ROYALE: A Framework for Universally Composable Card Games with Financial Rewards and Penalties Enforcement

Bernardo David[1], Rafael Dowsley[24⋆], and Mario Larangeira[34⋆⋆]

[1] IT University of Copenhagen, Denmark
bernardo@bmdavid.com
[2] Aarhus University, Denmark
rafael@cs.au.dk
[3] Tokyo Institute of Technology, Japan
mario@c.titech.ac.jp
[4] IOHK, Hong Kong

Abstract. While many tailor made card game protocols are known, the vast majority of those lack three important features: mechanisms for distributing financial rewards and punishing cheaters, composability guarantees and flexibility, focusing on the specific game of poker. Even though folklore holds that poker protocols can be used to play any card game, this conjecture remains unproven and, in fact, does not hold for a number of protocols (including recent results). We both tackle the problem of constructing protocols for general card games and initiate a treatment of such protocols in the Universal Composability (UC) framework, introducing an ideal functionality that captures card games that use a set of core card operations. Based on this formalism, we introduce Royale, the first UC-secure general card games which supports financial rewards/penalties enforcement. We remark that Royale also yields the first UC-secure poker protocol. Interestingly, Royale performs better than most previous works (that do not have composability guarantees), which we highlight through a detailed concrete complexity analysis and benchmarks from a prototype implementation.

1 Introduction

Online card games have become highly popular with the advent of online casinos, which act as trusted third parties performing the roles of both dealers and cashiers. However, a malicious casino (potentially compromised by an insider attacker) can easily subvert game outcomes [34]. Solving this issue has inspired a long line of research on *mental poker*, *i.e.* playing poker among distrustful players without relying on a trusted third party [3, 38, 19, 20, 31, 28, 17, 24, 37, 30, 33, 32]. Nevertheless, the aforementioned mental poker protocols did not provide formal security definitions or proofs. In fact, concrete flaws in the protocols of [38, 37] (resp. [3, 17]) have been

⋆ This project has received funding from the European research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme (grant agreement No 669255).

⋆⋆ This work was supported by the Input Output Cryptocurrency Collaborative Research Chair funded by Input Output HK.

identified in [30] (resp. [22]). Moreover, even if some of these protocols can be proven secure, they do not ensure that aborting adversaries cannot prevent the game to reach an outcome or that honest players receive the resulting financial rewards.

Techniques for ensuring that players receive their rewards according to game outcomes were only developed recently by Andrychowicz *et al.* [2, 1, 6, 25], building on decentralized cryptocurrencies. Their techniques also prevent misbehavior (including aborts) by imposing financial penalties to adversaries who are caught deviating from the protocol. Basically, they ensure that honest players either receive the rewards determined by the game outcome or a share of the penalty imposed to the adversary in case an outcome is not reached. These techniques were subsequently improved by Kumaresan *et al.* [26, 7], who also applied them to constructing protocols for secure card games with financial rewards/penalties. However, neither of these works provided formal security definitions and proofs for their card game protocols.

The first security definition and provably secure protocol for secure poker with financial rewards/penalties enforcement were recently proposed by David *et al.* [22], which still only captures the specific game of poker. Moreover, the protocol of [22] lacks composability guarantees, meaning that it cannot be arbitrarily executed along with copies of itself and other protocols. In fact, none of the previous mental poker protocols are composable and, consequently, re-purposing them for playing other games would void their security guarantees, contradicting the folklore belief that poker protocols yield protocols for any card game. While the recent work of [21] constructs composable card game protocols, it only captures games without secret state (*i.e.* it cannot be used to instantiate games where bluffing is a key element, such as poker). Our work closes this gap by proposing a protocol for playing general card games that use a set of core card operations with security proven in the Universal Composability (UC) [11] framework, also yielding the first UC-secure protocol for the specific case of poker.

1.1 Our Contributions

We initiate a composable treatment of card game protocols, introducing both the first ideal functionality for general card games and the first UC-secure tailor-made protocol for *general* card games. Our functionality and matching protocol support core operations that can be used to construct a large number of different card games, as opposed to previous protocols, which focus specifically on the game of poker. Besides capturing a large number of card games, our protocol enforces financial rewards/penalties while achieving efficiency comparable to previous works without UC-security. In fact, for practical parameters, a DDH-based instantiation of our protocol is concretely more efficient than most previous works, most of which have no provable security guarantees. Our contributions are summarized as follows:

- The *first* ideal functionality for general card games that can be expressed in terms of a set of core card operations: $\mathcal{F}_{\mathsf{CG}}$;
- Royale, the *first* provably secure protocol for *general* card games satisfying $\mathcal{F}_{\mathsf{CG}}$;
- Royale is proven to UC-realize our functionality in the restricted programmable and observable global random oracle model [9], being the *first universally composable* card game protocol (also yielding the first UC-secure poker protocol);
- An efficient mechanism for financial rewards/penalties enforcement in Royale, and a detailed efficiency analysis showing it outperforms previous works for practical parameters and benchmarks obtained from a prototype implementation.

As a first step in providing a composable treatment, we introduce an ideal functionality that captures general card games. It is parameterized by a program describing the flow of the game being modeled, differently from the ideal functionality introduced in [22], which only captures the flow of a poker game. This program determines the order in which the functionality carries out a number of operations that are used throughout the game, as well as the conditions under which a player wins or loses the game. Namely, the game rules can request a number of core card operations: public shuffling of closed cards on the table, private opening of cards (towards only one player, used for drawing cards), public opening of cards and shuffling of cards in a player's private hand (which can be used to securely swap cards among players). Moreover we provide an interface for the game rules to request public actions from the players (allowing players to broadcast their course of action), such as placing a bet or choosing a card from the table. We achieve financial rewards/penalties enforcement by following the basic approach of [7] based on stateful contracts, which are modeled as a separate ideal functionality in our construction. Each player deposits a *collateral* that is forfeited (and distributed among the other players) in case he behaves maliciously during protocol execution. If a player suspects that another player is misbehaving (*e.g.* failing to send a message), a complaint is sent to the stateful contract functionality, which mediates the protocol execution until the conflict is resolved or a culprit is found, resulting in the termination of the protocol after collateral deposit distribution. As pointed out in [7], such a stateful contract functionality can be implemented based on smart contracts on blockchain-based systems such as Ethereum [8].

Finally, we construct Royale, a protocol for general card games that is proven to UC-realize our functionality with the help of a stateful contract. It is constructed in a modular fashion based on generic signature, threshold encryption and non-interactive zero-knowledge (NIZK) proofs that can be efficiently instantiated under standard computational assumptions (DDH) in the restricted programmable and observable global random oracle model of [9]. As the contract is ultimately implemented by a blockchain-based solution, one of the main bottlenecks in such a protocol is the amount of on-chain storage required for executing the stateful contract, which must analyze the protocol execution and determine whether a player has correctly executed the protocol or not when a complaint is issued. We achieve low on-chain storage complexity by providing compact *checkpoint witnesses* that allow the players to prove that the protocol has been correctly executed (or not), differently from [7], which requires large amounts of the protocol transcript to be sent to the contract.

The individual card operations in our protocol are inspired by Kaleidoscope [22], which achieves the desired efficiency for the specific case of poker. However, Kaleidoscope is based specifically on the DDH assumption and does not achieve UC-security, Kaleidoscope's security proof involves a simulator that makes heavy use of extraction of witnesses of NIZK proofs of knowledge based on the Fiat-Shamir heuristic, which require rewinding the adversary in the security proof, an operation that is not allowed in proofs in the UC framework. While substituting such Fiat-Shamir NIZKs for UC-secure ZK proofs would solve this issue, the efficiency of the resulting protocol would be greatly affected, since current UC-secure constructions [10] are significantly less efficient than the simple NIZKs used in Kaleidoscope. We overcome this obstacle without sacrificing efficiency through subtle modifications to the protocol itself, employing NIZK proofs of membership and a novel proof strategy that only requires the simulator to generate simulated proofs, eliminating the need for rewinding.

1.2 Related Works

Even though there is a large number of previous works on protocols for secure card games, the problem of aborting adversaries and reward distribution for poker games has only been (efficiently) addressed recently [7]. Moreover, as previously discussed, formal security definitions and proofs for secure card game protocols were only recently introduced in Kaleidoscope [22]. Since we aim at addressing both the issues of composability and financial penalties/rewards distribution, we center our discussion on the works of [7, 22], which are more closely related to this goal. See [22] for a comprehensive discussion of efficiency and concrete security issues of previous works.

Enforcing Financial Rewards and Penalties: Most games of poker are played with money at stake, posing two central challenges that were overlooked in the first poker protocols but need to be solved in order to allow for practical deployment: (1) protecting against potentially aborting cheaters and (2) ensuring that winners receive their rewards. In the case of general secure computation, these challenges were only recently addressed in an efficient way by Bentov *et al.* [7] with further optimizations of an approach previously developed and pursued in [2, 1, 6, 25, 26]. The central idea in the general purpose secure computation protocol of [7] is to execute an unfair protocol without any interaction with the cryptocurrency network, relying on a single stateful contract that handles funds distribution and financially punishes misbehaving parties. Before the unfair protocol is executed, the stateful contract receives deposits of funds that will be distributed according to the protocol output as well as of collateral funds that will be used to punish misbehaving parties and compensate honest parties. In case a party suspects cheating, it "complains" to the stateful contract, which will mediate the protocol execution until a cheater is found or the complaint is solved (so that execution can proceed off-chain). In case a party is found to be cheating, its collateral funds are distributed among the honest parties and the protocol execution ends. If the protocol reaches an output, the stateful contract distributes the funds deposited at the onset of execution according to the output. Bentov *et al.* [7] apply this general approach to tailor-made poker protocols [33, 32], aiming at implementing a secure poker protocol with higher efficiency than their general purpose secure computation protocol. However, their tailor-made protocol is not formally proven secure and, even if found to be secure, has efficiency issues, as discussed in the remainder of this section.

Formal Security Guarantees: The vast majority of poker protocols [31, 19, 20, 28, 3, 38, 17, 24, 37, 30, 33, 32, 26, 7] claim different levels of security but do not provide formal securities. Besides making it hard to assess the exact security offered by such protocols, the lack of clear security definitions and proofs has led to concrete security flaws in many of these protocols [38, 37, 3, 17], as pointed out in [30, 22]. While Bentov *et al.* [7] argue that their framework can be directly applied to tailor-made poker protocols to provide financial rewards/penalties enforcement with high efficiency, they do not provide a security proof for such a direct application of their framework to tailor-made protocols nor describe the properties the underlying poker protocol should satisfy. Their work specifically mentions the protocols of [33, 32] as potential building blocks. However, [33, 32] are not formally proven secure. Using such protocols as building blocks in a black-box way without a clear security definition and proofs can lead to both security and composition issues. Moreover, even if proven secure, [33, 32] face efficiency issues for practical parameters. In the poker case, the lack of formal security definitions and proofs was only recently remedied by Kaleidoscope [22], which introduced both the first security definition for poker functionalities and a matching protocol, considering financial rewards/penalties enforcement.

Efficiency Issues: As Royale is the first work to consider general card games, we compare the efficiency of each card operation provided by Royale to the similar operations provided in previous works on poker protocols. The most costly operation is the shuffling of cards. The protocol of Barnett and Smart[3] (that serves as the basis for many subsequent protocols) and the protocol of Wei and Wang [33] (cited as a potential building block in [7]) rely on a cut-and-choose based ZK proof of shuffle correctness, incurring high computational and communication overheads. A subsequent work by Wei [32] (also cited as a potential building block in [7]) improves on the complexity of the shuffle procedure by eliminating the need for cut-and-choose but still requires a large number of rounds (more than $4n$ rounds, where n is the number of players), which is also the case of [33]. The Kaleidoscope [22] protocol employs a novel shuffling phase based on efficient NIZK proofs of shuffle correctness, achieving better concrete efficiency both in terms of communication and computation than previous works for practical parameters, while only requiring n rounds (for n players). The shuffling procedure of a DDH-based instantiation of Royale (Section 3) inherits the same high efficiency of the Kaleidoscope shuffle while achieving UC-security. The computational, communication and round complexities of opening cards in Royale are very similar to those of previous works, which already achieved high efficiency for these operations. For a more detailed discussion, we refer to Section 4.

Composability Issues: The need for arbitrary composability naturally arises in poker and general card game protocols with financial rewards/penalties enforcement, since those protocols need to use other cryptographic protocols, *e.g.* secure channels and cryptocurrency protocols. This is specially critical in the case of general card game protocols, where card operations are arbitrarily mixed and matched in order to create different games, which can potentially cause serious security issues in protocols without arbitrary composability guarantees. However, none of the previous works on poker or card games protocols have considered this issue, and Kaleidoscope [22], the only poker protocol with provable security guarantees, only achieves sequential composability. The UC framework [11] is widely used to reason about arbitrary composability for cryptographic protocols. The main obstacle to providing a proof of security for Kaleidoscope as well as other previous poker protocols lies in their use of NIZK proofs of Knowledge obtained from applying the Fiat-Shamir transformation to Sigma protocols, heavily relying on rewinding for extracting witnesses in their security proofs. In Royale, this is solved by employing a proof strategy that only requires the simulator to generate simulated NIZKs without sacrificing efficiency.

2 Preliminaries

We denote the security parameter by κ and sampling an element x uniformly at random from a set $\mathcal{X}$ by $x \xleftarrow{\$} \mathcal{X}$. See Appendix A for further notation.

Re-Randomizable Threshold PKE: A re-randomizable threshold public key encryption (RTE [36]) scheme is a central in our protocols. Intuitively, we focus of the (n, n)-Threshold case, where the n parties need to cooperate in the decryption. We present formal definitions in Appendix A. A summary of the main RTE algorithms used in our construction is given below:

– $\mathsf{KeyGen}(\mathsf{param})$ takes as input parameters param and outputs a public key pk_i and a secret key sk_i.
– $\mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ is a deterministic algorithm that takes as input a set of public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ and outputs a combined public key pk.
– $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$ takes as input a public key pk and a plaintext message m, and outputs a ciphertext ct.
– $\mathsf{ReRand}(\mathsf{pk}, \mathsf{ct})$ is a re-randomization algorithm that takes as input a public key pk and a ciphertext ct, and outputs a re-randomized ciphertext ct'.
– $\mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct})$ is a deterministic algorithm that takes as input a secret key share sk_i and a ciphertext ct, and outputs a decryption share d_i.
– $\mathsf{ShareCombine}(\mathsf{ct}, \mathsf{d}_1, \ldots, \mathsf{d}_n)$ is a deterministic decryption share combining algorithm that takes as input a ciphertext ct and a set of decryption shares $(\mathsf{d}_1, \ldots, \mathsf{d}_n)$, and outputs a plaintext message m.

NIZKs for Relations over RTE: We need a number of NIZKs for relations over the RTE scheme we employ. Basically, a NIZK scheme $\mathsf{NIZK}_{\mathcal{R}}$ for relation $\mathcal{R}$ and algorithm Prov that takes as input $(x, w) \in \mathcal{R}$ and outputs a proof π and an algorithm Verify that takes as input (x, π) and outputs 1 if the proof is valid and 0 otherwise. For the sake of clarity, we define the following generic relations for which we need to prove statements in zero-knowledge and describe our protocols and simulators in terms of those: *(1)* $\mathcal{R}_1$ *- Correctness of public key share:* This relation shows that the prover knows the randomness used for generating a public/secret key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ and the secret key sk_i; *(2)* $\mathcal{R}_2$ *- Correctness of decryption share:* This relation shows that the prover used the secret key sk_i corresponding to its public key pk_i for computing a decryption share d_i of a ciphertext ct; *(3)* $\mathcal{R}_3$ *- Correctness of shuffle:* This relation shows that the prover correctly shuffled a set of ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$ by re-randomizing them with randomness $(r_1, \ldots, r_m)$ and permuting them with a permutation Π. Formal definitions for these NIZKs and an instantiation from sigma protocols in the Global Random Oracle model are presented in Appendix A and Appendix B.

Security Model: We prove our protocols secure in the UC framework [11]. UC-secure protocols retain their security even when used in parallel with other cryptographic protocols or as building blocks of more complex applications. We consider static malicious adversaries, who can arbitrarily deviate from the protocol but only corrupt parties before execution starts. It is known that UC-secure two-party and multiparty protocols for non-trivial functionalities require a setup assumption [14]. The main setup assumption for our work is the global random oracle model [5] modelled as the $\mathcal{G}_{\mathsf{rpoRO}}$-hybrid model [9], a digital signature functionality $\mathcal{F}_{\mathsf{DSIG}}$ from [12], and a smart contract functionality (defined in Section 3). See Appendix A for details.

