All steps in the MPF, with the exception of UniVPs, are defined to be executed sequentially.

UniVPs can be securely parallelizable, because they withstand dishonest verifiers. The result of all operations performed by the prover are passed through a one-way hash function and prover never performs decryptions. Also the prover does not provide any additional computationally distinguishable information to the verifier (computational zero knowledge property), so parallel runs of the UniVPs can never be used to obtain any secret information. Also, parallel runs cannot be used to impersonate a prover and provide a valid proof of knowledge for a unknown fact, as in a man-in-the-middle attack. This is prevented by design because free card values are attached to a certain player in the Card-Holder table. Also we use of two additional protective techniques: prover watermarking and verifier Nonce watermarking. The former binds a proof to the prover identity, so it cannot be reused. The later binds all nonces sent by the verifier to its identity, and nonces source is verified by the prover before any information is given.

6.1. Example Attacks

During the design stage of MPF we identified a number of possible attacks. MPF has countermeasures for these attacks. Nevertheless, the leakage of even some bits information through an external channel or in conjunction with additional attacks can make these attacks feasible. The following sections contain some possible attacks against MPF in general and against MPF implemented on a homomorphic cipher.

6.1.1. Tracking attack

The masking key should never be used twice to encrypt a list of cards: there should a single masking round for a masking key. Suppose is Alice's turn to shuffle-mask a card list X and output a card list Y. Suppose that Mallory is the previous player on the masking round order. If Mallory knows a pair (x,y) such as $E_m(x)=y$ such as x belongs to X, and m is Alice's private masking key, then he can track the position of y in Y, revealing some information regarding the secret permutation.

6.1.2. Sandwich tracking attack against CO-VP

If the CGC is homomorphic, Mallory can improve the tracking attack during a CO-VP protocol. He needs to collude with the player Oscar following Alice order in the round. Suppose Mallory knows a list of pairs (a[i],b[i]) E_m(a[i])=b[i] (1<=i<=n/2) (although he may know what actual card values map to). Suppose Mallory's input card-list is W. For simplicity, let's assume that Mallory's private key and permutation are the identity. Let P be Alice's permutation function. When is Mallory's turn, he chooses a set of cards w[i+k] from W (1<=i<=k) that he knows the card values that they map to, and he wants to track them. He outputs a card list X such as:

X= < a[1]*w[k+1], .., a[2]*w[k+2] , a[k]*w[2*k] , w[k+1], ..., w[n] >

Then, after Alice encrypts and permutes the list X, she outputs the set:

Y= { E_m(a[1]*w[k+1]), .., E_m(a[2]*w[k+2]) ,E_m(a[k]*w[2*k]) , E_m(w[k+1]),.., E_m(w[n]) } =

Y= { E_m(a[1])*E_m(w[k+1]),..,E_m(a[2])*E_m(w[k+2]) ,E_m(a[k])*E_m(w[2*k]),E_m(w[k+1]),..,E_m(w[n]) } =

Y= { b[1]*E_m(w[k+1]),..., b[2]*E_m(w[k+1]), b[k]*E_m(w[2*k]) , E_m(w[k+1]), ... E_m(w[n]) } =

Now Oscar can detect Alice's permutation function for the values w[k+1],..,w[2*k]. For example, to recover P[k+t] (output index of the element w[k+t]) Oscar searches for two indexes i,j such as y[i] * $y[j]^{-1}$ = b[t]. Then P[k+t] = j. He can reconstruct a valid Y' to transfer to the following player in the round by replacing the terms b[t]*E_m(w[k+t]) with the known b[t].

7. Detailed Comparison of Mental Poker Protocols

Here is a review of the most complete MP protocols [CR05] and the properties satisfied. [SRA81] is added for comparison. Table 8, compares properties of various protocols, including those of MPF.

Key for Table 8:

R1. Uniqueness of card
R2. Uniform random distribution of cards
R3. Cheating detection with a very high probability
R4. Complete confidentiality of cards
R5. Minimal effect of coalitions
R6. Complete confidentiality of strategy
R7. Absence of trusted third party
R8. Polite Drop-out tolerance
R9. Abrupt Drop-out tolerance
R10. Real-world comparable performance
R11. Variable number of players
R12. Card transfers
R13. Protection against suicide cheaters

Protocol	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14
VSM-VL-VUM	√	√	√	√	√	√	√	√	√	√	√	√	√	√
VSM-VL	√	√	√	√	√	√	√	√	√	√	√	√	√	√
VSM-VPUM	√	√	√	√	√	√	√	√	√	√	√		√	
VSM-L-OL	√	√	√	√	√	√	√	√		√	√	√		
KKOT90	√	√	√	√	√	√	√	√			√			
BS03	√	√	√	√	√	√	√	√			√	√	√	√
CSD05	√	√	√	√	√	√	√	√	√		√		√	
SRA81	√	√			√		√	√		√				

Table 8: Comparison of Mental Poker Protocols

8. PHMP (MPF VSM-VL with Pohlig-Hellman as CGC)

In this section we present PHMP, an implementation of PHMP that uses the Pohlig-Hellman symmetric cipher [PH78] as the underlying CGC for the MPF with the VSM-VL base protocol.
As stated before, to create a MPF protocol we must specify a CGC and ad-hoc protocols, if desired.
We will create:

- The CGC function (E)
- A protocol to create the cipher parameters from a stream of random bytes
- A protocol to create the a cipher key from a stream of random bytes
- A protocol to create a single card (Create-Deck by Locking).
- A protocol to create random card values from a stream of random bytes (Create-Deck by CO-PRNGP)
- Ad-hoc protocols.

