How to Use Bitcoin to Play Internet Poker

Ranjit Kumaresan
talm@idc.ac.il
iddo@cs.technion.ac.il

Abstract—We present practical protocols for playing poker over the internet without having to trust a third party. Our poker protocols are dropout-tolerant in the sense that any party that drops out during the middle of the game is forced to pay a monetary penalty to all other parties.

More generally, we introduce and design protocols for secure cash distribution with penalties, a primitive that simultaneously generalizes secure computation with penalties of Bentov and Kumaresan (Crypto 2014), and secure lottery with penalties of Andrychowicz et al. (Security and Privacy 2014). Since secure cash distribution with penalties is a reactive functionality by design, it suffices to capture a wide variety of card games (including poker), board games, and auctions. Our protocol realizing secure cash distribution with penalties works in a hybrid model where parties have access to a claim-or-refund transaction functionality $\mathcal{FCR}$ which can be efficiently realized in (a variant of) Bitcoin.

Keywords: Poker, Bitcoin, Secure Computation.

I. INTRODUCTION

Once there were two “mental chess” experts who had become tired of their favorite pastime. Let’s play “mental poker” for some variety suggested one. “Sure” said the other. “Just let me deal!”

Motivated by this anecdote, Shamir, Rivest, and Adleman set forth in their seminal paper [1] to propose protocols that allow a pair of parties to play “fair” mental poker. Arguably their paper gave birth to the concept of secure multiparty computation [2], a primitive that allows a set of mutually distrusting parties to carry out a distributed computation without compromising on the privacy of inputs or the correctness of the end result. Indeed mental poker has since been used as a metaphor for secure multiparty computation [3]. Clearly, secure computation can be used to allow a set of parties to play poker over the internet without having to trust a third party. However this comes with certain caveats.

Today poker over the internet (more commonly known as online poker) is estimated to be a multi-billion dollar industry [4], [5], [6]. However, all of this traffic is handled by online poker websites that host games for its subscribers to play. In such settings, the players completely trust the poker website to conduct the game properly, e.g., that a perfectly random shuffling of cards has been done, the hole cards are kept private, etc. Now one might wonder why players need to trust a third party (i.e., the poker website) if they may very well use the tool of secure computation to play poker. Well, the obvious problem is that secure computation as defined can only allow players to play mental poker. One might say that poker is not poker unless it involves real money, so this is indeed a major problem. Another serious problem is that secure computation in the presence of a dishonest majority (including the important two-party case) does not provide dropout-tolerant solutions to mental poker. That is, a player who obtains hole cards that are not particularly good can simply drop out of the hand, and perhaps rejoin at a later time. Even worse, players may wait until the end of the hand to decide whether they want to drop out, i.e., after they have a much better idea of whether they are going to win or lose. These and other issues (which we discuss later) are the reasons why players trust a third party to play a game of poker over internet. However, trusting a third party does not come without a cost, and the fear, uncertainty, and doubt amongst players who play online poker is not without proper justification.

Risks in trusting a third party. Any security system is only as strong as the weakest link, and in the case of an online poker website the weakest link is perhaps “the human element,” e.g., disgruntled or greedy employees who have access to players’ hole cards during the game, or insert software backdoors in order to simply loot money from the players. Prominent poker websites such as Absolute Poker and Ultimate Bet have had (at least claim to have had) their integrity breached in this fashion [7], [8], [9], [10], [11]. Note that when the incidents happened, these websites were amongst the world’s ten largest online cardrooms. That is, the issue here is not about trusting a website whose reputation has not yet been fully established, but that no matter how reputable the online website is, it is always susceptible to such insider attacks.

Another major issue pertains to the legal status of online poker companies. In order to process transfers to and from their customers, poker companies may often have to resort to means whose legal status is either unclear or downright illegal. See [12], [13] for details on US government indictments against the founders of the three largest online poker companies, PokerStars, Full Tilt Poker and Cereus, and a handful of their associates.
Thus these companies seem to always be on the verge of getting “caught,” and when they do (which often happens without warning), their customers either simply lose all the money in their online poker accounts (which may sometimes be millions of dollars in a single account), or may have to travel overseas to claim the money in their accounts (cf. Everleaf scandal [13]). In summary, one never knows when online poker companies may fall prey to either the human element or government control. Thus, customers of online poker companies are gambling with their money in more than one way!

**Tying payments to secure computation.** Coming back to secure computation, we see that it helps in removing the human element, i.e., there is no need to trust a third party to order to carry out the computation or protect secret information, such as dealt cards. Unfortunately, as mentioned earlier, handling payments (as well as dropout-tolerance) falls outside its scope. Using a bank to handle payments in a secure computation protocol may potentially bring both the human element as well as government control back into the picture. On the other hand, as suggested in [15], [16], using a digital currency or cryptocurrency to handle payments may be a very attractive option. This is especially true for Bitcoin [17] which has gained popularity in recent times. Although the attitude of governments towards Bitcoin is mixed [18], [19], [20], [21], [22], [23], it is safe to say that it is more popular than online poker companies (at least with some governments).

Also, there are several websites that allow players to play using Bitcoins e.g., Gridpoker, SealsWithClubs, PrimeDice, 777coin. The popularity of these websites stems from the fact that there is no restriction on who can play and further there is no need for a payment processor which dramatically increases the speed of cashouts. Thus, it seems Bitcoin is a natural choice to handle payments to parties. We also note that several prior works [24], [25], [26], [27] have used Bitcoin to tie payments to secure computation. Some of the key features that allow this are Bitcoin scripts and timelocks—both of which will be extensively used in this paper.

**Dropout tolerance.** There are two kinds of dropouts: intentional and accidental. In this work, we treat all dropouts as intentional. Thus, we assume that a player that drops out does so because their cards aren’t good enough to win the pot. In such a situation we want to penalize the player that drops out. More precisely, every other player will get a compensation amount from the player that dropped out of the game during a hand (i.e., sitting out of an entire hand is not penalized). Note that the above is somewhat similar to the penalty model used to enforce fairness in standard secure computation [28], [24], [26], [27]. Prior to our work, the protocols designed for poker only ensured a very weak form of dropout tolerance, namely they support restarting the hand e.g., by reshuffling the (remaining) cards, and more importantly the aborting party does not pay any penalty [29], [30]. Such a weak notion does not quite make the cut for us since such a protocol incentivizes dishonest behavior: a player who drops out whenever the chances of winning are low will have an advantage over an honest player unless dropouts are penalized.

**Our contributions.** We present practical protocols for playing poker over the internet without having to trust a third party while being dropout-tolerant in the sense described above.

**Generalizing secure computation with money.** More generally, we introduce and design protocols for secure cash distribution with penalties, a primitive that simultaneously generalizes secure computation with penalties [26], [25] and secure lottery with penalties [26], [24]. We define secure cash distribution with penalties as a reactive functionality. More precisely, the computation proceeds in stages, where in each stage parties provide inputs and obtain outputs, and secret state can be maintained between stages. A party that aborts between stages will be penalized. Indeed this is one of the main differences from secure computation with penalties (which is limited to a single stage computation). In the last stage, the money deposited by the parties is redistributed among them according to the output of the last stage. See Section [11] for more details.