The Stateful Smart Contract Functionality $\mathcal{F}_{\mathsf{SC}}$: We follow the approach of Bentov *et al.* [7] in describing a functionality $\mathcal{F}_{\mathsf{SC}}$ that models a *stateful contract.* Such a contract receives coins from the players in a check-in procedure and, after that, is only activated in case a player wishes to report misbehavior or wishes to leave the game, retrieving the coins that he owns at that point. While Bentov *et al.* describe a stateful contract functionality that models execution of general programs with secure cash distribution (*i.e.* the output of the computation determines how coins are distributed among honest players) and penalties for misbehavior, we focus on the specific case of card games. That means that our functionality only allows a program GR that specifies the game rules to execute specific card operations instead of general computation. The card operations supported by our protocol are the ones described in functionality $\mathcal{F}_{\mathsf{CG}}$. However, as discussed in Apprendix C, we can extend $\mathcal{F}_{\mathsf{CG}}$ by incorporating other functionalities for which UC protocols exist. In this case, GR is also allowed to specify the operations described in these functionalities and the stateful contract modelled by GR is also responsible for ensuring that the protocols realizing these functionalities are correctly executed. We describe $\mathcal{F}_{\mathsf{SC}}$ in Figure 1.

3 Secure Protocol for Playing Card Games

In this section we describe a protocol that realizes functionality $\mathcal{F}_{\mathsf{CG}}$ (defined in Appendix C) with the help of a smart contract. The role of the smart contract is to make sure that all players are executing the card operations (and other game actions) as specified by the game rules programmed in GR and punish (resp. compensate) malicious (resp. honest) players in case of dispute. The basic idea is to follow the secure computation with financial penalties framework initiated by [2, 1] and have each player send to the contract an amount of coins that will be used for betting in the protocol and another amount of coins used as collateral. If a player suspects that another player is cheating in the game or misbehaving in protocol execution, it sends a request to the smart contract, which verifies protocol execution and, in case a player was actually found to be misbehaving, financially punishes the malicious player by distributing its collateral coins among the honest players.

Protocol π_{CG}: We construct a Protocol π_{CG} that realizes $\mathcal{F}_{\mathsf{CG}}$ in a modular fashion. The main building block of this protocol is a re-randomizable threshold public key encryption (RTE) and associated non-interactive zero-knowledge proofs (NIZK). Moreover, we will rely on a global random oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$ to apply the Fiat-Shamir heuristic to sigma protocols used for instantiating these NIZKs as described in Appendix B. Additionally, a standard digital signature functionality $\mathcal{F}_{\mathsf{DSIG}}$ will be used as building block in this protocol. Later on, we will describe a concrete instantiation of the protocol under the DDH assumption.

In this protocol, the players start by jointly generating a public key for the RTE scheme along with individual secret key shares. The main idea is to represent open cards as ciphertexts of the RTE scheme encrypting a card value $[1, \ldots, 52]$ without any randomness (or randomness 0) while closed cards are shuffled such that they are represented by a re-randomized ciphertext that is permuted in way that cannot be reversed by any proper subset of the players (so that no collusion of players can trace the shuffling back to the open cards). The shuffle operation is done by having each player act in sequence, taking turns in rerandomizing all ciphertexts representing cards and permuting the resulting rerandomized ciphertexts, while proving in zero-knowledge that these operations were executed correctly. When a closed (shuffled) card has to be revealed to a player, all other players send decryption shares of the ciphertext representing this card computed with their respective secret keys, along with proofs that these decryption shares have been correctly computed.

Throughout the protocol, after the players perform a card operation or answer an action request from GR, they jointly generate a checkpoint witness proving that the operation has been completed successfully. These checkpoint witnesses contain signatures by all users on the current state of the protocol,*i.e.* ciphertexts representing

Functionality $\mathcal{F}_{\mathsf{SC}}$

$\mathcal{F}_{\mathsf{SC}}$ is executed with players $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and is parametrized by a timeout limit τ, and the values of the initial stake t, the compensation q and the security deposit $d \geq (n-1)q$. There is an embedded program GR that represents the game's rules and a protocol verification mechanism pv.

- **Players Check-in:** When execution starts, $\mathcal{F}_{\mathsf{SC}}$ waits to receive from each player $\mathcal{P}_i$ the message $(\textsc{checkin}, sid, \mathcal{P}_i, \mathsf{coins}(d+t), \mathsf{SIG}.vk_i, \mathsf{pk}_i, \pi^i_{\mathcal{R}_1})$ containing the necessary coins, its signature verification key, its share of the threshold ElGamal public-key and the zero-knowledge proof of knowledge of the secret-key's share. Record the values and send $(\textsc{checkedin}, sid, \mathcal{P}_i, \mathsf{SIG}.vk_i, \mathsf{pk}_i, \pi^i_{\mathcal{R}_1})$ to all players. If some player fails to check-in within the timeout limit τ or if a message $(\textsc{checkin-fail}, sid)$ is received from any player, then send $(\textsc{compensation}, \mathsf{coins}(d+t))$ to all players who have checked in and halt.
- **Player Check-out:** Upon receiving $(\textsc{checkout-init}, sid, \mathcal{P}_j)$ from $\mathcal{P}_j$, send to all players $(\textsc{checkout-init}, sid, \mathcal{P}_j)$. Upon receiving $(\textsc{checkout}, sid, \mathcal{P}_j, \mathsf{payout}, \sigma_1, \ldots, \sigma_n)$ from $\mathcal{P}_j$, verify that $\sigma_1, \ldots, \sigma_n$ are valid signatures by the players $\mathcal{P}_1, \ldots, \mathcal{P}_n$ on $(\mathtt{CHECKOUT}|\mathsf{payout})$ according to $\mathcal{F}_{\mathsf{DSIG}}$. If all tests succeed, for $i = 1, \ldots, n$, send $(\textsc{payout}, sid, \mathcal{P}_i, \mathsf{coins}(w))$ to $\mathcal{P}_i$, where $w = \mathsf{payout}[i] + d$, and halt.
- **Recovery:** Upon receiving a recovery request $(\textsc{recovery}, sid)$ from a player $\mathcal{P}_i$, send the message $(\textsc{request}, sid)$ to all players. Upon receiving $(\textsc{response}, sid, \mathcal{P}_j, \mathsf{Checkpoint}_j, \mathsf{proc}_j)$ from some player $\mathcal{P}_j$ with checkpoint witnesses (which are not necessarily relative to the same checkpoint as the ones received from other players) and witnesses for the current procedure; or an acknowledgement of the witnesses previous submitted by another player, forward this message to the other players. Upon receiving replies from all players or reaching the timeout limit τ, fix the current procedure by picking the most recent checkpoint that has valid witnesses (*i.e.* the most recent checkpoint witness signed by all players $\mathcal{P}_i$). Verify the last valid point of the protocol execution using the current procedure's witnesses, the rules of the game GR, and pv. If some player $\mathcal{P}_i$ misbehaved in the current phase (by sending an invalid message), then send $(\textsc{compensation}, \mathsf{coins}(d + q + \mathsf{balance}[j] + \mathsf{bets}[j]))$ to each $\mathcal{P}_j \neq \mathcal{P}_i$, send the leftover coins to $\mathcal{P}_i$ and halt. Otherwise, proceed with a mediated execution of the protocol until the next checkpoint using the rules of the game GR and pv to determine the course of the actions and check the validity of the answer. Messages $(\textsc{nxt-stp}, sid, \mathcal{P}_i, \mathsf{proc}, \mathsf{round})$ are used to request from player $\mathcal{P}_i$ the protocol message for round round of procedure proc according to the game's rules specified in GR, who answer with messages $(\textsc{nxt-stp-rsp}, sid, \mathcal{P}_i, \mathsf{proc}, \mathsf{round}, \mathsf{msg})$, where msg is the requested protocol message. All messages $(\textsc{nxt-stp}, sid, \ldots)$ and $(\textsc{nxt-stp-rsp}, sid, \ldots)$ are delivered to all players. If during this mediated execution a player misbehaves or does not answer within the timeout limit τ, penalize him and compensate the others as above, and halt. Otherwise send $(\textsc{recovered}, sid, \mathsf{proc}, \mathsf{Checkpoint})$, to the parties once the next checkpoint $\mathsf{Checkpoint}$ is reached, where proc is the procedure for which $\mathsf{Checkpoint}$ was generated.

Fig. 1. Functionality $\mathcal{F}_{\mathsf{SC}}$.

cards and each player's balance and current bets. If a player suspects that any other player is cheating (or has aborted) during an execution, it complains to the smart contract, providing its latest checkpoint. The execution is then mediated by the smart contract, which receives (and broadcasts) all messages generated by the players. If the smart contract detects that a player is cheating in this execution (by examining the transcript), it punishes the misbehaving player by distributing its collateral coins among the honest players. We describe Protocol π_{CG} in Figures 2, 3 and 4.

Security Analysis: Due to page limit the security analysis is given Appendix D.

A DDH-Based Instantiation: We now describe an instantiation of the Protocol π_{CG} that is secure under the popular DDH assumption in the random oracle model (*i.e.* substituting $\mathcal{F}_{\mathrm{RO}}$ for a cryptographic hash function). The main components we need to construct in order to instantiate our protocol are the re-randomizable threshold public-key encryption scheme RTE and the NIZKs Proof of Membership schemes $\mathsf{NIZK}_{\mathcal{R}_1}, \mathsf{NIZK}_{\mathcal{R}_2}, \mathsf{NIZK}_{\mathcal{R}_3}$ for relations $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3$. It was shown in [36, Appendix C.2], that the threshold version of the ElGamal cryptosystem is a secure re-randomizable threshold public-key encryption scheme under the DDH assumption. Moreover, it was also shown in [36, Appendix C.2] that there exist NIZKs $\mathsf{NIZK}_{\mathcal{R}_1}, \mathsf{NIZK}_{\mathcal{R}_2}, \mathsf{NIZK}_{\mathcal{R}_3}$ for relations $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3$ secure under the DDH assumption. $\mathsf{NIZK}_{\mathcal{R}_1}$ can be implemented by the sigma protocol of Schnorr [29], $\mathsf{NIZK}_{\mathcal{R}_2}$ can be implemented by the protocol of Chaum and Pedersen [18] and $\mathsf{NIZK}_{\mathcal{R}_3}$ can be implemented by the protocol of Bayer and Groth [4]. Notice that the zero-knowledge argument of shuffle correctness of Bayer and Groth [4] requires a common reference string that consists of random group elements such that the discrete logarithm of these elements in a given base is unknown. We point out that such a common reference string can be trivially constructed before π_{CG} is run by coin tossing, which can be UC-realized based on UC-secure commitments [11, 14]. UC-secure commitments can be efficiently constructed in the restricted programmable and observable global random oracle model as proven in [9]. Even though these protocols are interactive, they can be made non-interactive through the Fiat-Shamir heuristic [23, 27]. Notice that their simulators are straight-line since they only need to program the random oracle. As for the digital signature functionality $\mathcal{F}_{\mathsf{DSIG}}$, it is known that $\mathsf{EUF\text{-}CMA}$ signature schemes (*e.g.* DSA and ECDSA) realize $\mathcal{F}_{\mathsf{DSIG}}$. If we use the resulting DDH-based instantiation to implement poker, we obtain a protocol very similar to the Kaleidoscope [22], thus obtaining a universally composable protocol for poker with rewards and penalties that matches the best current (but not UC-secure) protocol.

4 Efficiency Analysis

Royale is both the first cryptographic protocol to support general card games that use a set of core card operations and one of the very few based on generic primitives, making it hard to compare its efficiency with previous works that are based on specific computational assumptions and focused on poker. Therefore, we estimate and compare the computational, communication and round complexities of each individual card operation in the works that introduce the previously most efficient (but unproven) poker protocols with the card operations in the DDH-based instantiation of Royale (described in Section 3). For the comparison, we consider the works of Barnett and Smart [3], and the protocols proposed as a building block for the (unproven) tailor-made poker protocol of Bentov *et al.* [7]: Wei and Wang [33] and Wei [32]. We remark that these previous works have not been formally proven secure. Moreover, differently from Royale, even if these previous works can be proven to implement a game of poker, using their card operations arbitrarily might cause security issues, as they are not composable.

Protocol π_{CG} (First Part)

Let RTE be a secure re-randomizable threshold public-key encryption For $i \in \{1, 2, 3\}$, let $\mathsf{NIZK}_{\mathcal{R}_i} = (\mathsf{Prov}, \mathsf{Verify}, \mathsf{Sim}, \mathsf{Ext})$ be a NIZK proof of membership scheme for the relation $\mathcal{R}_i$.Protocol π_{CG} is parametrized by a security parameter 1^κ, RTE parameters $\mathsf{param} \leftarrow \mathsf{Setup}(1^\kappa)$, a timeout limit τ, the values of the initial stake t, the compensation q, the security deposit $d \geq (n-1)q$ and an embedded program GR that represents the rules of the game. In all queries (SIGN, sid, m) to $\mathcal{F}_{\mathsf{DSIG}}$, the message m is implicitly concatenated with $\mathtt{NONCE}$ and $\mathtt{cnt}$, where $\mathtt{NONCE} \overset{\$}{\leftarrow} \{0, 1\}^\kappa$ is a fresh nonce (sampled individually for each query) and $\mathtt{cnt}$ is a counter that is increased after each query. Every player $\mathcal{P}_i$ rejecting signatures that reuse nonces and implicitly concatenates the corresponding $\mathtt{NONCE}$ and $\mathtt{cnt}$ values with message m in all queries $(\text{VERIFY}, sid, m, \sigma, \mathsf{SIG}.vk')$ to $\mathcal{F}_{\mathsf{DSIG}}$. Protocol π_{CG} is executed by players $\mathcal{P}_1, \ldots, \mathcal{P}_n$ interacting with functionalities $\mathcal{F}_{\mathsf{CG}}$, $\mathcal{G}_{\mathsf{rpoRO}}$ and $\mathcal{F}_{\mathsf{DSIG}}$ as follows:

- **Checkpoint Witnesses:** After the execution of a procedure, the players store a checkpoint witness that consists of the lists $\mathcal{C}_O, \mathcal{C}_C, \mathcal{C}_1, \ldots, \mathcal{C}_n$, the vectors $\mathsf{balance}$ and bets as well as a signature by each of the other players on the concatenation of all these values. Each signature is generated using $\mathcal{F}_{\mathsf{DSIG}}$ and all players check all signatures using the relevant procedure of $\mathcal{F}_{\mathsf{DSIG}}$. Old checkpoint witnesses are deleted. If any check fails for $\mathcal{P}_i$, he goes to the recovery procedure.
- **Recovery Triggers:** All signatures and zero-knowledge proofs in received messages are verified by default. Players are assumed to have loosely synchronized clocks and, after each round of the protocol starts, players expect to receive all messages sent in that round before a timeout limit τ. If a player $\mathcal{P}_i$ does not receive an expected message from a player $\mathcal{P}_j$ in a given round before the timeout limit τ, $\mathcal{P}_i$ considers that $\mathcal{P}_j$ has aborted. After the check-in procedure, if any player receives an invalid message or considers that another player has aborted, it proceeds to the recovery procedure.
- **Tracking Balance and Bets:** Every player $\mathcal{P}_i$ keeps a local copy of the vectors $\mathsf{balance}$ and bets, such that $\mathsf{balance}[j]$ and $\mathsf{bets}[j]$ represent the balance and current bets of each player $\mathcal{P}_j$, respectively. To keep the copies up to date, every player performs:
  - At each point that GR specifies that a betting action from $\mathcal{P}_i$ takes place, player $\mathcal{P}_i$ broadcasts a message $(\text{BET}, sid, \mathcal{P}_i, bet_i)$, where bet_i is the value of its bet. It updates $\mathsf{balance}[i] = \mathsf{balance}[i] - b_i$ and $\mathsf{bets}[i] = \mathsf{bets}[i] + b_i$.
  - Upon receiving a message $(\text{BET}, sid, \mathcal{P}_j, bet_j)$ from $\mathcal{P}_j$, player $\mathcal{P}_i$ sets $\mathsf{balance}[j] = \mathsf{balance}[j] - b_j$ and $\mathsf{bets}[j] = \mathsf{bets}[j] + b_j$.
  - When GR determines that player $\mathcal{P}_j$ receives an amount pay_j and has its bet amount updated to b'_j, player $\mathcal{P}_i$ sets $\mathsf{balance}[j] = \mathsf{balance}[j] + pay_j$ and $\mathsf{bets}[j] = b'_j$.
- **Executing Actions:** Each $\mathcal{P}_i$ follows GR that represents the rules of the game, performing the necessary card operations, as well as updates on the list of card and balance and bet vectors, in the order specified by GR. If GR request an action with description $act-desc$ from $\mathcal{P}_i$, all the players output $(\text{ACT}, sid, \mathcal{P}_i, act-desc)$ and $\mathcal{P}_i$ executes any necessary operations. $\mathcal{P}_i$ broadcasts $(\text{ACTION-RSP}, sid, \mathcal{P}_i, act-rsp, \sigma_i)$, where $act-rsp$ is his answer and σ_i his signature on $act - rsp$, and outputs $(\text{ACTION-RSP}, sid, \mathcal{P}_i, act - rsp)$. Upon receiving this message, all other players check the signature, and if it is valid output $(\text{ACTION-RSP}, sid, \mathcal{P}_i, act - rsp)$. If a player $\mathcal{P}_j$ believes cheating happened, he proceeds to the recovery procedure.
- **Compensation:** Upon receiving from $\mathcal{F}_{\mathsf{SC}}$ $(\text{COMPENSATION}, sid, \mathcal{P}_i, \mathsf{coins}(w))$, output this message and halt.