Note that PH is a malleable cipher, and because of this the protocol FI-UniVP becomes completely insecure, because plaintext products commute with the encryption required to build the challenge card-list. We'll describe a modified protocol to fix this problems, keeping the number of modular exponentiations low.
To be used as CGC, the external assumptions described in section 2 must hold. If we were to encrypt a single plaintext/ciphertext pair, Pohlig-Hellman security against COA, KPA and CPA is assured by the difficulty of the Discrete Logarithm Problem. But MPF reuses keys and any statistical information regarding the distributions of ciphertexts gained after a shuffle can be considered a successful attack, so PH security in MPF is guaranteed by the difficulty of the Decisional Diffie-Helman in the polynomial samples setting, which is shown in [BDH02] to be equivalent to DDH. The **Decisional Diffie–Hellman (DDH) assumption** is a computational hardness assumption. Let G be a multiplicative cyclic group of order *q*, with a generator *g*. The DDH assumption states that, given g^a and g^b for randomly-chosen a,b $\in Z_p$, the value g^{ab} is computationally indistinguishable from a random element in *G*. Note that DDH is an assumption of many common cryptographic schemes. For example, ElGamal cryptosystem has semantic security only if DDH holds.

8.1. Definition of E

Encryption:	$E_k(m) = m^k \pmod p$
Commutation:	$E_k(E_q(m)) = m^{qk} \pmod p = m^{kq} \pmod p = E_q(E_k(m))$
Composition:	$E_k(E_q(m)) = m^{qk} \pmod p = E_{k*q}(m)$
Inversion:	k^{-1} is such that $k * k^{-1} = 1 \pmod{p-1}$

Where $1 < m < p$, and $m \in Z$.

8.2. Cipher Parameters Creation

Pohlig-Hellman cipher requires a strong prime or pseudo-prime p. We'll use a strong prime in the format p = 2*q+1, where q is a big prime number. To create p from a PRNG we use the process described in the standard [FIPS186].

An integer *y* is called a **quadratic residue** or **QR** modulo p if it is congruent to a perfect square (mod *p*). Otherwise, y is called a **quadratic nonresidue** or **QNR**. Formally y is QR if there exists an integer *x* such that: $x^2 = y \pmod p$.

There are two possibilities for the plaintext space, either use a **Schnorr group**, the subgroup of quadratic residues (where DDH assumption has been studied more) or use the set of quadratic non-residues. If the later is used, then keys must be odd. If the QR subgroup is used, then each even key e < q has an equivalent odd key e'=q+e (mod p), and every even key d > q has an equivalent odd key d'=d-q (mod p). This equivalence allows any key k < p to be used. Will present both schemes.

8.3. Finding fixed generators

Because p=2*q+1, then p≡3 (mod 4) and then −1 is a nonresidue (mod *p*). This implies that the negative of a residue (mod *p*) is a nonresidue and the negative of a nonresidue is a residue. Also the only generator of the subgroup of order 2 is (p-1). For any p>5, then 4 is a quadratic residue because $4=2^2$ (mod p) and so 4 is generator of the QR subgroup. Then (p-4) is always a generator of (Z/pZ)*.

These generators can be used to create the deck with the procedure "Create-Deck by Locking".

8.4. Card Creation

To create the cards, we have two choices:

a) Execute repeatedly the procedure for finding random generators. The random number source stream is generated with the CO-PRNG protocol "Create-Deck by CO-PRNGP".

b) Generate a single fixed generator of the group and then create the rest of the cards by executing the protocol "Create-Deck by Locking".

8.5. Using quadratic residue cards

8.5.1. Finding a random generator of the QR subgroup

We describe a procedure that can be used to create a random generator g of order q of the QR subgroup.

To generate g:

```
Step 1. Set g := a random integer, where 1 < g < p - 1 and g differs from any value previously tried.

Step 2. If (g<=1) or (g=p-1) then Set g := g + 1 (p-1) and go to step 2.

Step 3. Set v := $g^q$ (mod p).

Step 4. If v <> 1 then Set g := g + 1 (p-1) and go to step 2.
```

Note that as p grows large the factor of generators g approximates ½.

8.5.2. Key Creation

To create a valid key k, find an integer value k in the range 1 < k < p such as gcd(k,q) = 1.

8.6. Using quadratic non-residue cards

8.6.1. Finding a generator of (Z/pZ)*

We describe a procedure that can be used to create a random generator g of order 2*q.

To generate g:

```
Step 1. Set g := a random integer, where 1 < g < p - 1 and g differs from any value previously tried.

Step 2. If (g<=1) or (g=p-1) then Set g := g + 1 (p-1) and go to step 2.

Step 3. Set v := $g^q$ (mod p).
```

Step 4. If v = 1 then Set g := g + 1 (p-1) and go to step 2.

8.6.2. Key Creation

To create a valid key k, find an integer value k in the range $1 < k < p$ such as $\gcd(k,2*q) = 1$. Note that all valid keys are odd.