Since secure cash distribution with penalties is a reactive functionality by design, a wide variety of card games (including poker), board games, and auctions can be realized with or without money via secure cash distribution with penalties.

**Generic protocols for secure cash distribution.** Our protocol realizing secure cash distribution with penalties (i.e., with full simulation security [32]) works in a hybrid model where parties have access to a claim-or-refund transaction functionality $\mathcal{F}_{\text{CR}}^*$ which can be efficiently realized in Bitcoin (or a variant of Bitcoin, see next). The main technical idea in the solution is the construction of the see-saw transaction mechanism which is a novel extension of the ladder transaction mechanism of [26]. Modifying the ladder mechanism (used in [26] to realize secure computation with penalties) to realize secure cash distribution turns out to be somewhat nontrivial. We present our basic design of the protocol for secure cash distribution in Section [14]. Then we present the see-saw transaction mechanism in Section [15]. We stress that the

1Our results apply to all variants of poker. However, when we get into the details of the poker protocol, we will by default use Texas hold ‘em variant [31] for concreteness.
protocol for secure cash distribution makes white-box use of an underlying secure computation protocol, and may be inefficient in practice.

**Efficient protocols optimized for poker.** Fortunately, for specific applications such as poker, we are able to design an optimized protocol which makes only black-box use of a simpler secure function evaluation. See Section IV.

**Efficient protocols in an honest-majority setting.** While assuming a majority of parties is honest may not make sense for poker, there may be situations in which this is a reasonable assumption (e.g., a protocol for voting on how to allocate funds). In those cases, we provide a very efficient protocol (in terms of Bitcoin use) that can be implemented using the existing version of the Bitcoin protocol (we make use of the fact that, in the honest-majority setting, generic multiparty computation protocols can provide guaranteed output delivery). See Appendix B.

**Handling collusions.** A well-known cheating strategy in poker, when more than two players are involved, is secret collusion among subsets of players (for example, by exchanging their hole cards and deciding on a combined strategy that increases their chances to win the game).

In the online setting, verifying that parties are not communicating secretly is extremely difficult. Even worse, a single player could create multiple identities and occupy several seats at a table and gain an unfair advantage. Reducing the possibility of such attacks is perhaps the biggest practical problem faced in providing a good online poker service, whether it be centralized or decentralized [15]. Unfortunately, we do not know of any sure-fire way to ensure that two players are not communicating with each other outside the game.

We note that centralized poker networks combat such cheaters using heuristic software and the help of human fraud specialists. Also, poker websites get real-world identification and financial information which puts them in a much better position to catch cheating coalitions. These solutions are all heuristic, however.

Moreover, the problem of identifying malicious collusions becomes much harder in a setting where there is no trusted party. Thus, although we acknowledge the importance of detecting cheating coalitions, our protocols cannot guarantee that collusions do not occur. Here we merely suggest applying practical workarounds such as using “IP filtering,” “blacklisting services” (as suggested in [33]) etc., to detect and prevent cheating coalitions.

What our protocols do guarantee, however, is that cheating coalitions of players cannot gain any advantage beyond learning the cards held by the coalition. In other words, no coalition can learn more about an honest party’s hole cards, or the hidden community cards, than what they deduce from the cards in their coalition. The above notion corresponds to achieving “minimal effect of coalitions” [34].

**How to use our poker protocols.** We provide a basic overview of the threat model that we are concerned with in this paper. Note that our protocol provides full simulation security. However when it comes to security systems, there is typically a notable difference between theoretical and practical security. In the following, we mention some real-world attacks on poker protocols in general, as well as specific attacks against our protocols and outline the necessary assumptions to safeguard against them. First, since we aim to satisfy a strong notion of dropout tolerance we assume that parties have a good internet connection, and are not susceptible to DoS attacks (cf. [35]). Such assumptions are reasonable especially in a pseudonymous decentralized network (such as Bitcoin) where clients can easily connect from different end-points, and also quite common in works on secure multiparty computation [36]. Next, since we will be using timelocked Bitcoin transactions in the plain model realization, we assume that the parties have (at least loosely) synchronized clocks (or rough consensus about the current state of the Bitcoin ledger). We already discussed at length about collusions in poker, and how we settle for “minimal effect of coalitions” [34]. On the other hand, our protocols can be trivially adapted to play offline poker, or simply used only to play Heads Up poker (i.e., poker with exactly two players) where there is no risk of collusion. There is a plethora of other security concerns when implementing (even centralized) poker protocols. These include protection against automated playing bots, DoS, vulnerable user authentication, malware, and malicious remote access [37]. We point the reader to the discussion in [38] on how to cope with these attacks.

Finally, we note that the secure computation used in our poker protocol can be preprocessed (i.e., the heavy computation can be performed before the game, so that players do not have to wait in order to make moves), and thus the efficiency of the underlying secure computation is not a primary issue. (It is also relevant here to note that there have been major improvements to the practicality of secure computation in the recent past e.g., [39], [40], [41].) This in turn ensures that the $F_{CR}$ deposit phase that follows the secure computation protocol can also be preprocessed. This essentially removes the issue of long waiting times for block confirmations done by proof-of-work. Next, we note that the plain model realizations of $F_{CR}$ rely on Bitcoin scripts. While we explicitly specify the checks that the scripts need to perform, the current Bitcoin scripting language is very conservative (many opcodes became blacklisted [42]), and therefore some of the required checks are not currently supported.
in Bitcoin. However, the scripts in our protocols are of similar complexity to that of the Bitcoin scripting language. More concretely, in the dishonest majority setting our constructions require signature verification of arbitrary messages (i.e., not more burdensome than the supported signature verification for the entire transaction data), and simple arithmetic calculations that depend on the specific game played. See for example [27] for further discussion on the prospects of Turing complete scripts in cryptocurrencies. Let us emphasize that since we design our protocols in the $\mathcal{F}^{\text{CR}}_{\text{CR}}$-hybrid model, we do not care which underlying cryptocurrency is used as long as it provides a faithful implementation of $\mathcal{F}^{\text{CR}}_{\text{CR}}$.

Related work. We discuss three main lines of work that are highly relevant.

Mental poker. The seminal paper of [1] attracted several follow up papers that pointed out errors [43], [44], suggested better security goals [45], [46], [34], alternate constructions [47], extensions to the multiparty setting [48], [49], improved the efficiency [50], [51], and even implemented the protocol [52], [53], [54], [30].

We also point out that Qixcoin [55], an implementation at a preliminary stage, attempts to create a blockchain (analogous to Bitcoin) for gambling related activities, and is based on the mental poker framework [30]. Almost all of the papers above use homomorphic encryption to construct mental poker protocols.