Fig. 2. Protocol π_{CG} (First Part).

Protocol π_{CG} (Second Part)

- **Check-in:** Every player $\mathcal{P}_i$ proceeds as follows:
  1. Send $(\textsc{keygen}, sid)$ to $\mathcal{F}_{\mathsf{DSIG}}$, receiving $(\textsc{verification key}, sid, \mathsf{SIG}.vk_i)$.
  2. Sample $r_i \xleftarrow{\$} \{0,1\}^\kappa$ and generate a key pair $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(\mathsf{param}, r_i)$ and a NIZK of public key correctness $\pi^i_{\mathcal{R}_1}$ by computing $\mathsf{NIZK}_{\mathcal{R}_1}.\mathsf{Prov}$ with (r_i, sk_i) as witness.
  3. Send $(\textsc{checkin}, sid, \mathcal{P}_i, \mathsf{coins}(d+t), \mathsf{SIG}.vk_i, \mathsf{pk}_i, \pi^i_{\mathcal{R}_1})$ to $\mathcal{F}_{\mathsf{SC}}$.
  4. For $\mathcal{P}_j \neq p_i$, upon receiving $(\textsc{checkedin}, sid, \mathcal{P}_j, \mathsf{SIG}.vk_j, \mathsf{pk}_j, \pi^j_{\mathcal{R}_1})$ from $\mathcal{F}_{\mathsf{SC}}$, check if $\pi^j_{\mathcal{R}_1}$ is valid. If valid, output $(\textsc{checkedin}, sid, \mathcal{P}_j)$.
  5. Upon receiving valid check-in from all parties, compute $\mathsf{pk} \leftarrow \mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. Initialize the internal lists of open cards $\mathcal{C}_O$, of closed cards $\mathcal{C}_C$ and of private cards of each player $\mathcal{P}_i$, $\mathcal{C}_i$, as empty sets. We assume parties have a sequence of unused card id values (*e.g.* a counter). Initialize vectors $\mathsf{balance}[j] = t$ and $\mathsf{bets}[j] = 0$ for $j = 1, \ldots, n$.
  6. If $\mathcal{P}_i$ fails to receive a check-in of another party $\mathcal{P}_j$ within the timeout limit τ, it requests $\mathcal{F}_{\mathsf{SC}}$ to dropout and receive its coins back.
- **Create Card:** To create a card with value v, every player $\mathcal{P}_i$ selects the next unused card id id, stores (id, v) in $\mathcal{C}_O$ and outputs $(\textsc{newcard}, sid, \mathsf{id}, v)$.
- **Shuffle Cards:** To shuffle a set of cards with id values $\mathsf{id}_1, \ldots, \mathsf{id}_m$, $\mathcal{P}_i$ removes all the (eventual) cards $(\mathsf{id}_k, v_{\mathsf{id}_k})$, for $k \in \{1, \ldots, m\}$, that are in the list of opened cards $\mathcal{C}_O$ from that list and adds $(\mathsf{id}, \mathsf{ct}_{\mathsf{id}_k})$, for $\mathsf{ct}_{\mathsf{id}_k} \leftarrow \mathsf{Enc}(\mathsf{pk}, v_{\mathsf{id}_k}, 0)$, in $\mathcal{C}_C$. Define $(\mathsf{ct}^0_{\mathsf{id}_1}, \ldots, \mathsf{ct}^0_{\mathsf{id}_m}) = (\mathsf{ct}_{\mathsf{id}_1}, \ldots, \mathsf{ct}_{\mathsf{id}_m})$, where the right-hand side cards are stored, together with the respective id values, in the internal list $\mathcal{C}_C$. For $j = 1, \ldots, n$:
  1. If $j \neq i$, upon receiving the message $(\textsc{shuffle}, sid, \mathcal{P}_j, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}^j_{\mathsf{id}_1}, \ldots, \mathsf{ct}^j_{\mathsf{id}_m}, \pi^j_{\mathcal{R}_3})$ from $\mathcal{P}_j$, $\mathcal{P}_i$ verifies if $\pi^j_{\mathcal{R}_3}$ is valid.
  2. If $j = i$, sample a random permutation Π and, for $k = 1, \ldots, m$, let $r_k \xleftarrow{\$} \{0,1\}^\kappa$ and $\mathsf{ct}^i_{\mathsf{id}_k} \leftarrow \mathsf{ReRand}(\mathsf{pk}, \mathsf{ct}^{i-1}_{\Pi(\mathsf{id}_k)}, r_k)$. Broadcast $(\textsc{shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}^i_{\mathsf{id}_1}, \ldots, \mathsf{ct}^i_{\mathsf{id}_m}, \pi^i_{\mathcal{R}_3})$, where $\pi^i_{\mathcal{R}_3}$ is generated by computing $\mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Prov}$ with $(\Pi, (r_1, \ldots, r_m))$ as witness.

  Every player $\mathcal{P}_i$ sets its internal list of closed cards $\mathcal{C}_C$ to $\big((\mathsf{id}_1, \mathsf{ct}^n_{\mathsf{id}_1}), \ldots, (\mathsf{id}_m, \mathsf{ct}^n_{\mathsf{id}_m})\big)$ and outputs $(\textsc{shuffled}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$.
- **Shuffle Private Cards:** In order to shuffle a set of private cards with id values $\mathsf{id}_1, \ldots, \mathsf{id}_m$ belonging to player $\mathcal{P}_j$, player $\mathcal{P}_i$ proceed as follows:
  - If $i = j$, sample a random permutation Π and let $r_k \xleftarrow{\$} \{0,1\}^\kappa$, $\mathsf{ct}'_{\mathsf{id}_k} \leftarrow \mathsf{ReRand}(\mathsf{pk}, \mathsf{ct}_{\Pi(\mathsf{id}_k)}, r_k)$ for $k = 1, \ldots, m$. Broadcast $(\textsc{privshuffle}, sid, \mathcal{P}_j, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}'_{\mathsf{id}_1}, \ldots, \mathsf{ct}'_{\mathsf{id}_m}, \pi^j_{\mathcal{R}_3})$, where $\pi^j_{\mathcal{R}_3}$ is generated by computing $\mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Prov}$ with $(\Pi, (r_1, \ldots, r_m))$ as witness.
  - If $i \neq j$, upon receiving $(\textsc{PrivShuffle}, sid, \mathcal{P}_j, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}'_{\mathsf{id}_1}, \ldots, \mathsf{ct}'_{\mathsf{id}_m}, \pi^j_{\mathcal{R}_3})$ from $\mathcal{P}_j$, verify if $\pi^i_{\mathcal{R}_3}$ is valid.

  $\mathcal{P}_j$ outputs $(\textsc{private-shuffled}, sid, (\mathsf{id}_1, v'_1), \ldots, (\mathsf{id}_m, v'_m))$, where the new card values $v'_1, \ldots, v'_m$ associated to each id value are known to him, and the other parties output $(\textsc{private-shuffled}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$. All players update their local list $\mathcal{C}_C$.

Fig. 3. Protocol π_{CG} (Second Part).

Protocol π_{CG} (Third Part)

- **Open Public Card:** In order to open a public card $\mathsf{ct}_{\mathsf{id}}$, each $\mathcal{P}_i$ proceeds as follows:
  - Compute $\mathsf{d}_i \leftarrow \mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct}_{\mathsf{id}})$ and generate a NIZK of decryption share correctness $\pi^i_{\mathcal{R}_2}$ by computing $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Prov}$ with (r_i, sk_i) as witness (where r_i was used in generating $(\mathsf{pk}_i, \mathsf{sk}_i)$) and broadcast $(\textsc{opencard}, sid, \mathcal{P}_i, \mathsf{id}, \mathsf{d}_i, \pi^i_{\mathcal{R}_2})$.
  - Upon receiving $(\textsc{opencard}, sid, \mathcal{P}_j, \mathsf{id}, \mathsf{d}_j, \pi^j_{\mathcal{R}_2})$ from $\mathcal{P}_j$, verify if $\pi^j_{\mathcal{R}_2}$ is valid. Upon receiving valid decryption shares from all players, retrieve the value of card $\mathsf{ct}_{\mathsf{id}}$ by computing $v_{\mathsf{id}} \leftarrow \mathsf{ShareCombine}(\mathsf{d}_1, \ldots, \mathsf{d}_n)$. Add $(\mathsf{id}, v_{\mathsf{id}})$ to $\mathcal{C}_O$ and output $(\textsc{card}, sid, \mathsf{id}, v_{\mathsf{id}})$.
- **Open Private Card:** To open a private card $\mathsf{ct}_{\mathsf{id}}$ towards player $\mathcal{P}_j$, all players proceed as follows:
  - For $i \neq j$, $\mathcal{P}_i$ computes $\mathsf{d}_i \leftarrow \mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct}_{\mathsf{id}})$ and generates a NIZK of decryption share correctness $\pi^i_{\mathcal{R}_2}$ by computing $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Prov}$ with (r_i, sk_i) as witness (r_i is the randomness used to generate $(\mathsf{pk}_i, \mathsf{sk}_i)$) and sends $(\textsc{opencard}, sid, \mathcal{P}_i, \mathsf{id}, \mathsf{d}_i, \pi^i_{\mathcal{R}_2})$ to $\mathcal{P}_j$. Add id to $\mathcal{C}_j$.
  - Player $\mathcal{P}_j$, upon receiving $(\textsc{opencard}, sid, \mathcal{P}_i, \mathsf{id}, \mathsf{d}_i, \pi^i_{\mathcal{R}_2})$ from $\mathcal{P}_i$, verifies if $\pi^i_{\mathcal{R}_2}$ is valid. Upon receiving valid decryption shares from all other players, $\mathcal{P}_j$ computes $\mathsf{d}_j \leftarrow \mathsf{ShareDec}(\mathsf{sk}_j, \mathsf{ct}_{\mathsf{id}})$ and retrieves the value of the card by computing $v_{\mathsf{id}} \leftarrow \mathsf{ShareCombine}(\mathsf{d}_1, \ldots, \mathsf{d}_n)$. $\mathcal{P}_j$ adds id to $\mathcal{C}_j$ and outputs $(\textsc{card}, sid, \mathsf{id}, v_{\mathsf{id}})$.
- **Check-out:** A player $\mathcal{P}_j$ can initiate the check-out procedure and leave the protocol at any point that GR allows, in which case all players will receive the money that they currently own plus their collateral refund. The players proceed as follows:
  1. $\mathcal{P}_j$ sends $(\textsc{checkout-init}, sid, \mathcal{P}_j)$ to $\mathcal{F}_{\mathsf{SC}}$.
  2. Upon receiving $(\textsc{checkout-init}, sid, \mathcal{P}_j)$ from $\mathcal{F}_{\mathsf{SC}}$, each $\mathcal{P}_i$ (for $i = 1, \ldots, n$) sends $(\textsc{sign}, sid, (\mathtt{CHECKOUT}|\mathsf{payout}))$ to $\mathcal{F}_{\mathsf{DSIG}}$ (where payout is a vector containing the amount of money that each player will receive according to GR), obtaining $(\textsc{signature}, sid, (\mathtt{CHECKOUT}|\mathsf{payout}), \sigma_i)$ as answer. Player $\mathcal{P}_i$ sends σ_i to $\mathcal{P}_j$.
  3. For all $i \neq j$, $\mathcal{P}_j$ sends $(\textsc{verify}, sid, (\mathtt{CHECKOUT}|\mathsf{payout}), \sigma_i, \mathsf{SIG}.vk_i)$ to $\mathcal{F}_{\mathsf{DSIG}}$, where payout is computed locally by $\mathcal{P}_j$. If $\mathcal{F}_{\mathsf{DSIG}}$ answers all queries $(\textsc{verify}, sid, (\mathtt{CHECKOUT}|\mathsf{payout}), \sigma_i, \mathsf{SIG}.vk_i)$ with $(\textsc{verified}, sid, (\mathtt{CHECKOUT}|\mathsf{payout}), 1)$, $\mathcal{P}_j$ sends $(\textsc{checkout}, sid, \mathsf{payout}, \sigma_1, \ldots, \sigma_n)$ to $\mathcal{F}_{\mathsf{SC}}$. Otherwise, go to Recovery.
  4. Upon receiving $(\textsc{payout}, sid, \mathcal{P}_i, \mathsf{coins}(w))$ from $\mathcal{F}_{\mathsf{SC}}$, $\mathcal{P}_i$ outputs that and halts.
- **Recovery:** Player $\mathcal{P}_i$ proceeds as follows:
  - If player $\mathcal{P}_i$ activates the Recovery procedure, it sends $(\textsc{recovery}, sid)$ to $\mathcal{F}_{\mathsf{SC}}$.
  - Upon receiving $(\textsc{request}, sid)$ from $\mathcal{F}_{\mathsf{SC}}$, every player $\mathcal{P}_i$ sends $(\textsc{response}, sid, \mathcal{P}_i, \mathsf{Checkpoint}_i, \mathsf{proc}_i)$ to $\mathcal{F}_{\mathsf{SC}}$, where $\mathsf{Checkpoint}_i$ is $\mathcal{P}_i$'s latest checkpoint witness and proc_i is $\mathcal{P}_i$'s witness for the protocol phase that started after the latest checkpoint; or acknowledges another player's witness if it matches $\mathsf{Checkpoint}_i$.
  - Upon receiving $(\textsc{nxt-stp}, sid, \mathcal{P}_i, \mathsf{proc}, \mathsf{round})$ from $\mathcal{F}_{\mathsf{SC}}$, $\mathcal{P}_i$ sends $(\textsc{nxt-stp-rsp}, sid, \mathcal{P}_i, \mathsf{proc}, \mathsf{round}, \mathsf{msg})$ to $\mathcal{F}_{\mathsf{SC}}$, where msg is the protocol message that should be sent at round round of procedure proc of the protocol according to GR.
  - Upon receiving a message $(\textsc{nxt-stp-rsp}, sid, \mathcal{P}_j, \mathsf{proc}, \mathsf{round}, \mathsf{msg})$ from $\mathcal{F}_{\mathsf{SC}}$, every player $\mathcal{P}_i$ considers msg as the protocol message sent by $\mathcal{P}_j$ in round of procedure proc and take it into consideration for future messages.
  - Upon receiving a message $(\textsc{recovered}, sid, \mathsf{proc}, \mathsf{Checkpoint})$ from $\mathcal{F}_{\mathsf{SC}}$, every player $\mathcal{P}_i$ records $\mathsf{Checkpoint}$ as the latest checkpoint and continues protocol execution according to the game rules GR.

Fig. 4. Protocol π_{CG} (Third Part).

Instantiating the Building Blocks: We consider the protocols of Barnett and Smart [3], Wei and Wang [33] and Wei [32] to be instantiated with the same random oracle-based commitments and NIZKs based on the Fiat-Shamir heuristic used in our DDH-based instantiation of Royale. For the protocols of [3] and [33] a cut-and-choose security parameter of $s = 40$ is considered, while for the protocol of [32], we consider the parameter $k = 4$. In the NIZK of shuffle correctness used by Royale (the construction of [4]), the total number of cards is represented as $m = m_1 m_2$ and the choice of m_1 and m_2 affects both the computational and communication complexities. Even though the choice of m_1 and m_2 can be optimized to obtain either shorter or faster proofs, in our general comparison we assume that $m_1 = m_2 = \lceil \sqrt{m} \rceil$.

Computational Complexity: The estimation is in terms of modular exponentiations executed for each card operation, since these operations tend to dominate the complexity. We present the amount of local computation performed on Table 1. As previously observed, the Open Public Card and Open Private Card of all protocols in our comparison have roughly the same concrete complexity, while the Shuffle Cards phase is the main bottleneck. Notice that the two most efficient protocols in our comparison are Royale and Wei's protocol [32] (and consequently the instantiation of Bentov *et al.* [7] based on it), which has better asymptotic efficiency than Royale. However, we remark Royale achieves *better concrete efficiency* for *practical* parameters. For example, in a 6-player game and a standard deck of 52 cards (*e.g.* Poker), the Shuffle Cards phase of [32] requires approximately 3 times more exponentiations than Royale. Further estimations for practical parameters are in Appendix F.

Communication Complexity: We estimate the communication complexity in terms of the number of elements of $\mathbb{G}$ and elements of $\mathbb{Z}_p$ exchanged in each phase of the protocols in Table 1. In contrast to the case of computational complexity, we consider the total amount of data exchanged over the network by all players during each phase of the analyzed protocols. As it is the case with computational complexity, the Shuffle Cards phase constitutes the main bottleneck and dominates complexity. Notice that the most efficient protocols in our comparison are Royale and the protocol of Wei [32] (and consequently the instantiation of Bentov *et al.* [7] based on it). However, in this case, Royale actually achieves both better asymptotic communication complexity and *better concrete efficiency* than [32]. For example, in a 6-player game and a standard deck of 52 cards (*e.g.* Poker), the Shuffle Cards phase of [32] exchanges approximately 8 times more elements of $\mathbb{G}$ and twice more elements of $\mathbb{Z}_p$. Further estimations for practical parameters are in given in Appendix F.