8.7. Additional checks

We'll say that the k is an identity key if for any input card x, $y=E_k(x) = x$. It is possible, although unlikely, that an identity card key is created by composing non identity keys. If an identity key is obtained either privately or jointly, then the last protocol must be redone to create a new non-identity key. The same check can be applied for other sets of weak keys, like small keys or keys having too many zeros. Nevertheless, the probability of creating a weak key is negligible.

If a player violates the key creation protocol and chooses a key k=0 or a key k that divides q, then the following protocols will fail. Therefore, card-values equal to 1 or 0 should not be accepted by players. When using the QR subgroup, it's impossible that a QNR card will appear. Using QNR cards, the even keys and the key q will turn cards into quadratic residues. Each player should check that the input card values are all quadratic residues or all non-residues (depending on the group used). Also players can check that the input card-list have no duplicates. This is not strictly necessary, because the protocol design guarantees it. Nevertheless, these checks can prevent a failure in the protocol itself to expose players private information.

8.8. Ad-hoc verification protocols

Some UniVPs that come with MPF can withstand the malleability of PH cipher, so, in principle, there is no need to provide an ad-doc protocol. Nevertheless, there are more efficient alternatives to standard non-malleable UniVPs, so PHMP uses a new protocol called HMVP, and Chaum-Pedersen and Schnorr's Id Protocol when possible. HMVP is a hybrid protocol which uses both FI-UniVP and Chaum-Pedersen Protocol to achieve its goal.
Some of the alternatives for the UniVPs depending on the kind of verification are provided in Table 9.

	With permutation	Without permutation
Same key	For SMVP and RSMVP: Neff protocol adaption to PH [N04] Groth protocol adaption to PH [G05] FI-UniVP (for a single card only) NI-UniVP HMVP CO-VP	For UVP, UMVP, and RLVP: CP-UMVP (Chaum-Pedersen protocol) FI-UniVP (for a single card only) NI-UniVP
Different keys	For SLVP: FI-UniVP NI-UniVP	For LVP: S-LVP (one execution of the Schnorr's Id Protocol for each card) FI-UniVP NI-UniVP

Table 9: Alternatives for predefined UniVPs Depending on the Type of Verification

We've found that the protocol CO-VP can be disrupted in sandwich attacks. Suppose there is a masking round with players Mallory, Alice and Oscar in that exact order. Mallory can, instead of doing a normal shuffle-mask operation, output a card-list using a function of type T to the cards (see below for the definition).

In a function of type T, each card is encrypted as usual, but also multiplied by the encryption of the remaining cards. Oscar can recover a well-formed card list, where each output card is the encryption of a single input card, applying another function of type T. We haven't found a way to take advantage of this problem in the protocol, but we recommend using any other UniVP instead of CO-VP for homomorphic ciphers or use the protocol CO-SMVP for the steps 1 and 2.2 of of then CO-VP protocol.

Function T(X : Card-List; k,s :Key; P :Permutation) -> Y : Card-List , such that for each 1 <= i <= #X,
Y[i] =Product(1<=j<=#X: X[j])^k * (X[P[i]])^s

where P is a permutation of the set [1..#X], and v and k are integers (0<=s,k<p)

We show how Oscar can reconstruct a valid list of ciphertexts with another function of type T:

Suppose X is the card list that Mallory has to shuffle-Mask, and Y is the supposed corresponding output. Y is the input of Alice Shuffle-Mask and Z is Alice's output. Z is the input of Oscar supposed Shuffle-Mask and W is Oscar's output. The performed steps are:

1. Mallory performs Y := T(X,k,s,F_m)

2. Alice performs Z :=ShuffleMaskCards(Y,m,F_a), where m is the masking key and F_a is a random permutation.

3. Oscar performs W :=T(Z,k',s',F_o)

Mallory and Oscar can compute a set of values {k,s,k',s'} to recover a valid ciphertext list.

Let P = Product(1<=j<=#X: X[j]).

For each 1<=i<=#X: Y[i] = P^k * (X[F_m[i]])^s

For each 1<=i<=#X: Z[i] = Y[F_a[i]]^m = P^{km} * X[F_m[F_a[i]]]^{sm}

For each 1<=i<=#X: W[i] = $P^{nkmk'+smk'+kms'}$ * X[F_m[F_a[F_o[i]]]]$^{sm\ s'}$ = $P^{m(nkk'+sk'+ks')}$ * X[F_m[F_a[F_o[i]]]]$^{sm\ s'}$

Let r = nkk'+sk'+ks', and q = F_m[F_a[F_o[i]]], then

For each 1<=i<=#X: W[i] = P^{mr} * X[q]$^{sms'}$

To obtain valid W card values, the term that involves P must be canceled. This can be achieved by Mallory and Oscar by choosing the values k,s,k',s' so that r = 0 (mod p), which is easy.

8.8.1. HMVP

The Homomorphic ShuffleMasking Verification Protocol (HMVP) provides a computational zero knowledge argument for the mix and re-encryption of a set of plaintexts (shuffle-masking), preventing the use of the homomorphic property of the cipher.