Jakobsson et al. [16] consider a model with casinos and parties (but these are mutually distrustful and can also collude) and banks. In their scheme the casino and the bank together help in handling dropouts (without penalties). In comparison, our constructions do not assume the existence of a central trusted bank, and make do with a (much weaker) ideal claim-or-refund transaction functionality.

Secure Multiparty Computation. The general problem of secure computation was solved in the 2-party setting by Yao [56], in the multiparty setting in [57], [58], [3]. Besides not handling payments, none of the schemes above provide the type of dropout tolerance that is required for a fair poker game without trusted third parties.

In particular, in the dishonest majority setting, it is known that secure function evaluation (a single-stage computation) is equivalent to computing general reactive functionalities (with multiple stages), if we allow the adversary to abort [3]. However, this equivalence does not carry over to computation with penalties, because a one-round computation with penalties cannot enforce a penalty for parties who abort between rounds. Handling this kind of abort is one of the main technical contributions of this paper.

Bitcoin and secure computation. Andrychowicz et al. [24] designed a multiparty lottery protocol in the penalty model. The work of [25] designs a secure computation protocol in the penalty model but their protocol handles only the two-party setting. Secure multiparty computation with penalties and secure multiparty lottery with penalties were formalized, and protocols realizing these were constructed in [26]. The work of [27] show applications of Bitcoin to various interesting cryptographic primitives such as verifiable computing, etc.

II. Preliminaries

A. Definitions

A function $\mu(\cdot)$ is negligible in $\lambda$ if for every positive polynomial $p(\cdot)$ and all sufficiently large $\lambda$’s it holds that $\mu(\lambda) < 1/p(\lambda)$. A probability ensemble $X = \{X(a, \lambda)\}_{a \in \{0,1\}^*, \lambda \in \mathbb{N}}$ is an infinite sequence of random variables indexed by $a$ and $\lambda \in \mathbb{N}$. Two distribution ensembles $X = \{X(a, \lambda)\}_{\lambda \in \mathbb{N}}$ and $Y = \{Y(a, \lambda)\}_{\lambda \in \mathbb{N}}$ are said to be computationally indistinguishable, denoted $X \equiv Y$ if for every non-uniform polynomial-time algorithm $D$ there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$,

$$\Pr[D(X(a, \lambda)) = 1] - \Pr[D(Y(a, \lambda)) = 1] \leq \mu(\lambda).$$

All parties are assumed to run in time polynomial in the security parameter $\lambda$. We prove security in the “secure computation with coins” (SCC) model proposed in [26]. Note that the main difference from standard definitions of secure computation [36] is that now the view of $Z$ contains the distribution of coins. Let $\text{IDEAL}_{f,S,Z}(\lambda, z)$ denote the output of environment $Z$ initialized with input $z$ after interacting in the ideal process with ideal process adversary $S$ and (standard or special) ideal functionality $G_f$ on security parameter $\lambda$. Recall that our protocols will be run in a hybrid model where parties will have access to a (standard or special) ideal functionality $G_f$ on security parameter $\lambda$. We denote the output of $Z$ after interacting in an execution of $\pi$ in such a model with $A$ by $\text{HYBRID}^\pi_{\pi,A,Z}(\lambda, z)$, where $z$ denotes $Z$’s input. We are now ready to define what it means for a protocol to SCC realize a functionality.

Definition 1. Let $n \in \mathbb{N}$. Let $\pi$ be a probabilistic polynomial-time $n$-party protocol and let $G_f$ be a probabilistic polynomial-time $n$-party (standard or special) ideal functionality. We say that $\pi$ SCC realizes $G_f$ with abort in the $G_f$-hybrid model (where $G_f$ is a standard or a special ideal functionality) if for every non-uniform probabilistic polynomial-time adversary $A$ attacking $\pi$ there exists a non-uniform probabilistic polynomial-time adversary $S$ for the ideal model such that for every non-uniform probabilistic polynomial-time adversary $Z$,

$$\{\text{IDEAL}_{f,S,Z}(\lambda, z)\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*} \equiv \{\text{HYBRID}^\pi_{\pi,A,Z}(\lambda, z)\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*}. \diamond$$
The notion of fair secure computation as considered
in secure cash distribution generalizes secure computation with penalties) and refer the reader to [26], [27] for additional details.

### III. Secure Cash Distribution

In this section, we introduce secure cash distribution with penalties. Loosely speaking, secure cash distribution with penalties (or simply “secure cash distribution”) allows each party to first make a cash deposit and then supply additional inputs to a function. Then the deposited cash is distributed back to the parties depending on (and along with) the output of the function evaluation. Now any malicious party that aborts the protocol must pay a monetary penalty to all honest parties.

Clearly, such a primitive generalizes both secure computation with penalties [26], [25] and secure lottery with penalties [26], [24]. As it turns out, this informal definition of secure cash distribution is not strong enough to enable applications to multiplayer games (e.g., poker). What is needed is to handle the reactive setting, i.e., allowing multiple “stages” of computation with parties providing inputs to each stage and receiving outputs at the end of each stage. Let $F$ be a reactive functionality, i.e., one that keeps state across evaluations and proceeds in multiple stages. To keeps things simple, we assume an upper bound $\rho$ on the number of stages of $F$. That is, we assume $F = (f_1, \ldots, f_\rho)$ is a collection of functionalities which accumulate state with each evaluation.

More concretely, let $x_\ell = (x_{\ell,1}, \ldots, x_{\ell,n})$ denote the parties’ input to the $\ell$-th stage for $\ell \in [\rho]$, and let state_0 be initialized as state_0 := NULL. Then over the course of the computation, parties successively evaluate $f_\ell(x_{\ell}; \text{state}_{\ell-1})$ to obtain $(z_\ell, \text{state}_\ell)$ for $\ell = 1, \ldots, \rho$. Here $z_\ell := (z_{\ell,1}, \ldots, z_{\ell,n})$ represents the parties’ output, i.e., party $P_i$ obtains $z_{\ell,i}$. The value state_1 represents the state saved for the $(\ell + 1)$-th computation stage, and is kept private from the parties.

Although we now handle a reactive functionality, we stress that the cash that is deposited at the beginning of the protocol is distributed only at the end (i.e., no cash distribution occurs in any intermediate stage). That is, secure cash distribution provides a means to keep the cash deposited in escrow while parties’ learn output from each stage’s function evaluation, and thus can revise their inputs to a later stage. The capability to maintain an escrow turns out to be crucial in enabling applications such as poker, auctions, etc.

We now proceed to the formal details. Let $d_\ast = (d_1^\ast, \ldots, d_n^\ast)$ be the initial cash deposit from the parties, i.e., party $P_i$ deposits $\text{coins}(d_i^\ast)$ into the computation. Then at the end of the protocol all the deposited coins, i.e., $\text{coins}(\sum_{i \in [n]} d_i^\ast)$, are distributed back to the parties...
according to the evaluation of the reactive functionality $F$ on the parties’ inputs. More precisely, let $z_p$ denote the parties’ output at the end of the last stage of the computation. We assume that $z_p$ specifies how the coins are (re)distributed at the end of the entire computation. That is, we can parse $z_p = (z = (z_1, \ldots, z_n), z^* = (z^*_1, \ldots, z^*_n))$ where $z_i$ represents the parties’ output, and $z^*_i$ represents the amount of cash that $P_i$ is supposed to get back. We are now ready to define bounded zero-sum reactive distribution.