Round Complexity: As in the previous cases, the Shuffle Cards phase is the main bottleneck. Royale's Shuffle Card phase requires only n rounds (where n is the number of players) while [33] and [32] require respectively $4n + 1$ and $4n + 3$ rounds. Hence, Royale has a clear advantage in round complexity, which results in better performance in high latency networks such as the Internet.

Checkpoint and On-Chain Storage Complexity: When the smart contract functionality $\mathcal{F}_{\mathsf{SC}}$ is implemented by a smart contract system running on top of a blockchain, the information sent by the players to $\mathcal{F}_{\mathsf{SC}}$ has to be stored in space-constrained blocks, raising a concern about on-chain storage complexity. First, we remark that Royale is designed in such a way that only the Check-in, Check-out and Recovery phases cause any information to be sent to $\mathcal{F}_{\mathsf{SC}}$ (and consequently stored in the blockchain), with the Recovery phase only being activated if a player misbehaves. In the Check-in phase, signature verification keys and public key shares (plus associated proofs of validity) for each players are registered with the smart contract, amounting to storing $(2\ \mathbb{G} + 2\ \mathbb{Z}_p)n$ bits, where n is the number of players. In the Check-out phase, the vector payout (of size $|\mathsf{payout}|$) along with signatures by each player are sent to the smart contract, amounting to $|\mathsf{payout}| + 2n\ \mathbb{Z}_p$ of storage. In the Recovery phase, the most up-to-date checkpoint witness is sent to the smart contract, which subsequently registers all other player's messages for the phase to be executed after this checkpoint witness was generated. The worst case for checkpoint witness size is that where all cards are still closed, resulting in size $2m\ \mathbb{G} + |\mathsf{id}|m + |\mathsf{balance}| + |\mathsf{bets}| + 2n\ \mathbb{Z}_p$ bits, where n is the number of players, m is the number of cards and $|\mathsf{id}|$,$|\mathsf{balance}|$ and $|\mathsf{bets}|$ are the sizes of card identification string id, vector $\mathsf{balance}$ and vector bets, respectively. The messages of the phase executed after the latest checkpoint amount to extra on-chain storage equal to the communication complexity of each phase (as estimated above). On the other hand, the protocol of Bentov *et al.* [7] (based on [32] or [33]), does not specify checkpoint witnesses (seemingly requiring the full transcript of the current poker game to be sent to the smart contract) nor offers any complexity estimates for Check-in and Check-out phases, making it hard to provide a meaningful comparison.

	Computational Complexity			Communication Complexity		
	Shuffle Cards	Open Private Card (drawer ;others)	Open Public Card	Shuffle Cards	Open Private Card (drawer ;others)	Open Public Card
[3]	$240m(n-1)$ $+161m$	$4n-3;3$	$4n$	$164nm\ \mathbb{G}$, $122nm\ \mathbb{Z}_p$	$45nm\ \mathbb{G}$, $(2n^2+$ $80n+2nm)\ \mathbb{Z}_p$	$n(17m+5)\ \mathbb{G}$, $n(m+18)\ \mathbb{Z}_p$
[7] ([33])	$(44n+1)m$	$4n-3;3$	$4n$	$3(n-1)\ \mathbb{G}$, $2(n-1)\ \mathbb{Z}_p$	$(n-1)\ \mathbb{G}$, $2(n-1)\ \mathbb{Z}_p$	$(n-1)\ \mathbb{G}$, $2(n-1)\ \mathbb{Z}_p$
[7] ([32])	$81m+2n$ $+25$	$4n-3;3$	$4n$	$3n\ \mathbb{G}$, $2n\ \mathbb{Z}_p$	$n\ \mathbb{G}$, $2n\ \mathbb{Z}_p$	$n\ \mathbb{G}$, $2n\ \mathbb{Z}_p$
Royale	$(2\log(\lceil\sqrt{m}\rceil)$ $+4n-2)m$	$4n-3;3$	$4n$	$n(2m+\lceil\sqrt{m}\rceil)\ \mathbb{G}$, $5n\lceil\sqrt{m}\rceil\ \mathbb{Z}_p$	$(n-1)\ \mathbb{G}$, $2(n-1)\ \mathbb{Z}_p$	$n\ \mathbb{G}$, $2n\ \mathbb{Z}_p$

Table 1. Complexities for each player in terms of modular exponentiations and group and ring elements $\mathbb{G}$ and $\mathbb{Z}_p$, for n players and m cards.

Benchmarks. We now present benchmarks of Royale obtained with a prototype implementation of the DDH-based instantiation, showcasing the efficiency of our protocol for practical parameters. Our prototype implementation was done in Haskell using NIST curve P-256. Experiments were conducted on a XPS 9370 with a i7 8550U CPU and 16 GB RAM running with Linux Fedora 28 (kernel 4.16). We analyze the network communication and execution time of Royale with different numbers of cards (denoted by m in the tables) and players (denoted by n in the tables). We focus on the following phases of Royale: Check-Out, Check-Out, Shuffle Cards, Shuffle Private Cards. Moreover, we analyse on-chain storage requirements for the Checkpoint Witnesses used in the Recovery Phase considering an implementation of the smart contract functionality $\mathcal{F}_{\mathsf{SC}}$ based on a smart contract that verifies individual steps of

Royale (*i.e.* checking NIZK, signature and encryption validity). We evaluate the execution time required by the aforementioned phases of Royale in milliseconds (ms) and consider network delays in terms of Round Trip Times (RTT). Our analysis shows that Royale achieves high computational efficiency, with network delays representing the main bottleneck. We analyze the on-chain storage required by Royale in terms of the size in kilobytes (KB) of the data stored by the smart contract in each phase, which is zero for all phases, except for Check-in, Check-out and Recovery. Our analysis shows that the on-chain footprints of these three latter phases is reasonably small for practical parameters. While the Recovery phase always requires storage of the must up-to-date checkpoint witness, it also requires players' messages for the current phase to be stored (*i.e.* the network communication required for each phase).

n	Check-In	Check-Out
2	0.25	0.38
4	0.51	0.75
6	0.76	1.13
8	1.02	1.5
10	1.27	1.88
12	1.52	2.25

Table 2. On-Chain Storage Size (in KB).

m n	52	104	208
2	200.64 + 1 RTT	387.67 + 1 RTT	886.32 + 1 RTT
4	401.28 + 2 RTT	775.33 + 2 RTT	1772.64 + 2 RTT
6	601.93 + 3 RTT	1163 + 3 RTT	2658.96 + 3 RTT
8	802.57 + 4 RTT	1550.66 + 4 RTT	3545.28 + 4 RTT
10	1003.21 + 5 RTT	1938.33 + 5 RTT	4431.6 + 5 RTT
12	1203.85 + 6 RTT	2326 + 6 RTT	5317.92 + 6 RTT

Table 3. Execution time in ms and Round-trip time (RTT) for the Shuffle Card.

The on-chain storage requirements of the Check-in and Check-Out Phases are presented in Table 2. Notice that all communication in these phases is done via the smart contract and does not depend on the number of cards. The execution time and network communication for the Shuffle Cards phase are presented in Table 3 and Table 4, respectively. The execution time is presented as the sum of the local computation time required of each player and the network Round Trip Times necessary for delivering this phase's messages. Checkpoint witnesses size for our implementation is presented in Table 5. As previously discussed, we consider the size of checkpoint witnesses in the worst case, where all cards are closed (which results in the largest representation). For the setting of a poker game with 52 cards and 6 players, we obtain a worst case checkpoint witness of less than 4 KB. In case the Recovery Phase is activated, the smart contract receives (and stores on-chain) both the latest checkpoint witness and the next messages to be generated in the protocol, corresponding to the network communication of the current phase. Further benchmark data are presented in Appendix F.

m n	52	104	208
2	13.73	24.49	40.73
4	27.45	48.98	81.47
6	41.18	73.48	122.2
8	54.91	97.97	162.94
10	68.63	122.46	203.67
12	82.36	146.95	244.41

Table 4. Network communication in the Shuffle Cards phase in (KB).

m n	52	104	208
2	3.61	7.06	13.97
4	3.77	7.22	14.13
6	3.92	7.38	14.28
8	4.08	7.53	14.44
10	4.23	7.69	14.59
12	4.39	7.84	14.75

Table 5. Checkpoint Witnesses on-chain storage size (KB).

References

1. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *FC 2014 Workshops*, volume 8438 of *LNCS*, pages 105–121. Springer, Heidelberg, March 2014.
2. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.
3. Adam Barnett and Nigel P. Smart. Mental poker revisited. In Kenneth G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *LNCS*, pages 370–383. Springer, Heidelberg, December 2003.
4. Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280. Springer, Heidelberg, April 2012.
5. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.
6. Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014.
7. Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 410–440. Springer, Heidelberg, December 2017.
8. Vitalik Buterin. White paper. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013. Accessed on 5/12/2017.
9. Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 280–312. Springer, 2018.
10. Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 449–467. Springer, Heidelberg, December 2011.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
12. Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, page 219. IEEE Computer Society, 2004.
13. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.
14. Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001.
15. Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 597–608. ACM Press, November 2014.
16. Ignacio Cascudo, Ivan Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, and Roberto Trifiletti. Additively homomorphic UC commitments

with optimal amortized overhead. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 495–515. Springer, Heidelberg, March / April 2015.

17. Jordi Castellà-Roca, Francesc Sebé, and Josep Domingo-Ferrer. Dropout-tolerant ttp-free mental poker. In Sokratis Katsikas, Javier López, and Günther Pernul, editors, *Trust, Privacy, and Security in Digital Business: Second International Conference, TrustBus 2005, Copenhagen, Denmark, August 22-26, 2005. Proceedings*, pages 30–40, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
18. David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993.
19. Claude Crépeau. A secure poker protocol that minimizes the effect of player coalitions. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 73–86. Springer, Heidelberg, August 1986.
20. Claude Crépeau. A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 239–247. Springer, Heidelberg, August 1987.
21. Bernardo David, Rafael Dowsley, and Mario Larangeira. 21 - bringing down the complexity: Fast composable protocols for card games without secret state. In Willy Susilo and Guomin Yang, editors, *Information Security and Privacy - 23rd Australasian Conference, ACISP 2018, Wollongong, NSW, Australia, July 11-13, 2018, Proceedings*, volume 10946 of *Lecture Notes in Computer Science*, pages 45–63. Springer, 2018.
22. Bernardo David, Rafael Dowsley, and Mario Larangeira. Kaleidoscope: An efficient poker protocol with payment distribution and penalty enforcement. To appear on Financial Cryptography and Data Security (FC) 2018, 2018. `http://eprint.iacr.org/2017/899`.
23. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.
24. Philippe Golle. Dealing cards in poker games. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pages 506–511, 2005.
25. Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 30–41. ACM Press, November 2014.
26. Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 195–206. ACM Press, October 2015.
27. David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 387–398. Springer, Heidelberg, May 1996.
28. Christian Schindelhauer. A toolbox for mental card games. Technical report, University of Lübeck, 1998.
29. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
30. Francesc Sebe, Josep Domingo-Ferrer, and Jordi Castella-Roca. On the security of a repaired mental poker protocol. *Information Technology: New Generations, Third International Conference on*, 00:664–668, 2006.
31. Adi Shamir, Ronald L Rivest, and Leonard M Adleman. Mental poker. In *The mathematical gardner*, pages 37–43. Springer, 1981.
32. Tzer-jen Wei. Secure and practical constant round mental poker. *Information Sciences*, 273:352–386, 2014.

33. Tzer-jen Wei and Lih-Chung Wang. A fast mental poker protocol. *Journal of Mathematical Cryptology*, 6(1):39–68, 2012.
34. Wikipedia. Online Poker. `https://en.wikipedia.org/wiki/Online_poker`, 2017. [Online; accessed 29-August-2017].
35. Bingsheng Zhang and Hong-Sheng Zhou. Brief announcement: Statement voting and liquid democracy. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *36th ACM PODC*, pages 359–361. ACM, July 2017.
36. Bingsheng Zhang and Hong-Sheng Zhou. Digital liquid democracy: How to vote your delegation statement. Cryptology ePrint Archive, Report 2017/616, 2017. `http://eprint.iacr.org/2017/616`.
37. Weiliang Zhao and Vijay Varadharajan. Efficient ttp-free mental poker protocols. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pages 745–750, 2005.
38. Weiliang Zhao, Vijay Varadharajan, and Yi Mu. A secure mental poker protocol over the internet. In *Proceedings of the Australasian Information Security Workshop Conference on ACSW Frontiers 2003 - Volume 21*, ACSW Frontiers '03, pages 105–109, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

A Additional Preliminaries

We denote the security parameter by κ. For a randomized algorithm F, $y \overset{\$}{\leftarrow} F(x)$ denotes running F with input x and its random coins, obtaining an output y. If we need to specify the coins r, we will use the notation $y \leftarrow F(x; r)$. We denote sampling an element x uniformly at random from a set $\mathcal{X}$ by $x \overset{\$}{\leftarrow} \mathcal{X}$. For a distribution $\mathcal{Y}$, we denote sampling y according to the distribution $\mathcal{Y}$ by $y \overset{\$}{\leftarrow} \mathcal{Y}$. We say that a function f is negligible in n if for every positive polynomial p there exists a constant c such that $f(n) < \frac{1}{p(n)}$ when $n > c$. We denote by $\mathsf{negl}(\kappa)$ the set of negligible functions in κ. Two ensembles $X = \{X_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ and $Y = \{Y_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ of binary random variables are said to be *statistically indistinguishable*, denoted by $X \approx_s Y$, if for all z it holds that $|\Pr[\mathcal{D}(X_{\kappa,z}) = 1] - \Pr[\mathcal{D}(Y_{\kappa,z}) = 1]|$ is negligible in κ for every probabilistic distinguisher $\mathcal{D}$. In case this only holds for non-uniform probabilistic polynomial-time (PPT) distinguishers we say that X and Y are *computationally indistinguishable* and denote it by $X \approx_c Y$.

A.1 Re-Randomizable Threshold PKE

A re-randomizable threshold public key encryption (RTE) scheme is a central building block for our protocols. We use the definition of RTEs from [36]. Intuitively, we focus of the (n, n)-Threshold case, where n parties need to cooperate to decrypt the ciphertexts. A RTE consists of a following tuple of PPT algorithms:

- $\mathsf{Setup}(1^\kappa)$ on input the security parameter κ outputs public parameters param, which are an implicitly input to all the other algorithms and known by all parties.
- $\mathsf{KeyGen}(\mathsf{param})$ takes as input parameters param and outputs a public key pk_i and a secret key sk_i.
- $\mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ is a deterministic algorithm that takes as input a set of public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ and outputs a combined public key pk.
- $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$ takes as input a public key pk and a plaintext message m, and outputs a ciphertext ct.

– $\mathsf{ReRand}(\mathsf{pk}, \mathsf{ct})$ is a re-randomization algorithm that takes as input a public key pk and a ciphertext ct, and outputs a re-randomized ciphertext ct'.
– $\mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct})$ is a deterministic algorithm that takes as input a secret key share sk_i and a ciphertext ct, and outputs a decryption share d_i.
– $\mathsf{ShareCombine}(\mathsf{ct}, \mathsf{d}_1, \ldots, \mathsf{d}_n)$ is a deterministic decryption share combining algorithm that takes as input a ciphertext ct and a set of decryption shares $(\mathsf{d}_1, \ldots, \mathsf{d}_n)$, and outputs a plaintext message m.
– $\mathsf{SimshareDec}\,(\mathsf{sk}_j, \mathsf{ct}, \mathsf{m}, \mathsf{m}')$ is a deterministic algorithm that takes as input a secret key share sk_j (for $j \in \{1, \ldots, n\}$), a ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$, a message m and an alternative message m', outputting a decryption share d_j such that $\mathsf{m}' \leftarrow \mathsf{ShareCombine}(\mathsf{ct}, \mathsf{d}_1, \ldots, \mathsf{d}_n)$, where $\mathsf{d}_i \leftarrow \mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct})$ for $i \in \{1, \ldots, n\} \setminus j$. Intuitively, given a ciphertext encrypting a known message and an alternative message, this algorithm outputs a decryption share that results in the ciphertext being decrypted to the alternative message.
– $\mathsf{CombineSK}(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$ is a deterministic algorithm that takes as input a set of secret key shares $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$ and outputs a combined secret key sk.
– $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$ is a deterministic decryption algorithm that takes as input a ciphertext ct and a secret key sk, and outputs a message m.
– $\mathsf{Trans}(\mathsf{ct}, \{\mathsf{sk}_i\}_{i \in \{1,\ldots,n\} \setminus j})$ is a deterministic algorithm that takes as input a ciphertext ct and a set of secret keys $\{\mathsf{sk}_i\}_{i \in \{1,\ldots,n\} \setminus j}$, and outputs a ciphertext ct'.

We define the security of such schemes as in [36]. Notice that we do not use algorithms $\mathsf{CombineSK}$, Dec and Trans in our protocol or proofs but they are necessary for defining the scheme's security.