Protocol HMVP

Signature: HMVP (private in m :Key, private in T :Permutation, public in X̲ :Card-List,
public in Y̲ :Card-List, public in p :Player,
public in R̲X̲ :Card-List , public in R̲Y̲ : Card-List)

1) The verifier:
 1.1) Chooses a random number s̲
 1.2) Sends s̲ to the prover
2) The prover:
 2.1) Computes R.p = CreateCard(H(s))
 2.2) Computes R.c = E_m(R.p) (R is a representative of m)
 2.3) Publishes R̲
3) The verifier checks that R.p and R.c are valid blocks (not marked)
4) Execute S-LVP ([m],[R̲.p],[R̲.c],p);
5) Execute FI-SMVP(m ,T, X,Y,p, RX + [R.p], RY + [R.c])

8.8.2. Schnorr's Id Protocol based LVP (S-LVP)

Here is a protocol for the verification of locking rounds based on Schnorr's Id protocol:

Let p=2*q+1 be the Poling-Hellman cipher parameters.

Protocol S-LVP

```
Signature: S-LVP (private in L :Key-List, public  in X :Card-List,
          public  in Y :Card-List, public  in p :Player)

1) For each player v in increasing order, such as v <> p do
          1.1) For each i ,  1 <= i <= #X do
                    1.1.1) Execute T-S-LVP (L[i], X[i], Y[i], p,  v )


Signature: T-S-LVP (public in s :Key,  public in  a :Card, public in b :Card,
          public in prv :Player, public in v :Player)

1) Player prv:
          1.1) Picks a random number r ( r <= q) and computes x  := $a^r$ (mod p)
          1.2) Broadcast r
          1.3) Player v:
                    1.3.1) Chooses a random value e ( e < p)
                    1.3.2) Broadcast e
1.4) Player prv:
          1.4.1) Computes y := r + s*e (mod p)
          1.4.2) Broadcasts y
1.5) Player v:
          1.5.1) Verifies that  x = $a^y * b^{-e}$ (mod p)
```

8.8.3. Chaum-Pedersen based UMVP (CP-UMVP)

We can use Chaum-Pedersen protocol to obtain an alternative UMVP. The protocol verifies that the encryptions have been done using the same key. The protocol requires 3 encryptions per card, similar to FI-UniVP. The number of transferred bytes is also equivalent. Here is the protocol:

Protocol CP-UMVP

```
Signature: CP-UMVP (private in m :Key, public  in X :Card-List,
          public  in Y :Card-List, public  in p :Player,
          public  in RX :Card-List , public  in RY : Card-List)

1) For each player v in increasing order, such as v <> p do
          1.1) Execute T-CP-UMVP (m, X +RX, Y + RY , p,  v )
```

Protocol T-CP-UMVP

Let p=2*q+1 be the Poling-Hellman cipher parameters.

```
Signature: T-CP-UMVP (public in m :Key,  public in  X :Card-List, public in Y :Card-List,
          public in prv :Player, public in v :Player)

1) Player prv:
          1.1) Picks a random number s ( s < q).
          1.2) For each 1 <= i <= #X, computes Q[i] := $X[i]^s$ (mod p)
          1.3) Broadcast Q
2) Player v:
          2.1) Chooses a random value c ( c < p)
```

```
        2.2) Broadcast c
3) Player prv:
        3.1) Computes r := s + c*m
        3.2) Broadcasts r
4) Player v:
        4.1) For each 1 <= i <=#X, verifies that  $X[i]^r = Q[i] * Y[i]^c$
```

8.8.4. CO-SMVP

This protocol can be used in step 1 and step 2.2 of the CO-VP protocol to implement a Shuffle Masking round that is resistant against sandwich attacks. The protocol is only required for homomorphic ciphers.

Before starting the protocol, all players agree on a neutral card value u, such as the set of all open cards plus u is CU. This can achieved using the protocol CO-PRNG.

CO-VP step 1:

CO-SMVP(X, Y , u , Mask-Key, true)

CO-VP step 2.2:

CO-SMVP(Y, Z , u , R_i ,false)

```
Signature: CO-SMVP(
        public in  X: Card-List,
        public out Y: Card-List,
        public  u :Card,
        multi-private out Round-Key :Key,
        public in saveRepresentatives :boolean )

1) Let N be the number of players
2) Set f := 1
3) Set $Z_0$ := X
4) Set i :=0
5) Repeat until (f< s)
        5.1) Set f :=f*(1 / #$Z_i$)
        5.2) Execute Augmented-ShuffleMasking-Round(L, $Z_i$, $W_i$,i)
        5.3) For each player p in increasing order :
                5.3.1) Reveal the value $s_{p,p}$ that satisfies $s_{p,p}$  = E(Temp-Key$_p$[i],u).
                5.3.2) Let R.p := u and let R.c := $s_{p,p}$
                5.3.3) if (saveRepresentatives) then
                        5.3.3.1)  All players set Mask-Representative[p] := R
                5.3.4) Execute LVP([ Temp-Key$_p$[i] ] , [ u ] ,[$s_{p,p}$], p)
                5.3.5) For each player k in increasing order (for k > p )
                        5.3.5.1) Reveal the value $s_{p,k}$ that satisfies  $s_{p,k}$ = E(Temp-Key$_k$[i], $s_{p,k-1}$)
                        5.3.5.2) Execute LVP([ Temp-Key$_k$[i] ] , [ $s_{p,k-1}$ ] ,[$s_{p,k}$], p)
        5.4) All players calculate $Z_{i+1}$ as $W_i$ excluding  the values  $s_{p,N}$  for every player p. Note that afterwards
        #$Z_{i+1}$ = #$Z_i$ = #X.
        5.5) Let i :=i +1
6) Let Y =  $Z_i$
7) Each player p:
        7.1) Sets Round-Key$_p$ :=Product(1 <= j <= i : Temp-Key$_p$[j] )
```

```
Signature: Augmented-ShuffleMasking-Round(
        public in  X: Card-List,
        public out Y: Card-List,
        public in r :Integer)
```