**Definition 2** (Bounded zero-sum reactive distribution). For all $\ell \in [\rho]$, let $f_i : (\{0, 1\}^\ast)^n \times \{0, 1\}^\ast \times (\{0, 1\}^*)^n \times \{0, 1\}^\ast$ be a function. Let $d^* = (d_1^*, \ldots, d_n^*) \in \mathbb{Z}^n$ be a vector. We say that $(F = (f_1, \ldots, f_\rho), d^*)$ is a bounded zero-sum reactive distribution if $\forall x_1, \ldots, x_\rho \in (\{0, 1\}^\ast)^n$ it holds that the value $z_p = ((z_1, \ldots, z_n), (z^*_1, \ldots, z^*_n)) \in ((\{0, 1\}^*)^n \times \mathbb{Z}^n$ obtained from the sequence:

- $(z_1, \text{state}_1) \leftarrow f_1(x_1; \text{NULL})$;
- $(z_2, \text{state}_2) \leftarrow f_2(x_2; \text{state}_1)$;
- \ldots
- $(z_\rho, \text{state}_\rho) \leftarrow f_\rho(x_\rho; \text{state}_{\rho-1})$,

satisfies $\sum_i z^*_i = \sum_i d^*_i$.

**Observation 1.** Note that it may very well be the case that for some inputs $x$ the coins earned by $P_i$, namely $z^*_i$ may be such that $z^*_i > d^*_i$ (i.e., it earned more than it deposited). This for e.g., would be the case when $f^*$ implements the lottery functionality as in [24], [20]. To aid clarity and to make things concrete in such cases, we make use of a “helper” function $g$ which we describe below. The function $g$ on input $(d^*, z^*)$ returns a matrix $A$ whose $(i, j)$-th entry denoted $a_{i,j}$ specifies the amount of coins that need to be transferred from party $P_i$ to $P_j$. In particular, it must hold for all $i \in [n]$ that $\sum_{j\in [n]} a_{i,j} = d^*_i$ and for all $j \in [n]$ that $\sum_{i\in [n]} a_{i,j} = z^*_j$. We merely state that there are several ways to design such a function $g$ given $f^*$ such that $(f^*, d^*)$ is a zero-sum distribution, and defer further details to the full version.

Next we present the formal definition of $F_{F^*, d^*}$ which idealizes secure cash distribution with penalties.

**Ideal functionality $F_{F^*, d^*}$.** See Figure 2 for the formal definition. In an initial cash deposit phase, the functionality $F_{F^*, d^*}$ receives $\text{coins}(d + d^*_r)$ from each honest $P_i$, where $d$ represents a parameterizable safety deposit and $d^*_r$ represents the cash that will be stored in escrow. In addition, $F_{F^*, d^*}$ allows the ideal world adversary $S$ to deposit some coins which may be used to compensate honest parties if $S$ aborts after receiving the outputs. If there is an abort at this stage, that is $S$ does not submit the necessary amount of cash then the protocol terminates, and the honest parties get their deposit back. Note that at this stage there is no penalty for aborts; the penalties enter the picture only after this stage. Once the deposit phase ends, parties enter the computation phase. In the $\ell$-th stage of the computation phase, the honest parties supply their inputs to $\ell$-th stage of the computation. The functionality then waits to receive corrupt parties’ inputs for this stage. If $S$ aborts at this stage, then the honest parties receive $\text{coins}(d)$ in addition to getting their deposit $\text{coins}(d)$ back (and may also obtain some extra $\text{coins}(q_r)$), and the protocol is terminated. However, if $S$ does continue (i.e., provide inputs to this stage), then the functionality computes the output of the $\ell$-th stage. Now the simulator gets a chance to look at the output first, and then decide if it wants to
continue or not. If it decides to continue then the honest parties receive the output as well, and proceed to the next stage of the computation. On the other hand, if $S$ decides to abort, then the honest parties get compensated as before, i.e., with coins$(d + q + q_r)$, and the protocol is terminated. The computation phase terminates after the $\rho$-th stage ends. After this, parties enter the cash distribution phase where the cash is distributed according to the output of the $\rho$-th stage, i.e., $z_\rho$. The functionality parses $z_\rho$ to obtain $z^*$ which dictates how the cash is distributed among the parties. Using $z^*$, the functionality distributes the cash among the parties, and returns their original deposits as well. In addition, the functionality also sends the value $z^*$ to all parties, i.e., the way the cash gets distributed at the end is not private.

**How to implement poker using $F_{F,d}^*$.** We describe a naïve implementation of how to play poker hand (via $F_{F,d}^*$), and defer optimizations to Section [V]. We assume that there is a bound on the maximum number of betting stages within a single hand. Players start the protocol by depositing their “chips” or equivalently cash to $F_{F,d}^*$. This ends the deposit phase. Now players supply inputs to the first stage function whose purpose is to deal players’ hole cards. They do this by each picking uniform random string and sending it to $F_{F,d}^*$. That is player $P_i$ picks and sends $r_i$ to $F_{F,d}^*$. Then $F_{F,d}^*$ computes $f_1(r_1, \ldots, r_n)$ in the following way: first compute $\tilde{r} = \bigoplus_{i=1}^n r_i$ (note: no coalition of malicious players can influence $\tilde{r}_i$ in any way), then interpret uniform random string $\tilde{r}$ in a natural way to generate players’ hole cards as well as the community cards. This value $\tilde{r}$ is then saved to the private state. Now note that any player that aborts without supplying $\tilde{r}_i$ pays a penalty to every honest player. Otherwise, players get their hole cards (the community cards still remain hidden), and can start to place bets. Again note that any player that aborts after seeing its hole cards pays a penalty to every honest player. Each move by a single player is considered as a computation stage. In the stage corresponding to player $P_i$’s turn, $P_i$ simply submits its next move (e.g., “match,” “fold,” “raise by $1$”) as the input to the stage. (Other players have no inputs to this stage.) Then the stage computation is simply to append player $P_i$’s move to the saved transcript of bets made so far (i.e., the state of the previous stage), and then send $P_i$’s move to all players. It’s possible that a player $P_i$ submits an illegal move (i.e., inconsistent with the transcript, or simply overbets) in which case the last stage computation will reconstruct an illegal transcript, and ensure that the cash distribution phase compensates every honest party with coins$(q)$, i.e., $z^*_i = -hq$. Note that players never submit any additional coins (other than at the beginning, i.e., the deposit phase). In the stage corresponding to revealing community cards (say after the last player has placed its bet), the stage function simply uses $\tilde{r}$ to regenerate the community cards that need to be revealed, and additionally broadcasts the last player’s move. Again a player that aborts after seeing the community cards pays a penalty to every honest player. Players keep continuing to make their moves during their turn until it’s time for the last move to be made. Once this move is made, $F_{F,d}^*$ first determines the pot (using the bets made in the game that can be found in the saved state containing the transcript), and then send the pot earnings to the winner(s), and the remaining cash (from that deposited initially) back to the players. To play the next hand, players execute the above all over again.