Definition 1. RTE *is a secure re-randomizable threshold public key encryption if the following properties hold:*

Key Combination Correctness: *For every valid set of public/secret keys pairs* $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i \in \{1,\ldots,n\}}$*, for* $\mathsf{pk} \leftarrow \mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ *and* $\mathsf{sk} \leftarrow \mathsf{CombineSK}(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$*, the pair* $(\mathsf{pk}, \mathsf{sk})$ *is a valid key pair. Moreover, for every message* m *in* RTE*'s message space,* $\mathsf{ct} \overset{\$}{\leftarrow} \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$ *and* $d_i \leftarrow \mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct})$*, it holds that* $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = \mathsf{ShareCombine}(\mathsf{ct}, d_1, \ldots, d_n)$*.*

IND-CPA Security: *The public key cryptosystem* $\mathsf{PKE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is IND-CPA secure.*

Ciphertext transformative indistinguishability: *For every valid set of pairs of keys* $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i \in \{1,\ldots,n\}}$ *of public/secret key pairs,* $\mathsf{pk} \leftarrow \mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ *and* $\mathsf{sk} \leftarrow \mathsf{CombineSK}(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$*, and for every message* m *in* RTE*'s message space, ciphertext* $\mathsf{ct} \overset{\$}{\leftarrow} \mathsf{Enc}(\mathsf{pk}_j, \mathsf{m})$ *it holds that* $(\mathsf{param}, \mathsf{Trans}(\mathsf{ct}, \{\mathsf{sk}_i\}_{i \in \{1,\ldots,n\} \setminus j}) \approx_c (\mathsf{param}, \mathsf{Enc}(\mathsf{pk}, \mathsf{m}))$ *for every* $j \in \{1, \ldots, n\}$*.*

Unlinkability: *Let the experiment* $\mathsf{Unlink}_{\mathcal{A}}(1^\kappa)$ *be executed with an adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$*:*

1. *Generate* $\mathsf{param} \overset{\$}{\leftarrow} \mathsf{Setup}(1^\kappa)$ *and* $(\mathsf{pk}_i, \mathsf{sk}_i) \overset{\$}{\leftarrow} \mathsf{KeyGen}(\mathsf{param})$ *for* $i = 1, \ldots, n$*;*

2. $(\mathcal{I}, \mathsf{st}_1) \xleftarrow{\$} \mathcal{A}_1(\mathsf{param})$ *outputs a set* $\mathcal{I} \subset \{1, \ldots, n\}$ *of up to* $n-1$ *corrupted indices and state* st_1*;*
3. $(\mathsf{ct}_0, \mathsf{ct}_1, \mathsf{st}_2) \xleftarrow{\$} \mathcal{A}_2(\{\mathsf{pk}_i\}_{i \in \{1,\ldots,n\}}, \{\mathsf{sk}_j\}_{j \in \mathcal{I}}, \mathsf{st}_1)$*;*
4. *Choose* $b \xleftarrow{\$} \{0,1\}$ *and compute* $\mathsf{ct}' \xleftarrow{\$} \mathsf{ReRand}(\mathsf{pk}, \mathsf{ct}_b)$*, where the public key* $\mathsf{pk} \leftarrow \mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*;*
5. *For* $b' \xleftarrow{\$} \mathcal{A}_3(\mathsf{ct}', \mathsf{st}_2)$*, return 1 if* $b' = b$*, else return 0.*

For any PPT *adversary* $\mathcal{A}$*, it should hold that* $\mathsf{AdvUnlink}_{\mathcal{A}}(1^\kappa) \in \mathsf{negl}(1^\kappa)$*, where*

$$\mathsf{AdvUnlink}_{\mathcal{A}}(1^\kappa) = \left| \Pr\left[\, \mathsf{Unlink}_{\mathcal{A}}(1^\kappa) = 1 \,\right] - \frac{1}{2} \right| .$$

Share-simulation indistinguishability: *For every valid set of public/secret keys* $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i \in \{1,\ldots,n\}}$*, and* $\mathsf{pk} \leftarrow \mathsf{CombinePK}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*, for every plaintext messages* m, m' *in the message space of* RTE*, ciphertext* $\mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$*, index* $j \in \{1, \ldots, n\}$ *and every decryption share* $\mathsf{d}_j \leftarrow \mathsf{ShareDec}(\mathsf{sk}_j, \mathsf{ct})$*, the following holds*

$$(\mathsf{param}, \mathsf{ct}, \mathsf{SimshareDec}(\mathsf{sk}_j, \mathsf{ct}, \mathsf{m}, \mathsf{m}')) \approx_c (\mathsf{param}, \mathsf{ct}, \mathsf{d}_j).$$

A.2 NIZKs for Relations over RTE

We employ a number of non-interactive zero-knowledge proofs (NIZKs) proofs of membership for different relations, which we instantiate in the random oracle model (ROM) for the sake of efficiency. We adopt the notation and security definitions for NIZKs of [36, 35]. A scheme $\mathsf{NIZK}_{\mathcal{R}}$ for relation $\mathcal{R}$ is a tuple of algorithms $(\mathsf{Prov}, \mathsf{Verify}, \mathsf{Sim}, \mathsf{Ext})$ such that: Prov is a PPT algorithm that takes as input $(x, w) \in \mathcal{R}$ and outputs a proof π; Verify is a deterministic polynomial time algorithm that takes as input (x, π) and outputs 1 if the proof is valid and 0 otherwise; Sim is a PPT algorithm that takes as input an statement x and outputs a proof π and auxiliary string aux; Ext is a deterministic polynomial time algorithm that takes as input a pair (x, π) and outputs a witness w.

We need a number of NIZKs for relations over the RTE scheme we employ. For the sake of clarity, we define the generic relations for which we need to prove statements in zero-knowledge and describe our protocols and simulators in terms of those.

$\mathcal{R}_1$ **- Correctness of public key share:** This relation shows that the prover knows the randomness used for generating a public/secret key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ and the secret key sk_i.

$$\pi_{\mathcal{R}_1} \xleftarrow{\$} \mathsf{NIZK}_{\mathcal{R}_1}.\mathsf{Prov} \left\{ \begin{array}{c} (\mathsf{pk}_i), (r_i, \mathsf{sk}_i) : \\ (\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(\mathsf{param}; r_i) \end{array} \right\} .$$

$\mathcal{R}_2$ **- Correctness of decryption share:** This relation shows that the prover used the secret key sk_i corresponding to its public key pk_i for computing a decryption share d_i of a ciphertext ct.

$$\pi_{\mathcal{R}_2} \xleftarrow{\$} \mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Prov} \left\{ \begin{array}{c} (\mathsf{pk}_i, \mathsf{ct}, \mathsf{d}_i), (r_i, \mathsf{sk}_i) : \\ (\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(\mathsf{param}; r_i), \\ \mathsf{d}_i \leftarrow \mathsf{ShareDec}(\mathsf{sk}_i, \mathsf{ct}) \end{array} \right\} .$$

$\mathcal{R}_3$ **- Correctness of shuffle:** This relation shows that the prover correctly shuffled a set of ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$ by re-randomizing them with randomness $(r_1, \ldots, r_m)$ and permuting them with a permutation Π.

$$\pi_{\mathcal{R}_3} \leftarrow \mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Prov} \left\{ \begin{array}{c} (\mathsf{pk}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_m), (\mathsf{ct}'_1, \ldots, \\ \mathsf{ct}'_m)), (\Pi, (r_1, \ldots, r_m)) \ : \\ \forall i \in \{1, \ldots, m\}, \\ \mathsf{ct}'_{\Pi(i)} \leftarrow \mathsf{ReRand}(\mathsf{pk}, \mathsf{ct}_i; r_i) \end{array} \right\}.$$

Definition 2. *A NIZK Proof of Membership scheme in the ROM for relation* $\mathcal{R}$*,* $\mathsf{NIZK}^{\mathsf{RO}}_{\mathcal{R}}$*, is a tuple of algorithms* $(\mathsf{Prov}^{\mathsf{RO}}, \mathsf{Verify}^{\mathsf{RO}}, \mathsf{Sim}^{\mathsf{RO}}, \mathsf{Ext}^{\mathsf{RO}})$ *with access to a random oracle* RO *such that the following hold:*

- Completeness*: For any* $(x, w) \in \mathcal{R}$*,*

$$\Pr\left[\begin{array}{c} \rho \xleftarrow{\$} \{0,1\}^\kappa \ ; \ \pi \leftarrow \mathsf{Prov}^{\mathsf{RO}}(x, w; \rho) : \\ \mathsf{Verify}^{\mathsf{RO}}(x, \pi) = 0 \end{array} \right] \leq \mathsf{negl}(\kappa).$$

- Zero-Knowledge*: Let oracles* $\mathcal{O}_1$ *return* π *on queries* $(x, w) \in \mathcal{R}$ *for* $(\pi, \mathsf{aux}) \leftarrow \mathsf{Sim}^{\mathsf{O}}(x)$ *and* $\mathcal{O}_2$ *return* $\pi \leftarrow \mathsf{Prov}^{\mathsf{RO}}(x, w; \rho)$ *for* $\rho \xleftarrow{\$} \{0,1\}^\kappa$*. For any* PPT *distinguisher* $\mathcal{A}$*, we have*

$$\left| \begin{array}{c} \Pr\left[\mathcal{A}^{\mathsf{RO}, \mathcal{O}_1}(1^\kappa) = 1 \right] \\ -\Pr\left[\mathcal{A}^{\mathsf{RO}, \mathcal{O}_2}(1^\kappa) = 1 \right] \end{array} \right| \leq \mathsf{negl}(\kappa).$$

- Soundness*: For all adversaries* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c} (x, \pi) \leftarrow \mathcal{A}^{\mathsf{RO}}(1^\kappa) \ ; \ x \notin \mathcal{L}_{\mathcal{R}} \\ \wedge \ \mathsf{Verify}^{\mathsf{RO}}(x, \pi) = 1 \end{array} \right] \leq \mathsf{negl}(\kappa).$$

Since all of the NIZKs used in our protocols are in the (Global) ROM, we omit RO in the superscript $\mathsf{NIZK}^{\mathsf{RO}}$. We denote the individual algorithms $(\mathsf{Prov}, \mathsf{Verify}, \mathsf{Sim})$ of a NIZK Proof of Membership $\mathsf{NIZK}_{\mathcal{R}}$ for relation $\mathcal{R}$ by $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{Prov}$, $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{Verify}$, $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{Sim}$.

A.3 Universal Composability

We present a brief description of the UC framework originally given in [16] and refer interested readers to [11] for further details. In this framework, protocol security is analyzed under the real-world/ideal-world paradigm, *i.e.*, by comparing the real world execution of a protocol with an ideal world interaction with the primitive that it implements. The model includes a *composition theorem*, that basically states that UC secure protocols can be arbitrarily composed with each other without any security compromises. This desirable property not only allows UC secure protocols to effectively serve as building blocks for complex applications but also guarantees security in practical environments, where several protocols (or individual instances of protocols) are executed in parallel, such as the Internet.

In the UC framework, the entities involved in both the real and ideal world executions are modeled as PPT Interactive Turing Machines (ITM) that receive and deliver messages through their input and output tapes, respectively. In the ideal world execution, dummy parties (possibly controlled by an ideal adversary $\mathcal{S}$ referred to as the *simulator*) interact directly with the ideal functionality $\mathcal{F}$, which works as a trusted third party that computes the desired primitive. In the real world execution, several parties (possibly corrupted by a real world adversary $\mathcal{A}$) interact with each other by means of a protocol π that realizes the ideal functionality. The real and ideal executions are controlled by the *environment* $\mathcal{Z}$, an entity that delivers inputs and reads the outputs of the individual parties, the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$. After a real or ideal execution, $\mathcal{Z}$ outputs a bit, which is considered as the output of the execution. The rationale behind this framework lies in showing that the environment $\mathcal{Z}$ (that represents everything that happens outside of the protocol execution) is not able to efficiently distinguish between the real and ideal executions, thus implying that the real world protocol is as secure as the ideal functionality.

We denote by $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z, \bar{r})$ the output of the environment $\mathcal{Z}$ in the real-world execution of a protocol π between n parties with an adversary $\mathcal{A}$ under security parameter κ, input z and randomness $\bar{r} = (r_{\mathcal{Z}}, r_{\mathcal{A}}, r_{P_1}, \ldots, r_{P_n})$, where $(z, r_{\mathcal{Z}})$, $r_{\mathcal{A}}$ and r_{P_i} are respectively related to $\mathcal{Z}$, $\mathcal{A}$ and party i. Analogously, we denote by $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z, \bar{r})$ the output of the environment in the ideal interaction between the simulator $\mathcal{S}$ and the ideal functionality $\mathcal{F}$ under security parameter κ, input z and randomness $\bar{r} = (r_{\mathcal{Z}}, r_{\mathcal{S}}, r_{\mathcal{F}})$, where $(z, r_{\mathcal{Z}})$, $r_{\mathcal{S}}$ and $r_{\mathcal{F}}$ are respectively related to $\mathcal{Z}$, $\mathcal{S}$ and $\mathcal{F}$. The real world execution and the ideal executions are respectively represented by the ensembles $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} = \{\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z, \bar{r})\}_{\kappa\in\mathbb{N}}$ and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} = \{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z, \bar{r})\}_{\kappa\in\mathbb{N}}$ with $z \in \{0,1\}^*$ and a uniformly chosen $\bar{r}$.

In addition to these two models of computation, the UC framework also considers the $\mathcal{G}$-hybrid world, where the computation proceeds as in the real-world with the additional assumption that the parties have access to an auxiliary ideal functionality $\mathcal{G}$. In this model, honest parties do not communicate with the ideal functionality directly, but instead the adversary delivers all the messages to and from the ideal functionality. We consider the communication channels to be ideally authenticated, so that the adversary may read but not modify these messages. Unlike messages exchanged between parties, which can be read by the adversary, the messages exchanged between parties and the ideal functionality are divided into a *public header* and a *private header*. The public header can be read by the adversary and contains non-sensitive information (such as session identifiers, type of message, sender and receiver). On the other hand, the private header cannot be read by the adversary and contains information such as the parties' private inputs. We denote the ensemble of environment outputs that represents an execution of a protocol π in a $\mathcal{G}$-hybrid model as $\mathsf{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}$ (defined analogously to $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$). UC security is then formally defined as:

Definition 3. *An n-party ($n \in \mathbb{N}$) protocol π is said to UC-realize an ideal functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if, for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that, for every environment $\mathcal{Z}$, the following relation holds:*

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx_c \mathsf{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}.$$

We say that a protocol is *statistically secure*, if the same holds for all $\mathcal{Z}$ with unbounded computing power.

Adversarial Model: We consider malicious adversaries, who can arbitrarily deviate from the protocol. We consider *static* adversaries, meaning that the adversary has to corrupt parties before execution starts and the corrupted (or honest) parties remain so throughout the execution.

Setup Assumptions: It is known that UC-secure two-party and multiparty protocols for non-trivial functionalities require a setup assumption [14]. The main setup assumption for our work is the random oracle model [5] modelled as the $\mathcal{G}_{\mathsf{rpoRO}}$-hybrid model (desicussed in Appendix B). In order to obtain a generic and modular construction, we write our protocols in terms of a digital signature functionality $\mathcal{F}_{\mathsf{DSIG}}$, described in Figure 5 as defined in [12], and a smart contract functionality (defined in Section 3). It is also known that $\mathsf{EUF\text{-}CMA}$ signature schemes realize $\mathcal{F}_{\mathsf{DSIG}}$. Notice that this fact is used to show that our protocols can be realized based on practical digital signature schemes such DSA and ECDSA.

Functionality $\mathcal{F}_{\mathsf{DSIG}}$

$\mathcal{F}_{\mathsf{DSIG}}$ interacts with an ideal adversary $\mathcal{S}$, parties $P_1, \ldots, P_n$ and a signer P_s as follows:

- **Key Generation** Upon receiving a message $(\textsc{keygen}, sid)$ from some party P_s, verify that $sid = (P_s, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\textsc{keygen}, sid)$ to the adversary $\mathcal{S}$. Upon receiving $(\textsc{verification key}, sid, \mathsf{SIG}.vk)$ from $\mathcal{S}$, output $(\textsc{verification key}, sid, \mathsf{SIG}.vk)$ to P_s, and record the pair $(P_s, \mathsf{SIG}.vk)$.

- **Signature Generation** Upon receiving a message $(\textsc{sign}, sid, m)$ from P_s, verify that $sid = (P_s, sid')$ for some sid' . If not, then ignore the request. Else, send $(\textsc{sign}, sid, m)$ to $\mathcal{S}$. Upon receiving $(\textsc{signature}, sid, m, \sigma)$ from $\mathcal{S}$, verify that no entry $(m, \sigma, \mathsf{SIG}.vk, 0)$ is recorded. If it is, then output an error message to P_s and halt. Else, output $(\textsc{signature}, sid, m, \sigma)$ to P_s, and record the entry $(m, \sigma, v, 1)$.