- Before:
  - i is a local variable

```
1) For each player i, in increasing order:
    1.1) If i=0 then $X_i = \underline{X}$ else $\underline{X}_i = \underline{Y}_{i-1}$
    1.2) Set F :=RandomPermutation(#$\underline{X}_i$).
    1.3) Set m :=RandomKey()..
    1.4) Constructs a card list $Y_i$ := ShuffleMaskCards( $\underline{X}_i$ + [ u ], m , F )
    1.5) Broadcast $\underline{Y}_i$.
    1.6) Sets Temp-Key$_i$[r] := m.
2) $\underline{Y}$ = $\underline{Y}_p$ , where p is the last player in the round (Z is the round output).
```

8.9. Numeric Example

This is an example of the PHMP protocol, using the base protocol VM-VL protocol. We assume there are 3 players and 4 cards in the deck. Cards are quadratic residues. Keys are even, although this is not necessarily for QR cards.

Cipher parameters:

$q = 509$

$p = 2q+1 = 1019$ (p is a safe prime)

The fixed generator for the QR subgroup is g = 3. Note that the size of p chosen is too small to provide any real security. Generally, p is at least 1024 binary digits long or around 350 decimal digits. For simplicity, we show the only action protocols, and we skip the verification sub-protocols.

First all players execute the protocol "Create-Deck (Locking)" (section 3.7.2), so the following has the structure of a locking round.

Player 1:

1. Construct X := [g, g, g, g] = [3, 3 , 3, 3] (a vector with four copies of g, one for each card)
2. Chooses a random or pseudo-random key-List K (each key is constructed using the protocol "Key Creation".
   Let K := [7, 123 , 441, 9]
3. Computes
   Y_1 :=LockCards(X,K)
   Y_1 := [$g^{k[1]}$ (mod p) , $g^{k[2]}$ (mod p) , $g^{k[3]}$ (mod p) , $g^{k[4]}$ (mod p)]
   Y_1 :=[3^7 (mod 1019) , 3^{123} (mod 1019) , 3^{441} (mod 1019) , 3^9 (mod 1019)]
   Y_1 := [149, 778, 256, 322]
4. Broadcasts Y_1

Player 2:

5. Chooses a random or pseudo-random key-List K (each key is constructed using the protocol "Key Creation" (section 7.3).
   Let K := [21, 99 , 73, 901]
6. Computes
   Y_2 :=LockCards(Y_1,K)
   Y_2:= [$Y_1[1]^{k[1]}$ (mod p) , $Y_1[2]^{k[2]}$ (mod p) , $Y_1[3]^{k[3]}$ (mod p) , $Y_1[4]^{k[4]}$ (mod p)]
   Y_2 := [958 ,42 ,626 ,345]
7. Broadcasts Y_2

Player 3:

8. Chooses a random or pseudo-random key-List K (each key is constructed using the protocol "Key Creation" (section 7.3).
   Let K := [701, 373, 13 , 629]
9. Now computes
   Y_3 :=LockCards(Y_1,K)
   Y_3:= [$Y_2[1]^{k[1]}$ (mod p) , $Y_2[2]^{k[2]}$ (mod p) , $Y_2[3]^{k[3]}$ (mod p) , $Y_2[4]^{k[4]}$ (mod p)]
   Y_3 := [1011 ,731 ,118 ,380]

10. Broadcasts Y_3

Let Open-Deck := Y_3 . Now we assign a meaning to each open-card value in the deck.

Open-Deck[1] = 1011	is the ace of diamonds.
Open-Deck[2] = 731	is the two of spades
Open-Deck[3] = 118	is the three of hearts
Open-Deck[4] = 380	is the four of clubs.

Because we will only shuffle four cards, no additional card is required. Now we shuffle the deck with the protocol “Shuffle-Deck” (section 3.7.3). Previous Y_i values are disposed.

Player 1:

1. Set F :=RandomPermutation(4)
   F := [2, 3 , 1 , 4]
2. Set m := RandomKey()
   m := 445
3. Compute
   Tmp := PermuteCards(Open-Deck ,F)
   Tmp := [731 ,118 , 1011 , 380]
4. Now Y_1 := MaskCards(Open-Deck, m , F)=
   [Tmp[1]^{445} (mod p), Tmp[2]^{445} (mod p), Tmp[2]^{445} (mod p), Tmp[2]^{445} (mod p)]
   Y_1 := [731^{445} (mod 1019), 118^{445} (mod 1019), 1011^{445} (mod 1019), 380^{445} (mod 1019)]
   Y_1 := [229 ,825 ,358 ,687]

Player 2:

5. Set F :=RandomPermutation(4)
   F := [3, 4 , 1 , 2]
6. Set m := RandomKey()
   m := 299
7. Compute
   Tmp := PermuteCards(Y_1,F)
   Tmp := [358, 687, 229, 825]
8. Now Y_2 := MaskCards(Y_1, m , F) =
   [Tmp[1]^{299} (mod p), Tmp[2]^{299} (mod p), Tmp[2]^{299} (mod p), Tmp[2]^{299} (mod p)]
   Y_2 := [668 ,685 ,395 ,42]

Player 3:

9. Set F :=RandomPermutation(4)
   F := [4, 3 , 2 , 1]
10. Set m := RandomKey()
    m := 101
11. Compute
    Tmp := PermuteCards(Y_2,F)
    Tmp := [42, 395, 685, 668]
12. Now Y_3 := MaskCards(Y_2, m , F) =
    [Tmp[1]^{101} (mod p), Tmp[2]^{101} (mod p), Tmp[2]^{101} (mod p), Tmp[2]^{101} (mod p)]
    Y_3 := [545 ,283 ,196 ,193]

Now, Main-Deck := Y_3 = [545 ,283 ,196 ,193].

We now execute the protocol Prepare-Cards-To-Deal (for VM-VL) (section 3.7.5). Because we prepare all the cards in the deck, we won't use the Prepare-Card table, but just modify the Main-Deck as we prepare the cards. We dispose previous Y_i values and K values, and execute a locking round.

Player 1:

1. Chooses a random or pseudo-random key-List K (each key is constructed using the protocol “Key Creation”.

Let K := [99, 183, 875, 571]

2. Computes
   Y_1 :=LockCards(X,K)
   Y_1 :=[Main-Deck $^{k[1]}$ (mod p) , Main-Deck $^{k[2]}$ (mod p) , Main-Deck $^{k[3]}$ (mod p) , Main-Deck $^{k[4]}$ (mod p)]
   Y_1 :=[545^{99} (mod 1019) , 283^{183} (mod 1019) , 196^{875} (mod 1019) , 193^{571} (mod 1019)]
   Y_1 := [768 ,45 ,239 ,917]
3. Broadcasts Y_1

Player 2:

4. Chooses a random or pseudo-random key-List K (each key is constructed using the protocol "Key Creation".
   Let K := [601, 47, 867, 29]
5. Computes
   Y_2 :=LockCards(Y_1,K)
   Y_2:= [$Y_1[1]^{k[1]}$ (mod p) , $Y_1[2]^{k[2]}$ (mod p) , $Y_1[3]^{k[3]}$ (mod p) , $Y_1[4]^{k[4]}$ (mod p)]
   Y_2 := [168 ,530 ,55 ,755]
6. Broadcasts Y_2

Player 3:

7. Chooses a random or pseudo-random key-List K (each key is constructed using the protocol "Key Creation".
   Let K := [107, 95 , 925, 461]
8. Computes
   Y_3 :=LockCards(Y_1,K)
   Y_3:= [$Y_2[1]^{k[1]}$ (mod p) , $Y_2[2]^{k[2]}$ (mod p) , $Y_2[3]^{k[3]}$ (mod p) , $Y_2[4]^{k[4]}$ (mod p)]
   Y_3 := [142 ,76 ,100 ,462]
9. Broadcasts Y_3

Now we set Main-Deck := Y_3 = [142 ,76 ,100 ,462]. All cards have been masked and locked. Now we deal the first card in the Main-Deck to to player 3. We'll execute the protocol "Single-Card-Deal (for VM-L-OL, VM-VL and VM-VL-VUM)" (section 3.7.14).

Let x := Main-Deck[1] = 142 .

Player 1:

1. Set q_1 := 99 * 445 (mod 1018) = 281 and broadcast q_1. (q_1 is the card key : the product of the key used for the first card in the locking round and the masking key)

Player 2:

1. Set q_2 := 601 * 299 (mod 1018) = 531 and broadcast q_2.

Player 3:

1. Set q_3 := 107 * 101 (mod 1018) = 627 and keeps q_3 secret.
2. Computes w := $\underline{q_1}$* .. * $\underline{q_n}$. (the q values broadcast by the players)
   w := 281 * 531 * 627 (mod 1018) = 79
3. Computes y := OpenCard(x,w) = $D_w(x)$. First we compute the v, the key inverse of w.
   v := w^{-1} = 79^{-1} (mod 1018)
   v := 567
   y := x^v
   y := 142^{567} = 118
4. Now we can see that 118 = Open-Deck[3] so the card dealt to player 3 is the "three of hearts". No other player knows this card, because q_3 was kept secret.

If we compose the three permutations we see that this is correct. These are the movements the third card (3) has done while being shuffled:

Open-Deck[3] = 118 (the first third place in the open deck)

$F_1^{-1}[3] = 2$	(the second place after player 1 shuffles)
$F_2^{-1}[2] = 4$	(then the fourth place after player 2 shuffles)
$F_3^{-1}[4] = 1$	(then the first place of the Main-Deck, after player 3 shuffles)

And we dealt the first card (1) of the Main-Deck, so the dealt card is the correct one.

9. Performance

The main disadvantage of existing protocols is their poor performance for current home PCs. MPF overcomes this problem. We've analyzed an implementation of MPF and simulated other implementations and compared them against theoretical performance of other protocols as described in [CR05].

Our aim has been to compare the protocols on a realistic environment, which should take into account that:

- The protocol is run over the Internet, and users are spread all over the world.
- User computers are home PCs.
- There is no hardware acceleration.
- Users will play several games together.
- CPU usage for the GUI during game play is 10%.