It is instructive to note why just secure computation with penalties does not seem powerful enough to implement poker. Note that secure computation with penalties can indeed implement each stage of the computation. At first glance chaining them together seems to solve the problem. However, this is an incorrect approach since there is no way to force players’ to continue to the next stage (in particular to supply inputs to the next stage). Indeed, the only guarantees that we get from such an approach is that malicious players who learn the output of a stage of computation cannot prevent honest parties from learning the same (except by paying a penalty). This is not enough to satisfy the notion of dropout tolerance that we desire since a player may dropout in the middle of a hand without getting penalized.

**IV. Realizing Secure Cash Distribution**

In this section we provide the blueprint of our protocol that realizes secure cash distribution with penalties. As we will see soon, our general strategy is to use a protocol that securely realizes a standard reactive functionality$^2$ (with no coins, and unfair abort), denoted $F_F$, to set things up such that the see-saw transaction mechanism of Section [V] applies to ensure that either the protocol is completed until the very end or all honest parties get compensated. Then, to make the final transfers between parties we will make use of a cash distribution mechanism that we describe later in this section.

To simplify the presentation of our protocol, we consider the case when there is only a single stage in the computation, i.e., $\rho = 1$. Essentially this means that we are dealing with secure function evaluation but with an important difference: namely, aborts anywhere during the computation (i.e., not only at the output delivery step) will be penalized. Once we describe our single-stage protocol, it will be clear how to design a protocol for multiple stages.

$^2$Recall that in such a realization the updated state at the end of a stage is $n$-out-of-$n$ secret shared among the parties.
First let us set up some notation. Recall that we say 
\((r, i) > (r', i')\) iff either (1) \(r > r'\), or (2) \(r = r'\) 
and \(i > i'\). For \((r, i)\), let \(\text{pred}(r, i)\) be \((r', i')\) such 
that for every \((r'', i'')\) it holds that \((r'', i'') < (r, i)\) iff 
\((r'', i'') \leq (r', i')\). In other words, \(\text{pred}(r, i)\) is the 
“predecessor” of \((r, i)\). Let \(\pi_f\) be a \(m\)-round protocol 
that realizes function \(f\). For each \(i \in \n\), let \(x_i\) denote 
party \(P_i\)'s input to \(f\). We assume that in each round 
of the protocol, parties take turns to broadcast their 
message, i.e., the entire protocol transcript is public. 

Let \(TT_{r,i}\) denote the transcript of protocol \(\pi_f\) up until 
party \(P_i\)'s message in the \(r\)-th round. Let \(\text{nmf}_{r,i}\) denote 
the next message function for party \(P_i\) in round \(r\). The 
function \(\text{nmf}_{r,i}\) takes as input the actual input \(x_i\), the 
private randomness of party \(P_i\), denoted \(\omega_i\), and the pub-
lic transcript seen so far, i.e., \(TT_{\text{pred}(r,i)}\). In other words, 
we have that \(TT_{r,i} = \text{nmf}_{r,i}(TT_{\text{pred}(r,i)}; (x_i, \omega_i))\). Also, 
since all messages are public broadcasts, there 
exists a function \(TV_{r,i}\) which checks if a given transcript \(TT_{r,i}\) 
(that contains all messages until and including party \(P_i\)'s 
message in round \(r\)) is valid or not. By definition, we have 
that \(TV_{r,i}(TT_{r,i}) = 1\). For simplicity and wlog, we 
assume that all messages (i.e., transcripts) broadcasted 
are signed by the sending party\(^3\). This in turn means 
that the function \(TV\) checks validity of the transcript 
also checks to see if it contains the necessary signatures.

Our strategy is to force each party \(P_i\) to deliver its 
round \(r\) message during its turn. That is, first, we want 
party \(P_i\) to either reveal its first round message \(TT_{1,i}\) 
to all parties, or pay a penalty to all parties. If \(P_i\) revealed 
\(TT_{1,i}\), then \(P_2\) can apply \(\text{nmf}_{1,2}(TT_{1,2}; (x_2, \omega_2))\) to 
get \(TT_{1,2}\). Now we want \(P_2\) to either reveal \(TT_{1,2}\) 
to all parties or otherwise pay a penalty to all parties. 
This way, we want to force every party to either make its 
move or otherwise pay a penalty. If we implement this 
strategy successfully, then we have ensured that each 
party either learned its output, or is compensated with 
a penalty. (Note that cash distribution at the end still 
needs to handled.) Designing a transaction mechanism 
for implementing the above strategy is one of our main 
contributions in this paper. We defer the presentation 
of the transaction mechanism to Section\(^V\) and devote the 
rest of this section to handling other important issues.

Handling multiple valid transcripts. In an actual 
implementation of the above strategy in the \(FCR^\ast\)-hybrid 
model, we will have parties receive multiple \(F_{CR}^\ast\) trans-
actions from other parties that can claimed if they 
produce a valid transcript. Herein lies a problem since 
a malicious party may claim a subset of these \(F_{CR}^\ast\) tran-
sactions using one valid transcript and a different 
subset using a different valid transcript. Such an “at-
tack” may indeed be possible, for e.g., by varying the 
actual input and private randomness input to the next 
message function. In a multiparty setting, the problem 
is exacerbated by the fact that malicious coalitions of 
k consecutive parties can potentially change the last 
k messages in the transcript (since they possess the 
required signing keys to do this). Such attacks could 
de devastate the applications that we are interested in. 
For e.g., in applications to poker, a player (admittedly 
a novice) may leak an “expression of surprise” upon 
seeing a (malicious) player’s “confirmed” move, only to 
see this move modified by the next (malicious) player. 
In any case, we consider such attacks as violations, 
and must compensate the honest parties upon such violations.