- **Signature Verification** Upon receiving a message $(\textsc{verify}, sid, m, \sigma, \mathsf{SIG}.vk')$ from some party P_i, hand $(\textsc{verify}, sid, m, \sigma, \mathsf{SIG}.vk')$ to $\mathcal{S}$. Upon receiving $(\textsc{verified}, sid, m, \phi)$ from $\mathcal{S}$ do:
  1. If $\mathsf{SIG}.vk' = \mathsf{SIG}.vk$ and the entry $(m, \sigma, \mathsf{SIG}.vk, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key $\mathsf{SIG}.vk'$ is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
  2. Else, if $\mathsf{SIG}.vk' = \mathsf{SIG}.vk$, the signer P_s is not corrupted, and no entry $(m, \sigma', \mathsf{SIG}.vk, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, \mathsf{SIG}.vk, 0)$. (This condition guarantees unforgeability: If $\mathsf{SIG}.vk'$ is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
  3. Else, if there is an entry $(m, \sigma, \mathsf{SIG}.vk', f')$ recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
  4. Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{SIG}.vk', \phi)$.

  Output $(\textsc{verified}, sid, m, f)$ to P_i.

Fig. 5. Functionality $\mathcal{F}_{\mathsf{DSIG}}$.

B Instantianting NIZKs in the Global Random Oracle Model

We will rely on a Fiat-Shamit style transformation to instantiate NIZK proofs of membership for relations $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$ from special honest verifier zero-knowledge proof protocols for these relations. It is well known that the Fiat-Shamir transformation does not work in general in the UC framework due to its fundamental use of rewinding and programming of the random oracle. However, for the specific relations we want to prove in zero knowledge, as we are not interested in obtaining proofs of knowledge, we can use a Fiat-Shamir style transformation to obtain NIZK proofs of membership without requiring rewinding as long as the simulator can program the random oracle specifically on points that it randomly sampled (and are thus unknown to the adversary or the environment). While the original formulations of the Global Random Oracle model by Canetti *et al.* [13, 15] do not allow for any form of programmability, a recent result by Camenisch *et al.* [9] shows that a global random oracle can be programmed in a *restricted* form that is sufficient for for instantiating these NIZKs. The the restricted programmable and observable global random oracle model of [9] is formally defined by functionality $\mathcal{G}_{\mathsf{rpoRO}}$.

Basically, in the restricted programmable and observable global random oracle model of [9], the simulator can program the random oracle on points sampled uniformly at random in its execution with an internal copy of the adversary while keeping the simulation indistinguishable from a real execution for the environment. Camenisch *et al.* use this property to prove the security of cannonical commitment schemes where the message and uniform randomness are given as input to the random oracle in order to obtain the commitment as the resulting output. In this construction, in order to open a commitment to an arbitrary message, the simulator must program the random oracle in a random point that is unknown to the adversary and the environment (*i.e.* the message obtained from the commitment functionality concatenated with the randomness). Similarly, this property can be used to simulate NIZK proofs of membership obtained through the Fiat-Shamir transformation.

Notice that the Fiat-Shamir transformation consists in obtaining the challenge by querying the random oracle on the first message of a sigma protocol concatenated with the public information available to both the prover and verifier. The prover first generates the first message of the protocol, obtains the challenge from the random oracle and uses this challenge to generate the last message of the protocol, providing the first as the last messages as a NIZK. The verifier can check the validity of this proof by querying the random oracle on the first message provided by the prover concatenated with the public information, checking that the response is valid with respect to the response obtained from the random oracle. If the challenge can be arbitrarily chosen by the prover, it can craft first and last messages that form an accepting transcript together with the arbitrary challenge. The prover is precluded from doing so since it cannot predict the output of the random oracle. The simulator can program the random oracle in order to obtain such an arbitrary challenge and generate a proof without knwoing a witness. On the other hand, extracting a witness from such a NIZK requires the simulator to follow the steps of the sigma protocol simulator, rewinding the prover a number of times, which is not compatible with the UC framework.

Functionality $\mathcal{G}_{\mathsf{rpoRO}}$

$\mathcal{G}_{\mathsf{rpoRO}}$ is parameterized by an output size function ℓ, keeps initally empty lists $\mathsf{List}_{\mathcal{H}}$,$\mathsf{prog}$ and operates as follows:

- **Query:** On input $(\mathsf{HashQuery}, m)$ from a machine $(\mathcal{P}, \mathsf{sid})$ or from the adversary, parse m as $(s, m'$ and proceed as follows:
  - Look up h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists, sample $h \xleftarrow{\$} \{0,1\}^{\ell(n)}$ and set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$.
  - If this query is made by the adverssary, or if $s \neq \mathsf{sid}$, then add (s, m', h) to the (initally empty) list of illegitimate queries $\mathcal{Q}_s$.
  - Output $(\mathsf{HashConfirm}, h)$ to the caller.
- **Observe:** On input $(\mathsf{Observe}, \mathsf{sid})$ from the adversary, if $\mathcal{Q}_{\mathsf{sid}}$ does not exist yet, set $\mathcal{Q}_{\mathsf{sid}} = \emptyset$. Output $(\mathsf{ListObserve}, \mathcal{Q}_{\mathsf{sid}})$ to the adversary.
- **Program:** O input $(\mathsf{ProgramRO}, m, h)$ with $h \in \{0,1\}^{\ell(n)}$ from the adversary, ignore the input if there exists $h' \in \{0,1\}^{\ell(n)}$ such that $(m, h') \in \mathsf{List}_{\mathcal{H}}$ and $h \neq h'$. Otherwise, set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$ and $\mathsf{prog} = \mathsf{prog} \cup \{m\}$ and output $(\mathsf{ProgramConfirm})$ to the adversary.
- **IsProgrammed:** On input $(\mathsf{IsProgrammed}, \mathsf{m})$ from a machine $(\mathcal{P}, \mathsf{sid})$ or from the adversary, if the input was given by $(\mathcal{P}, \mathsf{sid})$, parse m as $(s, m'$ and, if $s \neq \mathsf{sid}$, ignore this input. Set $b = 1$ if $m \in mathsfprog$ (and $b = 0$ otherwise) and output $(\mathsf{IsProgrammed}, b)$ to the caller.

Fig. 6. Functionality $\mathcal{G}_{\mathsf{rpoRO}}$.

In our case, we construct NIZK proofs of membership, where only the soundness and zero-knowledge properties are needed without requiring any witness extraction (*i.e.* the proof of knowledge property). In the case of the DDH based protocols we employ in Section 3 for proving relations $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$, the first message of the protocol can be randomly chosen by the simulator. Hence, in order to simulate a proof, the simulator only needs to program the random oracle on a random point, setting the challenge to an arbitrary value that allows it to craft the last message to form an accepting transcript. Similarly to the case of random oracle based commitments in the restricted programmable and observable global random oracle model, the simulator fails if the environment queries the random oracle in the random point that must be programmed, which happens with negligible probability since the simulator chooses this point at random.

Notice that the NIZK for relation $\mathcal{R}_3$ is based on the protocol of [4], which requires a common reference string (CRS) that consists of random group elements such that the discrete logarithm of these elements in a given base is unknown. However, such a CRS can be generated by standard coin tossing based on UC-secure commitments in the restricted programmable and observable global random oracle model, which have been efficiently constructed in [9].

C Formalization of Secure Card Games

In this section we formalize the notion of a secure card game. Our intention is to capture a large variety of card games, therefore we opt to formalize it using a functionality $\mathcal{F}_{\mathsf{CG}}$ that has an embedded program GR that expresses the rules of the game under consideration. $\mathcal{F}_{\mathsf{CG}}$ offers procedures to check-in and check-out the players into the game, to create cards, to shuffle a specified set of cards, to open cards both to the public as well as privately to one player. Additionally, it offers a channel for GR to communicate with the players. It also offers a mechanism to financially compensate the honest players in case that some adversarial player misbehaves or aborts. GR mediates the execution of the game between the players and performs actions such as: (1) requesting the operations with cards; (2) determining who is the next player that should move and requesting an action from it; (3) processing the outcomes of the players actions and of the operations with cards; (4) updating the players' money balance and bet amounts; (5) invoking the check-out procedure to allow one player to leave the game; (6) requesting the compensation mechanism to be executed if some player does not respond in an appropriate way (the appropriate responses depend on the specifics rules of the game, which are known to the players as well). Later on, the game program GR will also be used to parametrize the protocol, where it will inform the players which card operations have to be executed for each step of the game. Notice that we choose to only capture basic card operations and basic game financial transactions in $\mathcal{F}_{\mathsf{CG}}$. However, it can be easily combined with other functionalities that capture further operations that can come in handy in different games and that can be UC realized with cryptographic protocols. In this case, GR can also request the players to perform operations related to the extra functionalities aggregated into $\mathcal{F}_{\mathsf{CG}}$. A prime example of such an extension to $\mathcal{F}_{\mathsf{CG}}$ is adding a coin tossing functionality (easily and efficiently UC-realized in the random oracle model), which can provide players with a common source of uniform randomness for actions such as choosing the order in which players act at random. The functionality $\mathcal{F}_{\mathsf{CG}}$ that captures secure card games is described below:

D Security Analysis

We analyze the security of π_{CG} in the UC framework by constructing a simulator such that any environment has a negligible chance to distinguish between an ideal world execution with the simulator and $\mathcal{F}_{\mathsf{CG}}$ and a real world execution of π_{CG} with an adversary. The main idea behind our simulator is to replace all ciphertexts representing cards by ciphertexts containing a known plaintext message. It does that by using the simulator of $\mathsf{NIZK}_{\mathcal{R}_3}$ to generate a simulated proof of correct shuffle given arbitrary ciphertexts (on known messages) without being caught by the other parties. In the private and public card opening procedures, when the simulator learns the value of the opened card from $\mathcal{F}_{\mathsf{CG}}$, it uses the decryption share simulation algorithm $\mathsf{SimShareDec}$ along with the simulator of $\mathsf{NIZK}_{\mathcal{R}_2}$ to generate a decryption share such that the ciphertext representing that card is decrypted to the intended card value. The security of Protocol π_{CG} is formally captured by the following theorem:

Theorem 1. *Let* RTE *be a secure re-randomizable threshold public-key encryption scheme as defined in Appendix A.1. Let* $\mathsf{NIZK}_{\mathcal{R}_i} = (\mathsf{Prov}, \mathsf{Verify}, \mathsf{Sim}, \mathsf{Ext})$ *be a NIZK Proof of Membership scheme for* $i \in \{1, 2, 3\}$*, and the relation* $\mathcal{R}_i$ *as defined in Appendix A.2. For every static active adversary* $\mathcal{A}$ *who corrupts at most* $n - 1$ *parties, there exists a simulator* $\mathcal{S}$ *such that, for every environment* $\mathcal{Z}$*, the following relation holds:*

Functionality $\mathcal{F}_{\mathsf{CG}}$ (First Part)

The functionality is executed with players $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and is parameterized by a timeout limit τ, and the values of the initial stake t, the security deposit d and of the compensation q. There is an embedded program GR that represents the rules of the game and is responsible for mediating the execution: it requests actions from the players, processes their answers, and can invoke the procedures of $\mathcal{F}_{\mathsf{CG}}$. $\mathcal{F}_{\mathsf{CG}}$ provides a check-in procedure that is run in the beginning of the execution, a check-out procedure that allows a player to leave the game (which is requested by the player via GR) and a compensation procedure that is invoked by GR if some player misbehaves/aborts. It also provides a channel for GR to request public actions from the players and card operations as described below. GR is also responsible for updating the vectors $\mathsf{balance}$ and bets. Whenever a message is sent to $\mathcal{S}$ for confirmation or action selection, $\mathcal{S}$ should answer, but can always answers $(\textsc{abort}, sid)$, in which case the compensation procedure is executed; this option will not be explicitly mentioned in the functionality description henceforth.

- **Check-in:** Executed during the initialization, it waits for a check-in message $(\textsc{checkin}, sid, \mathsf{coins}(d+t))$ from each $\mathcal{P}_i$ and sends $(\textsc{checkedin}, sid, \mathcal{P}_i)$ to the remaining players and GR. If some player fails to check-in within the timeout limit τ, then allow the players that checked-in to dropout and reclaim their coins. Initialize a balance vector as $\mathsf{balance} = (t, \ldots, t)$ and a bets vector as $\mathsf{bets} = (0, \ldots, 0)$.
- **Check-out:** Whenever GR requests the check-out of the players with payouts specified by the vector payout, send $(\textsc{checkout}, sid, \mathsf{payout})$ to $\mathcal{S}$. If $\mathcal{S}$ answers $(\textsc{checkout}, sid, \mathsf{payout})$, send the message $(\textsc{payout}, sid, \mathcal{P}_i, \mathsf{coins}(d + \mathsf{payout}[i]))$ to each $\mathcal{P}_i$ and stop the execution.
- **Compensation:** This procedure is triggered whenever $\mathcal{S}$ answers a request for confirmation of an action with $(\textsc{abort}, sid)$. For each active honest player $\mathcal{P}_i$, send him $(\textsc{compensation}, sid, \mathsf{coins}(d+q+\mathsf{balance}[i]+\mathsf{bets}[i]))$. Send the remaining locked coins to $\mathcal{S}$ and stop the execution.
- **Request Action:** Whenever GR requests an action with description $act - desc$ from $\mathcal{P}_i$, send a message $(\textsc{action}, sid, \mathcal{P}_i, act - desc)$ to the players. Upon receiving an answer $(\textsc{action-rsp}, sid, \mathcal{P}_i, act - rsp)$ from $\mathcal{P}_i$, forward it to all other players and GR.
- **Create Card:** Whenever GR requests the creation of a card with value v, choose a new identifier id, store the card (id, v) and send the message $(\textsc{newcard}, sid, \mathsf{id}, v)$ to all players and GR.
- **Shuffle Cards:** Whenever GR requests the cards with identifiers $(\mathsf{id}_1, \ldots, \mathsf{id}_m)$ to be shuffled, send the message $(\textsc{shuffle}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to $\mathcal{S}$. If $\mathcal{S}$ answers $(\textsc{shuffle}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$, a random permutation Π is applied to the corresponding values $(v_1, \ldots, v_m)$ to obtain the updated cards $(\mathsf{id}_1, v_1'), \ldots, (\mathsf{id}_m, v_m')$ such that $(v_1', \ldots, v_m') = \Pi(v_1, \ldots, v_m)$. Send $(\textsc{shuffled}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to all players and GR.
- **Open Private Card:** Whenever GR requests to reveal the card with identifier id to $\mathcal{P}_i$, send the message $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id})$ to $\mathcal{S}$. If $\mathcal{S}$ answers $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id})$, read (id, v) from the memory and send the message $(\textsc{card}, sid, \mathsf{id}, v)$ to $\mathcal{P}_i$.
- **Open Public Card:** Whenever GR requests to reveal the card (id, v) in public, read the card (id, v) from the memory and send the message $(\textsc{card}, sid, \mathsf{id}, v)$ to $\mathcal{S}$. If $\mathcal{S}$ answers $(\textsc{card}, sid, \mathsf{id}, v)$, forward this message to all players and GR.

Fig. 7. Functionality $\mathcal{F}_{\mathsf{CG}}$ (First Part).

$$\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{CG}},\mathcal{S},\mathcal{Z}} \approx_c \mathsf{HYBRID}^{\mathcal{G}_{\mathsf{rpoRO}},\mathcal{F}_{\mathsf{DSIG}},\mathcal{F}_{\mathsf{SC}}}_{\pi_{\mathsf{CG}},\mathcal{A},\mathcal{Z}}.$$

Functionality $\mathcal{F}_{\mathsf{CG}}$ (Second Part)

- **Shuffle Private Cards:** Whenever GR requests a set of private cards $(\mathsf{id}_1, v_1), \ldots, (\mathsf{id}_m, v_m)$ from player $\mathcal{P}_i$ to be shuffled, send the message $(\textsc{private-shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to $\mathcal{S}$. If $\mathcal{S}$ answers $(\textsc{private-shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m)$, a random permutation $(v'_1, \ldots, v'_m) = \Pi(v_1, \ldots, v_m)$ is applied to the values $(v_1, \ldots, v_m)$ to obtain the updated cards $(\mathsf{id}_1, v'_m), \ldots, (\mathsf{id}_m, v'_m)$. Send $(\textsc{private-shuffled}, sid, (\mathsf{id}_1, v'_1), \ldots, (\mathsf{id}_m, v'_m))$ to $\mathcal{P}_i$ and the message $(\textsc{private-shuffled}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to the other players and GR.

Fig. 8. Functionality $\mathcal{F}_{\mathsf{CG}}$ (Second Part).

Proof. In order to prove the security of π_{CG}, we construct a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish between interactions with an adversary $\mathcal{A}$ in the real world and with $\mathcal{S}$ in the ideal world. $\mathcal{S}$ writes all the messages received from $\mathcal{Z}$ in $\mathcal{A}$'s input tape, simulating $\mathcal{A}$'s environment. Also, $\mathcal{S}$ writes all messages from $\mathcal{A}$'s output tape to its own output tape, forwarding them to $\mathcal{Z}$.