Instead of actually running the protocol on the Internet (which makes results very difficult to repeat), we used a LAN but forced restrictions on the latency of packets (simulating high round-trip time) and throughput (simulating low bandwidth). Because users send each other data, the limiting factor in bandwidth is up-stream direction and not down-stream bandwidth, which is considerably higher for an average ISP. We've tried to be conservative in numbers not to over-estimate performance. We simulate a simple poker-like game where cards are dealt and afterwards there is a showdown. During game play we use the free CPU time to pre-compute the following shuffles. Processors are never left idle, either they are computing or sending/receiving data, and never both at the same time. Verification protocols are run in parallel, so as to maximize CPU use. Amortized game time represents the time users must wait for a new game to begin, after the first game has been completed, which takes into account pre-calculation. Parameters are presented in Table 10.

Property	Value
Computer type	1.8 Mhz CPU, single core.
Number of users	10
Number of cards in the deck	52
Number of games to play	10
Average game time (not including protocol computation time)	40 seconds (*)
Time of 1024-bit modular exponentiation	1,5 ms (using GMP)
Time of 1024-bit modular multiplication	87 uS
Security threshold for interactive protocols	2^{-20}
Security threshold for non-interactive proofs	2^{-80}
Cards dealt to each player	5
Internet round-trip time	150 ms
Up-stream bandwidth	20 Kb/sec
Multiplication time on an Elliptic Curve over the Z/pZ finite field with a 160-bit prime.	1,5 ms
(*) This is an average online poker game time, according to Wikipedia.	

Table 10: Simulation Scenario

For classical cryptography we use a 1024 modulus, which provides adequate security for current communications as stated [NIST800-57]. For Elliptic Curve Cryptography (ECC) we use a 160-bit modulus and assume multiplication performance comparable to Z/pZ modular exponentiation. This is a matter of discussion which method is faster for 1024 bit finite fields (ECC Diffie-Hellman vs. Diffie-Hellman). It is widely agreed that ECC performance is superior when p becomes larger, such as 2048 bits, but here we limit our analysis to 1024 bit modulus. It should be noted that we assume the figures given in [CR05] also take into account full CPU utilization due to parallelization. Also all protocols assume computers have access to a broadcast medium or there is central server with unlimited input and output bandwidth which broadcasts received messages to all the remaining players, but bandwidth will still be limited by senders and receivers. We also assume the broadcasting server can also send private messages, without consuming the remaining players bandwidth. We have not taken into account Internet round-trip time and the performance penalty (overhead) in sending and receiving a message due to the difficulty of calculating how the other protocols can benefit from parallelization. For example, [Cr86] protocol sends large amounts of tiny messages and so its protocol time may be grater than the value shown by the fact that message overhead is not accounted.

Table 11 summarizes the results of the comparison.

Operation	MPF over ECC base VSM-VL using CO-VP	MPF over ECC base VSM-VL using HMVP	MPF over PH base VSM-VL using CO-VP	MPF over PH base VSM-VL using HMVP	KKOT90	BS03	Cr86	CSD04b
Shuffle Time	14.61 s	27.31 s	36.26 s	57.97 s	333.80 s	273.39 s	415.54 s	102.29 s
All cards draw time (5 cards for each player)	0.17 s	0.17 s	0.43 s	0.43 s	21.00 s	35.94 s	17.28 s	46.29 s
All cards show time (showdown)	0.12 s	0.12 s	0.39 s	0.39 s	0.78 s	46.30 s	0.08 s	46.30 s
Total processing time for first game	14.90 s	27.60 s	37.08 s	58.79 s	355.58 s	355.63 s	432.89 s	194.88 s
Amortized processing time per game	1.60 s	2.87 s	4.06 s	22.79 s	319.58 s	319.63 s	396.89 s	158.88 s

Table 11: Comparison of Protocol Times

10. Open Questions & Conclusions

Mental Poker protocol inspired Goldwasser and Micali to invent probabilistic encryption, in order to provide a provable framework to secure mental poker games. This fact has deviated the development of mental poker protocols to those that rely on probabilistic encryption. Nevertheless commutative ciphers bring performance benefits and can be as strong as some standardized cryptographic assumptions, such as the Decisional Diffie-Hellman (DDH) assumption. We conjecture that probabilistic encryption prevents the existence of abrupt drop-out recovery protocols resistant against collusion, which excludes protocols based on threshold schemes for secret sharing. Abrupt drop-out recovery protocols are required for any real life usage affecting money or reputation scores. If the conjecture is true, then Mental Poker protocol development would need to shift back to deterministic ciphers.

In this thesis we have proposed a new framework to create secure mental poker protocols (MPF). The framework addresses theoretical and practical issues, such as security, performance, drop-out tolerance and modular design. We've also built PHMP and ECMP protocols derived from MPF. The performance of PHMP and ECMP was analyzed theoretically and PHMP was then implemented and tested successfully. As far as our knowledge of state of the art in mental poker protocols, PHMP/ECMP are the first to provide acceptable performance for common card games over the Internet. In the design of MPF, at least six novel ideas have contributed: the use of the CUOC property, the use double encryptions per player (masking/locking), the CO-VP and FI-UniVP protocols, and the abrupt drop out recovery protocol.