Fortunately, compensating honest parties in the face 
of such violations is quite easy since a “proof” of any 
such violation is readily obtained from the inconsistent 
transcripts. In more detail, we ask each party \(P_i\) to make 
\(FCR\) transactions to every other party that can be claimed 
by revealing a proof of violation: i.e., pair of transcripts 
\(T^\ast_{vi} = (TT_{vi}, TT'_{vi})\) such that for some \(r \in [m]\), it holds that 
\(TV_{r,i}(TT_{r,i}) = TV_{r,i}(TT'_{r,i})\) and yet \(TT_{r,i} \neq TT'_{r,i}\). Note 
that since transcripts are signed, a proof of violation against 
an honest party can never be obtained (except with 
negligible probability). Following the notation in \([26]\), we 
use \(P_1 \xrightarrow{\tau_{q,\tau}} P_2\) to denote an \(FCR\) transaction for 
coins\((q)\) made by \(P_1\) that can be claimed by \(P_2\) if \(P_2\) 
does produces witness \(\tau\) within time \(\tau\). Thus to safeguard 
against violations we ask each \(P_i\) to make the following 
set of transactions for each \(j \in [n] \setminus \{i\}\): 
\[
P_i \xrightarrow{T^\ast_{vi}, \nuq, \tau} P_j (TX^\ast_{ij})
\]

Here \(\tau\) is such that the transaction can be claimed 
until the end of the protocol. Note that the transaction if 
claimed will transfer coins\((\nuq)\) from the violating party 
\(P_i\). This is because, upon such a violation an honest \(P_j\) 
will be asked to abort the rest of the protocol and directly 
claim \(TX^\ast_{ij}\) where \(P_i\) is the violating party. Since \(P_j\) 
aborts the rest of the protocol, it may be forced to pay a 
total compensation of coins\((n - 1)q\) to the remaining 
parties. Thus upon any violation by malicious parties, we 
ensure that each honest \(P_j\) will still be coins\((q)\) up at 
the end of the protocol execution. Note that since honest 
\(P_j\) aborted immediately after a violation, no malicious 
party will be able to accuse \(P_i\) of violating the protocol 
(except with negligible probability).
Handling multiple stages of computation. At an abstract level, adding stages to a reactive computation merely amounts to adding more “next messages” to the transcript. Indeed an intermediate stage of computation simply begins by reconstructing the current state, and then performing the computation on this state and the current inputs. Thus it is trivial to merge multiple stages of computation into a single stage—simply append the protocol messages of the multiple stages together. Since our strategy works by keeping track of the protocol transcript, it ensures that an abort at any round/stage of a multi-stage computation will be penalized.

Handling the cash distribution. Although our strategy ensures that parties either continue to the end of the protocol or compensate all honest parties, there is no actual money transfer taking place (even in an honest execution)! Indeed the strategy described so far handles only the case when \( d^* = (0, \ldots, 0) \). We now rectify the situation. To do this, we first need parties to make deposits at the beginning of the protocol that will allow them to claim their returns at the end of the protocol. Note that parties might have to transfer an arbitrary amount of coins between each other. Thus, it seems we need to ask the parties to make several \( F_{CR} \) transactions each one of which transfers coins(1) (i.e., a single unit of money). This can be quite prohibitive in general. Luckily, we use a different technique that lets parties transfer money. This can be quite prohibitive in general. Luckily, we use a different technique that lets parties transfer money. This technique turns out to be useful since it allows reduction in number of \( F_{CR} \) transactions relative to transferring money in units of 1, while still allowing parties to transfer arbitrary amounts of coins between each other.

We describe our technique in more detail. Let \( d^* = (d^*_1, \ldots, d^*_n) \), and for each \( i \in [n] \), let \( m_i = \lceil \log(d^*_i) \rceil \). The high level idea is to have, for every ordered pair \((i, j)\) with \( i, j \in [n] \) and \( i \neq j \), and for each \( k \in [m_i] \), party \( P_i \) make an \( F^* \) transaction as follows:

\[
P_i \xrightarrow{\rho_{i,j,k}} P_j \quad (\text{Tx}^n_{i,j,k})
\]

Given these transactions, it is easy to see that \( P_j \) can claim any arbitrary amount of coins from the rest of the parties. Also, we need to ensure that \( P_j \) obtains exactly the correct amount of coins, no more and no less. That is, suppose the output of the reactive computation is \( z^* = (z_1, \ldots, z^n) \) with \( z^* = (z^*_1, \ldots, z^*_n) \), then we want \( P_j \) to obtain coins(\( z^*_j \)) at the end of the protocol. In other words, we need to provide \( P_j \) with the right subset of \( \{T^\text{fin}_{i,j,k}\}_{i,k} \) that will allow it to claim exactly coins(\( z^*_j \)).

This subset will obviously need to be transferred in the last computation stage \( f_\rho \). However to make sure the deposits are made at the very beginning (after all there is no way to enforce that parties make \( F_{CR} \) deposits at the end of the computation stage), the parties need to know the corresponding verification circuits \( \phi^\text{fin}_{i,j,k} \) at the beginning as well. To design the verification circuits, we employ honest binding commitments [62]. [26]. Let \((S, R)\) be a honest binding commitment scheme (cf. Appendix A). More precisely we require parties to execute a secure computation protocol executed at the very beginning that for all \( i \in [n], j \in [n] \setminus \{i\}, k \in [m_i] \):

- chooses \( T^\text{fin}_{i,j,k} \leftarrow \{0, 1\}^\lambda \) at random;
- computes \( \text{conv}^\text{fin}_{i,j,k} \leftarrow S(1^\lambda, T^\text{fin}_{i,j,k}, \omega^\text{fin}_{i,j,k}) \);
- \( n \)-out-of-\( n \) secret shares each \( T^\text{fin}_{i,j,k} \);
- outputs \( \text{conv}^\text{fin}_{i,j,k} \) and \( i \)-th share of \( T^\text{fin}_{i,j,k} \) to each \( P_i \).

The secret sharing is done so that parties can reconstruct the \( T^\text{fin}_{i,j,k} \) values (saved as part of the state) at the beginning of the (last) stage of the computation. Note that now parties possess the verification circuits \( \phi^\text{fin}_{i,j,k} \) to make the transaction \( \text{Tx}^n_{i,j,k} \). Next we describe the modification to the (last) stage. Instead of realizing \( f_\rho \) in the last stage, parties realize \( f'_\rho \) which:

- computes \( z^* = (z^*_1, \ldots, z^*_n) \) by invoking \( f_\rho \);
- computes \( A \leftarrow g(d^*, z^*) \) (cf. Observation 4), lets \( a_{i,j} \) denote the \((i, j)\)-th entry of matrix \( A \), and lets \( b^*_i, i = 1, \ldots, m \), be the binary representation of \( a_{i,j} \);
- for all \( i \in [n], j \in [n] \setminus \{i\}, k \in [m_i] \):
  - reconstructs \( T^\text{fin}_{i,j,k} \) (from state \( e_{r-1} \));
  - outputs \( T^\text{fin}_{i,j,k} \) if \( b^*_i = 0 \), else outputs 0.

Given the above it is easy to see that the set of transactions \( \{\text{Tx}^n_{i,j,k}\} \) indeed transfer the right amounts of money according to the output \( z^* \).

Now all that remains to be seen is how to design a \( F_{CR} \) transaction mechanism that implements our strategy of forcing parties to send the next message of the protocol realizing \( F_F \).