We now describe the simulator $\mathcal{S}$. Let $\mathcal{P}_h$ denote one of honest parties (any arbitrary one), which we have special procedures during the simulation. As in the protocol, the simulator $\mathcal{S}$ is parametrized by a security parameter 1^κ, RTE parameters $\mathsf{param} \leftarrow \mathsf{Setup}(1^\kappa)$, a timeout limit τ, the values of the initial stake t, the compensation q, the security deposit $d \geq (n-1)q$ and an embedded program GR that represents the rules of the game. $\mathcal{S}$ simulates an execution with an internal copy of the adversary $\mathcal{A}$ (that controls the malicious parties), and generates the protocol messages from the honest parties. $\mathcal{S}$ proceeds as follows to simulate each procedure of Protocol π_{CG}:

- **Simulating $\mathcal{F}_{\mathrm{RO}}$:** $\mathcal{S}$ simulates the answers to the random oracle queries from $\mathcal{A}$ exactly as $\mathcal{F}_{\mathrm{RO}}$ would (and stores the lists of queries/answers), except when stated otherwise in $\mathcal{S}$'s description.
- **Simulating $\mathcal{F}_{\mathsf{DSIG}}$:** $\mathcal{S}$ simulates queries from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{DSIG}}$ exactly as $\mathcal{F}_{\mathsf{DSIG}}$ would.
- **Simulating $\mathcal{F}_{\mathsf{SC}}$:** $\mathcal{S}$ simulates queries from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{SC}}$ exactly as $\mathcal{F}_{\mathsf{SC}}$ would.
- **Checkpoint Witnesses, Recovery Trigger, Tracking Balance and Bets, Executing Actions, Check-in, Create Card, Check-out, Compensation:** $\mathcal{S}$ simulates the execution of the respective procedures of π_{CG} for the honest parties. If the procedure finishes correctly in this internal simulation, then $\mathcal{S}$ forwards the necessary messages to $\mathcal{F}_{\mathsf{CG}}$ to let it continue.
- **Recovery:** When the recovery phase is activated, $\mathcal{S}$ proceeds by following the steps of an honest party in π_{CG} by sending its most up to date checkpoint and current procedure witnesses to the simulated $\mathcal{F}_{\mathsf{SC}}$ and proceeding by sending the next messages of the honestly simulated execution of π_{CG} to $\mathcal{F}_{\mathsf{SC}}$ instead of sending them directly to $\mathcal{A}$. However, when a card operation (shuffling or opening of cards) is required while execution is mediated by the simulated $\mathcal{F}_{\mathsf{SC}}$, the messages required by such operation are simulated as described below. If the recovery phase succeeds, $\mathcal{S}$ sends $\mathcal{F}_{\mathsf{CG}}$ a message allowing the execution to proceed normally and, if it fails, $\mathcal{S}$ sends a failure message to $\mathcal{F}_{\mathsf{CG}}$ notifying that the operation has failed as described below in the steps for simulating each operation. In case of a failure, $\mathcal{S}$ proceeds to simulate the compensation phase.
- **Shuffle Cards:** In this procedure, $\mathcal{S}$ will replace all ciphertexts representing cards (*i.e.* encrypting a card number) with ciphertexts encrypting 1, in such a way that $\mathcal{S}$ later knows the encrypted values and can generate decryption shares that result

in decrypting each ciphertext to an arbitrary value. Upon receiving the message $(\textsc{shuffle}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ from $\mathcal{F}_{\mathsf{CG}}$, $\mathcal{S}$ proceeds as follows to simulate each round of the shuffle cards procedure:

  - Each honest party define $\mathsf{ct}^0_{\mathsf{id}_1}, \ldots, \mathsf{ct}^0_{\mathsf{id}_m}$ using the procedures from π_{CG}.
  - Upon receiving a message $(\textsc{shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}^i_{\mathsf{id}_1}, \ldots, \mathsf{ct}^i_{\mathsf{id}_m}, \pi^{i-1}_{\mathcal{R}_3})$ on behalf of $\mathcal{P}_i$ from $\mathcal{A}$, $\mathcal{S}$ verifies that $\pi^i_{\mathcal{R}_3}$ is valid using the steps of Protocol π_{CG} that the honest parties would use. If this check fails or the message from the next player that is supposed to shuffle is not received before the timeout limit τ, $\mathcal{S}$ proceeds to the recovery procedure. If the recovery procedure fails, $\mathcal{S}$ sends $(\textsc{abort}, sid)$ to $\mathcal{F}_{\mathsf{CG}}$ and proceeds to the compensation procedure. If the recovery procedure succeeds, proceed with the simulation.
  - If it is the turn of an honest party (except $\mathcal{P}_h$) to shuffle, apart from performing the checks of the previous step, $\mathcal{S}$ simulates the shuffle procedure of π_{CG} and sends the resulting messages to $\mathcal{A}$.
  - If it is $\mathcal{P}_h$'s turn to shuffle, apart from performing the checks of the first step, $\mathcal{S}$ computes ciphertexts $\mathsf{ct}^h_{\mathsf{id}_j} \leftarrow \mathsf{Enc}(\mathsf{pk}, 1)$, for $j = 1, \ldots, m$ (with fresh randomness for each ciphertext). Next, $\mathcal{S}$ generates a proof of shuffle correctness $\pi^h_{\mathcal{R}_3}$ by executing $\mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Sim}(\mathsf{pk}, (\mathsf{ct}^{h-1}_{\mathsf{id}_1}, \ldots, \mathsf{ct}^{h-1}_{\mathsf{id}_m}), (\mathsf{ct}^h_{\mathsf{id}_1}, \ldots, \mathsf{ct}^h_{\mathsf{id}_m}))$ and sends $(\textsc{shuffle}, sid, \mathcal{P}_h, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}^h_{\mathsf{id}_1}, \ldots, \mathsf{ct}^h_{\mathsf{id}_m}, \pi^h_{\mathcal{R}_3})$ to $\mathcal{A}$.

  If all rounds of the shuffle cards procedure are successfully completed, $\mathcal{S}$ sends $(\textsc{shuffle}, sid, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to $\mathcal{F}_{\mathsf{CG}}$.

– **Shuffle Private Cards:** In this procedure, $\mathcal{S}$ relays corrupted players' private shuffling requests to $\mathcal{F}_{\mathsf{CG}}$ and simulates the honest party's private shuffling requests. Upon receiving from $\mathcal{F}_{\mathsf{CG}}$ $(\textsc{private-shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m)$, $\mathcal{S}$ proceeds as follows:

  - If $\mathcal{P}_i$ is corrupted, $\mathcal{S}$ waits for a message $(\textsc{privshuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}'_{\mathsf{id}_1}, \ldots, \mathsf{ct}'_{\mathsf{id}_m}, \pi^i_{\mathcal{R}_3})$ from $\mathcal{P}^{\mathcal{A}}_i$ and verifies $\pi^i_{\mathcal{R}_3}$ by checking the verification procedure $\mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Verify}((\mathsf{pk}, (\mathsf{ct}_{\mathsf{id}_1}, \ldots, \mathsf{ct}_{\mathsf{id}_m}), (\mathsf{ct}'_{\mathsf{id}_1}, \ldots, \mathsf{ct}'_{\mathsf{id}_m})), \pi^i_{\mathcal{R}_3}) = 1$, where the tuple $(\mathsf{ct}_{\mathsf{id}_1}, \ldots, \mathsf{ct}_{\mathsf{id}_m})$ are the corresponding ciphertexts before the private shuffle. If this check succeeds, $\mathcal{S}$ sends the message $(\textsc{private-shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to $\mathcal{F}_{\mathsf{CG}}$. If this check fails or the message is not received before the timeout limit τ, $\mathcal{S}$ proceeds to the recovery procedure. If the recovery procedure fails, $\mathcal{S}$ proceeds to the compensation procedure after sending $(\textsc{abort}, sid)$ to $\mathcal{F}_{\mathsf{CG}}$. If the recovery procedure succeeds, $\mathcal{S}$ sends $(\textsc{private-shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m)$ to $\mathcal{F}_{\mathsf{CG}}$.
  - If $\mathcal{P}_i$ is honest, $\mathcal{S}$ simulates the honest steps of π_{CG} for doing a private shuffle and sends $(\textsc{privshuffle}, sid, \mathcal{P}_h, \mathsf{id}_1, \ldots, \mathsf{id}_m, \mathsf{ct}'_{\mathsf{id}_1}, \ldots, \mathsf{ct}'_{\mathsf{id}_m}, \pi^h_{\mathcal{R}_3})$ to $\mathcal{A}$. Next, $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{CG}}$ the message $(\textsc{private-shuffle}, sid, \mathcal{P}_i, \mathsf{id}_1, \ldots, \mathsf{id}_m)$.

– **Open Private Card:** Upon receiving a message $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id})$ from $\mathcal{F}_{\mathsf{CG}}$, $\mathcal{S}$ proceeds as follows:

  - If $\mathcal{P}_i$ is corrupted, $\mathcal{S}$ sends $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id})$ to $\mathcal{F}_{\mathsf{CG}}$, obtaining $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id}, v)$ as answer. Next, $\mathcal{S}$ simulates an opening of the private card with id represented by ciphertext $\mathsf{ct}_{\mathsf{id}}$ towards party $\mathcal{P}_i$ in such a way that decrypting $\mathsf{ct}_{\mathsf{id}}$ results in v. For all honest parties other than $\mathcal{P}_h$, $\mathcal{S}$ generates the decryption shares honestly using the procedures of π_{CG} and send the resulting messages to $\mathcal{A}$. Additionally, $\mathcal{S}$

generates the decryption share of $\mathcal{P}_h$, d_h, using $\mathsf{SimShareDec}$ such that decrypting $\mathsf{ct}_{\mathsf{id}}$ yields v. Next, $\mathcal{S}$ generates a simulated NIZK $\pi^h_{\mathcal{R}_2}$ showing that the decryption share d_h was correctly computed by executing $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Sim}(\mathsf{pk}_h, \mathsf{ct}_{\mathsf{id}}, \mathsf{d}_h)$. These simulated decryption share d_h and proof $\pi^h_{\mathcal{R}_2}$ are recorded and reused if the same ciphertext is opened again. $\mathcal{S}$ sends $(\textsc{opencard}, sid, \mathcal{P}_h, \mathsf{id}, \mathsf{d}_h, \pi^h_{\mathcal{R}_2})$ to $\mathcal{A}$.

- If $\mathcal{P}_i$ is honest, $\mathcal{S}$ waits for messages $(\textsc{opencard}, sid, \mathcal{P}_j, \mathsf{id}, \mathsf{d}_j, \pi^j_{\mathcal{R}_2})$ from each malicious party $\mathcal{P}_j$ controlled by $\mathcal{A}$. Upon receiving each of these messages, $\mathcal{S}$ verifies $\pi^j_{\mathcal{R}_2}$ by checking that $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Verify}((\mathsf{pk}_j, \mathsf{ct}_{\mathsf{id}}, \mathsf{d}_j), \pi^j_{\mathcal{R}_2}) = 1$. If all of the messages are received and all NIZKs $\pi^j_{\mathcal{R}_2}$ are valid, $\mathcal{S}$ sends $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id})$ to $\mathcal{F}_{\mathsf{CG}}$. If any of these messages is not received before the timeout limit τ or any of the NIZKs $\pi^j_{\mathcal{R}_2}$ are invalid, $\mathcal{S}$ proceeds to the recovery procedure. If the recovery procedure succeeds, $\mathcal{S}$ sends $(\textsc{card}, sid, \mathcal{P}_i, \mathsf{id})$ to $\mathcal{F}_{\mathsf{CG}}$. Otherwise, $\mathcal{S}$ sends $(\textsc{abort}, sid)$ to $\mathcal{F}_{\mathsf{CG}}$ and proceeds to the compensation procedure.

– **Open Public Card** Upon receiving a message $(\textsc{card}, sid, \mathsf{id}, v)$ from $\mathcal{F}_{\mathsf{CG}}$, $\mathcal{S}$ simulates an opening of the public card with id represented by ciphertext $\mathsf{ct}_{\mathsf{id}}$ towards $\mathcal{A}$ in such a way that v is obtained, proceeding as follows:

- For all honest parties other than $\mathcal{P}_h$, $\mathcal{S}$ generates the decryption shares honestly using the procedures of π_{CG} and send the resulting messages to $\mathcal{A}$. Additionally, $\mathcal{S}$ generates the decryption share of $\mathcal{P}_h$, d_h, using $\mathsf{SimShareDec}$ such that decrypting $\mathsf{ct}_{\mathsf{id}}$ yields v. Next, $\mathcal{S}$ generates a simulated NIZK $\pi^h_{\mathcal{R}_2}$ showing that the decryption share d_h was correctly computed by executing $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Sim}(\mathsf{pk}_h, \mathsf{ct}_{\mathsf{id}}, \mathsf{d}_h)$. These simulated decryption share d_h and proof $\pi^h_{\mathcal{R}_2}$ are recorded and reused if the same ciphertext is opened again. $\mathcal{S}$ sends $(\textsc{opencard}, sid, \mathcal{P}_h, \mathsf{id}, \mathsf{d}_h, \pi^h_{\mathcal{R}_2})$ to $\mathcal{A}$.
- $\mathcal{S}$ waits for messages $(\textsc{opencard}, sid, \mathcal{P}_j, \mathsf{id}, \mathsf{d}_j, \pi^j_{\mathcal{R}_2})$ from each malicious party $\mathcal{P}_j$ controlled by $\mathcal{A}$. Upon receiving these messages, $\mathcal{S}$ verifies $\pi^j_{\mathcal{R}_2}$ by checking that $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Verify}((\mathsf{pk}_j, \mathsf{ct}_{\mathsf{id}}, \mathsf{d}_j), \pi^j_{\mathcal{R}_2}) = 1$. If all of the messages are received and all NIZKs $\pi^j_{\mathcal{R}_2}$ are valid, $\mathcal{S}$ sends $(\textsc{card}, sid, \mathsf{id}, v)$ to $\mathcal{F}_{\mathsf{CG}}$. If any of these messages is not received before the timeout limit τ or any of the NIZKs $\pi^j_{\mathcal{R}_2}$ are invalid, $\mathcal{S}$ proceeds to the recovery procedure. If the recovery procedure succeeds, $\mathcal{S}$ sends $(\textsc{card}, sid, \mathsf{id}, v)$ to $\mathcal{F}_{\mathsf{CG}}$. Otherwise, $\mathcal{S}$ sends $(\textsc{abort}, sid)$ to $\mathcal{F}_{\mathsf{CG}}$ and proceeds to the compensation procedure.

Simulation Indistinguishability: Notice that the simulator $\mathcal{S}$ only deviates from a real execution of Protocol π_{CG} in the shuffle cards, shuffle private cards, open private card and open public card procedures. In the case of the shuffle procedure, $\mathcal{S}$ acts exactly as an honest party in a real execution of π_{CG} except for when $\mathcal{P}_h$ is performing a shuffling operation (*i.e.* rerandomizing the ciphertexts representing cards, permuting them and generating proofs of shuffle correctness). In this case, $\mathcal{S}$ deviates from protocol execution by replacing the ciphertexts it receives from the previous party (or $\mathcal{A}$) by arbitrary ciphertexts containing the message 1 and generating a proof of shuffle correctness $\pi^h_{\mathcal{R}_3}$ by running $\mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Sim}(\mathsf{pk}, (\mathsf{ct}^{h-1}_{\mathsf{id}_1}, \dots, \mathsf{ct}^{h-1}_{\mathsf{id}_m}), (\mathsf{ct}^h_{\mathsf{id}_1}, \dots, \mathsf{ct}^h_{\mathsf{id}_m}))$. If the environment $\mathcal{Z}$ can distinguish an execution with ciphertexts $\mathsf{ct}^h_{\mathsf{id}_i} \leftarrow \mathsf{Enc}(\mathsf{pk}, 1)$ from an execution with the ciphertexts generated by an honest execution of π_{CG}, it can be used to break the unlinkability or IND-CPA security properties of RTE, since these ciphertexts are indistinguishable given these two properties. If $\mathcal{Z}$ can distinguish an execution with the proof of shuffle correctness $\pi^h_{\mathcal{R}_3}$ obtained by running $\mathsf{NIZK}_{\mathcal{R}_3}.\mathsf{Sim}(\mathsf{pk}, (\mathsf{ct}^{h-1}_{\mathsf{id}_1}, \ldots, \mathsf{ct}^{h-1}_{\mathsf{id}_m}), (\mathsf{ct}^{h}_{\mathsf{id}_1}, \ldots, \mathsf{ct}^{h}_{\mathsf{id}_m}))$ from an execution with the proof of shuffle correctness obtained by honestly running π_{CG}, it can be used to break the zero-knowledge property of $\mathsf{NIZK}_{\mathcal{R}_3}$. In the case of the shuffle private cards procedure, $\mathcal{S}$ instructs $\mathcal{F}_{\mathsf{CG}}$ to abort execution in case the adversary $\mathcal{A}$ provided an invalid message, which also causes an abort in the real execution of π_{CG}. In the cases of the open public card and open private card procedures, $\mathcal{S}$ deviates from the protocol by computing an arbitrary decryption share d_h using $\mathsf{SimShareDec}$ such that decrypting $\mathsf{ct}_{\mathsf{id}}$ yields v (obtained from $\mathcal{F}_{\mathsf{CG}}$) and generating a simulated NIZK $\pi^h_{\mathcal{R}_2}$ showing that the decryption share was correctly computed by running $\mathsf{NIZK}_{\mathcal{R}_2}.\mathsf{Sim}(\mathsf{pk}_h, \mathsf{ct}_{\mathsf{id}}, \mathsf{d}_h)$. If the environment distinguishes an execution with the simulated decryption share d_h from an honest execution of π_{CG}, it can be used to break the share-simualtion indistinguishability of RTE. If $\mathcal{Z}$ distinguishes an execution with the simulated NIZK $\pi^h_{\mathcal{R}_2}$ from an honest execution of π_{CG}, it can be used to break the zero-knowledge property of $\mathsf{NIZK}_{\mathcal{R}_3}$. Hence, the ideal execution with $\mathcal{S}$ is indistinguishable from a real execution of π_{CG} as long as RTE is a secure re-randomizable threshold public-key encryption scheme as defined in Appendix A.1 and, for $i \in \{1, 2, 3\}$, $\mathsf{NIZK}_{\mathcal{R}_i} = (\mathsf{Prov}, \mathsf{Verify}, \mathsf{Sim}, \mathsf{Ext})$ is a NIZK proof of membership scheme for the relation $\mathcal{R}_i$ as defined in Appendix A.2.