11. Bibliography

[BDH02] Feng Bao. Robert Deng, Huafei Zhu (2002). "Variations of Diffie–Hellman problem". In ICICS ’03, volume 2836 of LNCS 2003, pages 301—312, Springer-Verlag.

[BM07] Borisov, N. and Minami, K. 2007. Single-bit re-encryption with applications to distributed proof systems. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society* (Alexandria, Virginia, USA, October 29 - 29, 2007). WPES '07. ACM, New York, NY, 48-55. DOI= http://doi.acm.org/10.1145/1314333.1314341

[B96] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[BS03] A. Barnett and N. Smart. Mental poker revisited. In Proc. Cryptography and Coding, volume 2898 of Lecture Notes in Computer Science, pages 370--383. Springer-Verlag, December 2003.

[CD04] Jordi Castellà-roca and Josep Domingo-ferrer. A Non-Repudiable Bitstring Commitment Scheme Based on a Public-Key . Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2 - Volume 2, Page: 778.

[CR03] Jordi Castella-Roca, Josep Domingo-Ferrer, Andreu Riera, Joan Borrell. Practical Mental Poker without a TTP Based on Homomorphic Encryption (2003). Progress in Cryptology-Indocrypt'2003

[CR05] Jordi Castellà-Roca, Contributions to Mental Poker. Autonomous University of Barcelona, Doctoral Programme in Computer Science and Artificial Intelligence. Data of public reading: Sep. 9, 2005.

[Cr86] C. Crépeau. A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face. In A. M. Odlyzko, editor, Advances in Cryptology - Crypto '86, volume 263, pages 239--250, Berlin, 1986. Springer-Verlag. Lecture Notes in Computer Science.

[dBo90] B. den Boer. More efficient match-making and satisfiability: the five card trick. In Advances in Cryptology: EUROCRYPT ’89 Proceedings, Lecture Notes in Computer Science 434, pp. 208–217, 1990.

[E94]. J. Edwards, Implementing Electronic Poker: A Practical Exercise in Zero-Knowledge Interactive Proofs. Master’s thesis, Department of Computer Science, University of Kentucky, 1994.

[FIPS186] Federal Information Processing Standards, Publication 186, 1994 May 19.

[FLS90] Feige, U., Lapidot, D., and Shamir, A. 1990. Multiple non-interactive zero knowledge proofs based on a single random string. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science* (October 22 - 24, 1990). SFCS. IEEE Computer Society, Washington, DC, 308-317 vol.1. DOI=http://dx.doi.org/10.1109/FSCS.1990.89549

[HIRLL99] A Pseudorandom Generator from any One-way Function. Johan Håstad, Russell Impagliazzo, Leonid A. Levin, Michael Luby. SIAM Journal on Computing 1999, volume 28, pages12—24.

[K97] Neal Koblitz. *A Course in Number Theory and Cryptography*, Graduate Texts in Math. No. 114, Springer-Verlag, New York, 1987. Second edition, 1994.

[KKOT90] K. Kurosawa, Y. Katayama, W. Ogata, and S. Tsujii. General public key cryptosystems and mental poker protocols. In Ivan B. Damgšard, editor, Advances in Cryptology -Eurocrypt '90, volume 473 of Lecture Notes in Computer Science, pages 374--388. Springer-Verlag, 1990.

[Lys02] Anna Lysyanskaya. Unique signatures and verifiable random functions from the dh-ddh separation. Lecture Notes In Computer Science; Vol. 2442. Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology Pages: 597 – 612. Available from http://theory.lcs.mit.edu/anna/papers/lys02.ps.

[MRV99] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In 40th Annual Symposium on Foundations of Computer Science, pages 120–130, New York, October 1999. IEEE.

[N91] Moni Naor. Bit Commitment Using Pseudo-Randomness, Journal of Cryptology, 1991. Volume 5, pages 151—158.

[NIST800-57] Recommendation for Key Management – Part 1 : general, NIST Special Publication 800-57. March, 2007

[NR98] V. Niemi and A. Renvall. Secure Multiparty Computations Without Computers. Theoretical Computer Science, 191(1-2):173–183, 1998.

[PS04] P. Syverson , A Taxonomy of Replay Attacks, In Proceedings of the 7th IEEE Computer Security Foundations Workshop.

[SRA81] A. Shamir, R, L. Rivest, and L. Adleman. Mental poker. Mathematical Gardner, pages 37--43, 1981.

[St01] A. Stiglic. Computations with a deck of cards. Theoretical Computer Science, 259(1-2):671–678, 2001.

[VSP06] V. Cortier, S. Delaune and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. Journal of Computer Security 14 (2006) pages 1–43, IOS Press.

[WC01] D.S. Wong and A.H. Chan, Efficient and mutually authenticated key exchange protocol for low power computing devices, in: Proc. 7th International Conference on the Theory and Application of Cryptology and Information Security *(ASIACRYPT'01)*, Vol. 2248 of *LNCS*, Gold Coast, Australia, Springer, 2001, pp. 272–289.b

[W06] Stephen A. Weis, New Foundations for Efficient Authentication, Commutative Cryptography, and Private Disjointness Testing. 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 MIT, May 2006.

[WW09] Tzer-jen Wei and Lih-Chung Wang, Fast Mental Poker Protocol, Cryptology ePrint 2009/439