V. See-saw Transaction Mechanism

Recall that our goal is to force parties to reveal their next message of say a \( m \)-round protocol for computing function \( f \), one-by-one in a round-robin fashion round after round. That is, party \( P_1 \) first computes and reveals “token” \( T_{1,1} = \text{TT}^f_{1,1} \) then party \( P_2 \) computes (using \( T_{1,1} \)) and reveals token \( T_{1,2} = \text{TT}^f_{1,2} \), and so on until party \( P_n \) computes and reveals token \( T_{1,n} = \text{TT}^f_{1,n} \). (Note that the order of revelations is important.) Following this, parties move on to the next round, and so on and so forth until at the end \( P_n \) reveals token \( T_{m,n} = \text{TT}^f_{m,n} \). What we need is a transaction mechanism that incentivizes parties to follow the above sequence of reveals. More precisely for every \( i \in [n], r \in [m] \), we force \( P_i \) to pay a penalty to all other parties if (a) all parties \( P_1, \ldots, P_n \) revealed their tokens until round \( r - 1 \); and (b) in round \( r \) parties \( P_1, \ldots, P_{r-1} \), revealed their tokens; and (c) in round \( r \) party \( P_i \) did not reveal \( T_{r,i} \).
Towards solving this problem, we let parties participate in a *initial deposit phase* where parties make some sequence of transactions. We are lenient towards any aborts during this initial deposit phase, i.e., we do not penalize any party for an abort during this deposit phase. However once this deposit phase ends, then we enter the *reveal phase*. From here on out, any party that deviates during its turn in any of the $m$ rounds has to pay a penalty to all the remaining parties. For the sake of clarity, we make two simplifying assumptions. The first is that our constructions will try to penalize deviations of party $P_1$ in round $r$ only when $(r, i) \neq (1, 1)$. Towards the end of the section, we handle the “bootstrapping” step of forcing $P_1$ to start the protocol. (This step is actually common to all our constructions.) The second is that we assume parties can use only unique witnesses to claim $F^\text{CR}$ transactions. In our constructions, the witnesses correspond to protocol transcripts, and we already discussed in the previous section how to handle the case when parties broadcast multiple valid transcripts. With these two simplifying assumptions in place, we construct our final protocol in a step-by-step manner. First we consider the simplest setting of $n = 2$ and $m = 1$.

**Single-round two-party case.** Since we are in the single-round case we use $T_i$ to denote the token $T_{i,1}$. Consider the following sequence of deposit transactions where $\tau_2 > \tau_1$:

\[
\begin{align*}
P_1 &\overset{T_1 \lor T_2}{\longrightarrow} q_{r, \tau_2} P_2 & (T_{X_2}) \\
Q &\overset{T_1}{\longrightarrow} q_{\tau_1} P_1 & (T_{X_1})
\end{align*}
\]

Note that the verification circuits for these transactions are simply the corresponding transcript checking functions $tv^n_{r, i}$, and are already known to the parties, and thus the deposits can be made. Once all the deposits are made, the deposits are claimed in reverse. That is, $P_1$ first claims $T_{X_1}$. Using $T_1$ revealed by $P_1$, party $P_2$ is able to claim $T_{X_2}$. We proceed to analyze what happens when parties try to abort. We first consider aborts during the initial deposit phase. If $P_1$ aborts without making $T_{X_2}$, then clearly no money changes hands and we are good. Now if $P_2$ aborts without making $T_{X_1}$, then note that $P_1$ does not enter the reveal phase, and so does not reveal $T_1$. This in turn ensures that $P_2$ will not be able to claim $T_{X_2}$, and thus no money changes hands, and we are good. These attacks ensure that we do not even get past the deposit phase. This however means that we are not required to penalize any party.

---

**Multi-round two-party see-saw mechanism.** The sequence of transactions are shown in Figure 3 where $\tau_{r, i'} > \tau_{r, i}$ iff $(r, i') > (r, i)$. As in the single-round case, the reveals are made in reverse: namely, $P_1$ first claims $T_{X_{1,1}}$. Using $TT_{1,1} = T_{1,1}$ revealed by $P_1$, party $P_2$ is now able to claim $T_{X_{1,2}}$ by revealing $TT_{1,2} = T_{1,2} \land T_{1,2}$. Likewise parties $P_1$ and $P_2$ take turns claiming each other’s $F^\text{CR}$ transactions.

We first consider aborts during the initial deposit phase. Suppose $P_1$ aborts without making $T_{X_{r,j}}$ for $j \neq i$ and some $r$. First, this ensures that (1) $P_1$ does not make $T_{X_{r,j}}$ for $(r, i') < (r, j)$, and (2) $P_2$ will never reveal $T_{r, j}$ (since $T_{r, j}$ needs to be revealed only to claim $T_{X_{r,j}}$ for $(r, i') \geq (r, j)$), and (3) no party can claim $T_{X_{r,j}}$ for $(r, i') \geq (r, j)$ (since $T_{r, j}$ is necessary to claim $T_{X_{r,j'}}$), and (4) all the deposits $T_{X_{r,j'}}$ for $(r, i') > (r, i)$ (i.e., those that were made so far) will get penalized. Note that in the “ladder” mechanism of [26], each party $P_i$ has a token $T_i$, and to reconstruct the final output all tokens need to be revealed. However, once the reveal phase starts, then a party may refuse to reveal its token and yet not get penalized. On the other hand in our setting, once the reveal phase starts, then any party that chooses not to reveal its token will be forced to pay a penalty to all remaining parties.
automatically refunded after $\tau_{r',i'}$ (since $T_{r,j}$ is need to claim this, but is never revealed by $P_j$). Thus in such a situation neither party stands to gain or lose coins. Next, we discuss aborts by parties in the reveal phase.

First suppose $P_1$ aborts without claiming $T_{X_{1,1}}$. In this case, dishonest $P_1$ will never obtain $T_{1,2}$. This is because $P_2$ would not have obtained $T_{1,1}$ from $P_1$, and hence would not claim $T_{X_{1,2}}$. Now note that all deposits $T_{X_{r,i}}$ for $(r, i) \geq (1, 2)$ require $T_{1,2}$, and hence none of these deposits can be claimed. Thus we have that neither party stands to lose or gain coins. Recall that this corresponds to the case where the reveal phase hasn’t started yet, and so parties don’t get penalized yet.

Recall that once the reveal phase starts, we must penalize every party that did not reveal its token during its turn. Suppose $P_1$ does claim $T_{X_{1,1}}$ (i.e., the reveal phase has started). Then in this case, $P_2$ is down coins$(q)$ while $P_1$ is up coins$(q)$. If $P_2$ aborts at this stage, then essentially $P_2$ has compensated $P_1$ with coins$(q)$. On the other hand if $P_2$ claims $T_{X_{1,2}}$, then note that it gets coins$(2q)$ from that claim. Thus, it is now coins$(q)$ up while $P_1$ is down coins$(q)$. It is easy to see that as the remaining claims are made, parties take turns going up and down coins$(q)$ (hence the name “see-saw”). Thus we have the property that whenever a party $P_i$ claims $T_{X_{r,i}}$ (except for $(r, i) = (m, 2)$), it gains coins$(q)$ while the other party loses coins$(q)$. This incentivizes the other party to go ahead and claim $F_{CR}$ transaction immediately above $T_{X_{r,i}}$, say $T_{X_{r',i'}}$. Indeed if the other party does not make the claim, then we have that the honest party (i.e., $P_1$) is compensated with coins$(q)$ at the end of the protocol. This is because if $T_{X_{r',i'}}$ is not claimed, then either (1) $(r', i') = (m, 2)$, and this case $P_1$ does not lose coins from this transaction, and simply ends the protocol with coins$(q)$ as compensation, or (2) $(r', i') \neq (m, 2)$, in which case $P_2$ will never reveal $T_{r+1,i}$ thus making it impossible for any $T_{X_{r'',i''}}$ to be claimed for any $(r'', i'') \geq (r + 1, i)$, essentially ensuring that no further money transfers happen, and that $P_1$ can end the protocol with coins$(q)$ as compensation.