E Unidentifiable Abort Situation on Wei's Poker Protocol

We now present a situation which can be provoked by a user while performing the poker protocol of Wei [32] that can make the protocol abort during the Shuffle phase without allowing the detection of the misbehaving party. Before detailing the situation, we review the whole protocol and the two phases, namely the *Wrap-A* and *Reveal* phases, which concern the unidentifiable abort situation. Later we describe the situation itself and discuss its consequences.

Overview of the Protocol. Each player n, from a set of N players, selects a private key x_n via the [32, Protocol 2] and all N players share the joint private key $x_s = \prod_{n=1}^{N} x_n$. Each face down card is represented by a tuple $(a^c, b) = (a^c, a^{x_s})$, for a joint computed $a \in \mathbb{G}$ as in [32, Protocol 5]. Moreover c is the public known value of the card. Therefore a deck of M face down cards, which in [32] is named CR deck, is the set of M tuples $D = (a_1^{c_1}, a_1^{x_s}), \cdots, (a_M^{c_M}, a_M^{x_s})$.

The shuffle protocol receives as input the CR deck D, and a general description of the procedure is as follows:

1. Wrap-A: All the M cards are jointly wrapped, among the N players, in order to generate a wrapped deck.
2. Wrap-B: Similarly to the previous phase, N players encode each card with a different wrapping method.
3. Shuffle: The main shuffle procedure is executed by each player via a random permutation.

4. Reveal: All the players broadcast the secret information used during the first two phases.
5. Unwrapping: The players unwrap each card and verify the integrity of the shuffling.

We focus on the two phases that can be manipulated by a malicious player.

The Wrap-A and Reveal Phases. The misbehaving player incorrectly performs the Wrap-A phase. First, recall that during Wrap-A, Player n encodes each card (a_m, b_m) as

$$e_{m,n} = e_{1,n-1} \cdot a_m^{\lambda_{1,n}} \cdot b_m^{\lambda_{2,n}} \cdot g^{\lambda_{3,n}} \tag{1}$$

for randomly chosen values $\lambda_{1,n}, \lambda_{2,n}$ and $\lambda_{3,n}$ picked by Player n and initial value $e_{m,0} = 1_{\mathbb{G}}$ and $g \in \mathbb{G}$. Each Player n broadcasts $e_{1,n}, \dots e_{M,n}$, however it keeps the values $\lambda_{1,n}, \lambda_{2,n}$ and $\lambda_{3,n}$ secret. They will be revealed during the Reveal phase.

During the Reveal phase, when the values for $((\lambda_{1,1}, \lambda_{2,1}, \lambda_{3,1}), \cdots, (\lambda_{1,N}, \lambda_{2,N}, \lambda_{3,N}))$ are already known to all participants, therefore each player j will compute $e'_{m,j}$ and $e'_{m,N}$ with the revealed values. The verification of the values is executed by [32, Procedure 21] which can be summarized by the following two equations:

$$e_{m,j} = a_m^{\sum\limits_{n=1}^{j} \lambda_{1,n}} \cdot b_m^{\sum\limits_{n=1}^{j} \lambda_{2,n}} \cdot g^{\sum\limits_{n=1}^{j} \lambda_{3,n}}, \tag{2}$$

$$e_{m,N} = a_m^{\sum\limits_{n=1}^{N} \lambda_{1,n}} \cdot b_m^{\sum\limits_{n=1}^{N} \lambda_{2,n}} \cdot g^{\sum\limits_{n=1}^{N} \lambda_{3,n}}. \tag{3}$$

Equations 2 and 3 are interpreted, respectively, as the checking for the value received up until the turn of the j-th player, and the resulting value after all the players had executed the Wrap-A procedure, that is, up to the point of the N-th Player.

The Sketch of the Abort Situation. An arbitrary Player i, which decides to make another Player t to abort, can, during the Wrap-A phase, follow the protocol and generate $e_{m,i}$ values and broadcasts them with one small change in Equation 1. Instead of computing using the previous value $e_{m,i-1}$, it computes for its place $e_{m,0} \cdot e_{m,1} \dots e_{m,t-1} \cdot e_{m,t+1} \dots e_{m,i-1}$ instead. Purposely excluding the item $e_{m,t}$ of the term, which is feasible since all players broadcast their values $e_{m,n}$ on each turn. As a result, the Equation 3 fails which leads the protocol to be aborted. More important is that both equations do not reveal the identity of Player i.

Collusion and Penalties May Worsen the Situation. The constraint that the target Player t has to compute before the attacker Player i, *i.e.* $t < i$, can be generalized for the situation that all other players are colluding against Player t. This situation is in fact not an unlike scenario in online gambling. Take for example, the case that in a poker room all players are in fact controlled by a simple player, except one. In that case, the adversary can choose freely i.

In a scenario where penalties are applied, financial penalties in the sense of Andrychowicz *et al.* [2, 1], all colluding players could show that their computation are in fact correct by presenting make their values $\lambda_{1,n}, \lambda_{2,n}$ and $\lambda_{3,n}$. Without a clear method of verification other than the Equations 2 and 3 alone, the cheater cannot be identified.

Or potentially only the single honest player can the penalized, which ultimately would make the adversary, who controls the colluding players, collect the collateral of the honest player.

This situation illustrates the consequences of the lack of formalization. More concretely, despite the fact that the work by Bentov et al. [7] directly indicates the poker protocol of [32] as an example of a use case of their ingenious framework, it is clear that they do not properly integrate with each other. And the reason is that [7] does not investigate the necessary properties of that composition. Here, a required feature of the poker protocol is the ability of identifying the malicious adversary (so the penalties can be applied). A natural conjecture is that a more formal treatment of the composability and security of these protocols would have spotted this gap.

n m	2	4	6	8	10	12
52	220 $\mathbb{G}$, 80 $\mathbb{Z}_p$	440 $\mathbb{G}$, 160 $\mathbb{Z}_p$	660 $\mathbb{G}$, 240 $\mathbb{Z}_p$	880 $\mathbb{G}$, 320 $\mathbb{Z}_p$	1100 $\mathbb{G}$, 400 $\mathbb{Z}_p$	1320 $\mathbb{G}$, 480 $\mathbb{Z}_p$
104	430 $\mathbb{G}$, 110 $\mathbb{Z}_p$	860 $\mathbb{G}$, 220 $\mathbb{Z}_p$	1290 $\mathbb{G}$, 330 $\mathbb{Z}_p$	1720 $\mathbb{G}$, 440 $\mathbb{Z}_p$	2150 $\mathbb{G}$, 550 $\mathbb{Z}_p$	2580 $\mathbb{G}$, 660 $\mathbb{Z}_p$
156	640 $\mathbb{G}$, 130 $\mathbb{Z}_p$	1280 $\mathbb{G}$, 260 $\mathbb{Z}_p$	1920 $\mathbb{G}$, 390 $\mathbb{Z}_p$	2560 $\mathbb{G}$, 520 $\mathbb{Z}_p$	3200 $\mathbb{G}$, 650 $\mathbb{Z}_p$	3840 $\mathbb{G}$, 780 $\mathbb{Z}_p$
208	848 $\mathbb{G}$, 150 $\mathbb{Z}_p$	1696 $\mathbb{G}$, 300 $\mathbb{Z}_p$	2544 $\mathbb{G}$, 450 $\mathbb{Z}_p$	3392 $\mathbb{G}$, 600 $\mathbb{Z}_p$	4240 $\mathbb{G}$, 750 $\mathbb{Z}_p$	5088 $\mathbb{G}$, 900 $\mathbb{Z}_p$
260	1058 $\mathbb{G}$, 170 $\mathbb{Z}_p$	2116 $\mathbb{G}$, 340 $\mathbb{Z}_p$	3174 $\mathbb{G}$, 510 $\mathbb{Z}_p$	4232 $\mathbb{G}$, 680 $\mathbb{Z}_p$	5290 $\mathbb{G}$, 850 $\mathbb{Z}_p$	6348 $\mathbb{G}$, 1020 $\mathbb{Z}_p$
312	1266 $\mathbb{G}$, 180 $\mathbb{Z}_p$	2532 $\mathbb{G}$, 360 $\mathbb{Z}_p$	3798 $\mathbb{G}$, 540 $\mathbb{Z}_p$	5064 $\mathbb{G}$, 720 $\mathbb{Z}_p$	6330 $\mathbb{G}$, 900 $\mathbb{Z}_p$	7596 $\mathbb{G}$, 1080 $\mathbb{Z}_p$

Table 6. Concrete communication complexity of the Shuffle Cards phase of Royale for practical numbers of players (n) and cards (m).

n m	2	4	6	8	10	12
52	2028 $\mathbb{G}$, 140 $\mathbb{Z}_p$	3796 $\mathbb{G}$, 280 $\mathbb{Z}_p$	5564 $\mathbb{G}$, 420 $\mathbb{Z}_p$	7332 $\mathbb{G}$, 560 $\mathbb{Z}_p$	9100 $\mathbb{G}$, 700 $\mathbb{Z}_p$	10868 $\mathbb{G}$, 840 $\mathbb{Z}_p$
104	4056 $\mathbb{G}$, 244 $\mathbb{Z}_p$	7592 $\mathbb{G}$, 488 $\mathbb{Z}_p$	11128 $\mathbb{G}$, 732 $\mathbb{Z}_p$	14664 $\mathbb{G}$, 976 $\mathbb{Z}_p$	18200 $\mathbb{G}$, 1220 $\mathbb{Z}_p$	21736 $\mathbb{G}$, 1464 $\mathbb{Z}_p$
156	6084 $\mathbb{G}$, 348 $\mathbb{Z}_p$	11388 $\mathbb{G}$, 696 $\mathbb{Z}_p$	16692 $\mathbb{G}$, 1044 $\mathbb{Z}_p$	21996 $\mathbb{G}$, 1392 $\mathbb{Z}_p$	27300 $\mathbb{G}$, 1740 $\mathbb{Z}_p$	32604 $\mathbb{G}$, 2088 $\mathbb{Z}_p$
208	8112 $\mathbb{G}$, 452 $\mathbb{Z}_p$	15184 $\mathbb{G}$, 904 $\mathbb{Z}_p$	22256 $\mathbb{G}$, 1356 $\mathbb{Z}_p$	29328 $\mathbb{G}$, 1808 $\mathbb{Z}_p$	36400 $\mathbb{G}$, 2260 $\mathbb{Z}_p$	43472 $\mathbb{G}$, 2712 $\mathbb{Z}_p$
260	10140 $\mathbb{G}$, 556 $\mathbb{Z}_p$	18980 $\mathbb{G}$, 1112 $\mathbb{Z}_p$	27820 $\mathbb{G}$, 1668 $\mathbb{Z}_p$	36660 $\mathbb{G}$, 2224 $\mathbb{Z}_p$	45500 $\mathbb{G}$, 2780 $\mathbb{Z}_p$	54340 $\mathbb{G}$, 3336 $\mathbb{Z}_p$
312	12168 $\mathbb{G}$, 660 $\mathbb{Z}_p$	22776 $\mathbb{G}$, 1320 $\mathbb{Z}_p$	33384 $\mathbb{G}$, 1980 $\mathbb{Z}_p$	43992 $\mathbb{G}$, 2640 $\mathbb{Z}_p$	54600 $\mathbb{G}$, 3300 $\mathbb{Z}_p$	65208 $\mathbb{G}$, 3960 $\mathbb{Z}_p$

Table 7. Concrete communication complexity of the Shuffle Cards phase of Wei [32] for practical numbers of players (n) and cards (m).

m n	52	104	156	208	260	312
2	624	1456	2184	2912	4160	4992
4	1024	2288	3432	4576	6240	7488
6	1456	3120	4680	6240	8320	9984
8	1872	3952	5928	7904	10400	12480
10	2288	4784	7176	9568	12480	14976
12	2704	5616	8424	11232	14560	17472

Table 8. Concrete computational complexity of the Shuffle Cards phase of Royale in terms of modular exponentiations for practical numbers of players (n) and cards (m).

m n	52	104	156	208	260	312
2	4241	8453	12665	16877	21089	25301
4	4245	8457	12669	16881	21093	25305
6	4249	8461	12673	16885	21097	25309
8	4253	8465	12677	16889	21101	25313
10	4257	8469	12681	16893	21105	25317
12	4261	8473	12685	16897	21109	25321

Table 9. Concrete computational complexity of the Shuffle Cards phase of Wei [32] in terms of modular exponentiation for practical numbers of players (n) and cards (m).

F Further Efficiency Analysis

F.1 Concrete Complexity of Royale for Practical Parameters

Communication Complexity Estimates of the total number of elements of $\mathbb{G}$ and of $\mathbb{Z}_p$ exchanged between all players in the Shuffle Cards phase of Royale with numbers of players and of cards commonly found in practical applications (*i.e.* popular games) are presented in Table 6, while the protocol of Wei [32] is considered in Table 7.

Computational Complexity Estimates of the number of modular exponentiations required of each player in the Shuffle Cards phase of Royale with numbers of users and of cards commonly found in practical applications (*i.e.* popular games) are presented in Table 8, while the same estimates for the protocol of Wei [32] are presented in Table 9.

F.2 Benchmark data for Open Public Card, Open Private Card and Shuffle Private Cards Phases

The execution time and network communication for the Shuffle Private Cards phase are presented in Table 13 and Table 10, respectively. The execution time is presented as the sum of the local computation time required of each player and the network Round Trip Time necessary for delivering this phase's messages. Differently from the Shuffle Cards procedure, player $\mathcal{P}_i$ only check the shuffle generated by $\mathcal{P}_j$ and broadcast their signatures on the checkpoint witness, later verifying all signatures.

m n	52	104	208
2	6.93 KB	12.31 KB	20.43 KB
4	7.05 KB	12.43 KB	20.55 KB
6	7.18 KB	12.56 KB	20.68 KB
8	7.3 KB	12.68 KB	20.8 KB
10	7.43 KB	12.81 KB	20.93 KB
12	7.55 KB	12.93 KB	21.05 KB

Table 10. Network communication in the Shuffle Private Cards phase.

n	Time	Communication
2	0.59 ms + 1 RTT	0.57 KB
4	1.00 ms + 1 RTT	1.89 KB
6	1.41 ms + 1 RTT	3.98 KB
8	1.82 ms + 1 RTT	6.82 KB
10	2.28 ms + 1 RTT	10.42 KB
12	2.64 ms + 1 RTT	14.78 KB

Table 11. Open Public Card Phase execution time.

n	Drawer - Time	Others - Time	Communication
2	0.39 ms + 1 RTT	0.18 ms + 1 RTT	0.38 KB
4	0.8 ms + 1 RTT	0.18 ms + 1 RTT	1.52 KB
6	1.21 ms + 1 RTT	0.18 ms + 1 RTT	3.41 KB
8	1.62 ms + 1 RTT	0.18 ms + 1 RTT	6.06 KB
10	2.05 ms + 1 RTT	0.18 ms + 1 RTT	9.47 KB
12	2.44 ms + 1 RTT	0.18 ms + 1 RTT	13.64 KB

Table 12. Open Private Card Phase execution time.

52	75.36 ms + 1 RTT	24.96 ms + 1 RTT
104	150.77 ms + 1 RTT	43.06 ms + 1 RTT
208	363.44 ms + 1 RTT	79.72 ms + 1 RTT

Table 13. Execution time for the Shuffle Private Cards phase.

The execution time and network communication for the Open Public Card and Open Private Card phases (opening one card) are presented respectively in Table 11 and 12.