Finally, in an honest execution, when $P_2$ claims the last transaction $T_{X_{2,m}}$, it gets only coins$(q)$ from that claim, and thus in this case both parties even out.

**Multiparty locked ladder mechanism.** Generalizing the two party solution turns out to be somewhat nontrivial. To better understand the complications we will first look a naïve 3-party protocol.

**Naïve single-round 3-party case.** The high level idea is to try and ensure that all parties are already compensated by $P_i$ just before the step where party $P_i$ is required to reveal $T_i$. Then after $P_i$ is supposed to reveal $T_i$, we get the compensation that was delivered to the parties back to $P_i$. (Observe that we do not need to apply the above strategy for $i = 1$.) Consider the following implementation of the above strategy:

<table>
<thead>
<tr>
<th>Roof deposits. For $j \in {1, 2}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_j \xrightarrow{T_{T3}^{\tau}} P_3$</td>
</tr>
<tr>
<td>$(T_{X_{1,2}^{(0)}})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Third stage deposits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_3 \xrightarrow{T_{T2}^{\tau}} P_2$</td>
</tr>
<tr>
<td>$(T_{X_{2,3}^{(1)}})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Second stage deposits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_2 \xrightarrow{T_{T1}^{\tau}} P_1$</td>
</tr>
<tr>
<td>$(T_{X_{1,2}^{(0)}})$</td>
</tr>
</tbody>
</table>

To see why the above may be a faithful implementation of the strategy, note that the end of the first two deposit stages, $P_2$ has already compensated both $P_1$ and $P_3$ with coins$(q)$, i.e., $P_2$ has lost coins$(2q)$. Then, in the third stage, it claims coins$(3q)$ from $P_3$ by revealing $T_2$. This is effectively equivalent to $P_3$ compensating $P_2$ with coins$(q)$, and learning $T_1$ and $T_2$. That is, at the end of the third stage, it is $P_3$’s turn to reveal $T_3$, and both $P_1$ and $P_2$ have already been compensated with coins$(q)$ by $P_3$. Then, in the roof stage, $P_3$ claims back coins$(q)$ from both $P_1$ and $P_2$ by revealing $T_3$ (along with $T_1, T_2$), and thus all parties even out.

The problem with the above scheme is that it is not resistant to a “coalition attack.” Consider a malicious $P_2$ that does not make the first and second stage deposits. Recall that the roof deposits and the third stage deposits have already been made, i.e., the protocol terminates at the “dotted line.” Now a malicious coalition of $P_1$ and $P_2$ possesses both $T_1$ and $T_2$, i.e., $TT_2$ and can claim the third stage deposit of coins$(3q)$. While $P_3$ can use $TT_2$ to claim the roof deposits, and learn all the tokens, it does so at an expense of coins$(q)$ (i.e., it claims coins$(2q)$ from the roof deposits but has lost coins$(3q)$ in the third stage deposits). This is clearly an undesirable situation as the honest party has lost coins$(q)$.

To avoid the “coalition attack,” we now introduce two new ideas that will help us construct our multiparty protocol. The first idea is a *locking mechanism* that prevents the collusion attack that we just described on our naïve 3-party protocol. The second is an integration of the first idea with the *ladder mechanism* of [26] which allows transitions between different stages of the protocol. We explain these two ideas below.
The mechanism. Recall that the high level idea in our naive 3-party protocol was to ensure that all parties are already compensated by $P_i$ just before the step where party $P_i$ is required to reveal $T_i$. Then after $P_i$ reveals $T_i$, we get the compensation that was delivered to the parties back to $P_i$. That is, we have a set of transactions $S_{+i}$ where $P_i$ claims coins($q$) each from a set of parties, followed by a set of transactions $S_{-i}$ where the same set of parties each claim coins($q$) from $P_i$. (Recall that transactions in $S_{-i}$ are claimed first, which forces $P_i$ to reveal $T_i$ and claim transactions in $S_{+i}$.)

The general form of the attack on the naive protocol is that $P_i$ aborts when it has to make transactions in $S_{-i}$. Then colluding with parties $P_1, \ldots, P_{i-1}$, party $P_i$ starts claiming transactions in $S_{+i}$. This allows $P_i$ to unfairly obtain additional coins from parties $P_{i+1}, \ldots, P_n$, while ensuring that they are unable to claim deposits in $S_{-i}$.

The main idea that we use to prevent such attacks is to “lock” transactions in $S_{+i}$ such that they can be “unlocked” and claimed only if the transactions in $S_{-i}$ were already claimed. To do this, we make use of “dummy tokens” $U_{i,j}$ that will be used by $P_j$ (and known only to $P_j$) to lock transactions in $S_{+i}$. (We will generate these dummy tokens via an initial secure computation protocol. A similar strategy is used to “bootstrap” the computation, and we defer details until then.) More concretely, to claim the transaction from $P_j$ in $S_{+i}$, now party $P_i$ has to produce $U_{i,j}$ in addition to $TT_i$. Then to enable an honest $P_i$ to claim transactions in $S_{+i}$, we let party $P_j$ to claim transactions in $S_{-i}$ only if it produces $U_{i,j}$ in addition to $TT_{i-1}$. A similar strategy is used to “bootstrap” the computation, and we defer details until then. More concretely, to claim the transaction from $P_j$ in $S_{+i}$, now party $P_i$ has to produce $U_{i,j}$ in addition to $TT_i$. Then to enable an honest $P_i$ to claim transactions in $S_{+i}$, we let party $P_j$ to claim transactions in $S_{-i}$ only if it produces $U_{i,j}$ in addition to $TT_{i-1}$.

Ladder mechanism. While the above locking mechanism deals with aborts in the deposit phase, we must obviously be wary of aborts in the reveal phase. Indeed, it turns out that the locking mechanism alone does not suffice. To see why, watch what happens when it is (honest) $P_i$’s turn to reveal the witness, and yet none of the parties claim transactions in $S_{-i}$ thus disabling $P_i$ from revealing its token. In effect, all parties other than $P_i$ have aborted, and yet $P_i$ does not receive any compensation, thus violating our requirements. For a more concrete example of what we refer to as the “locked-out attack,” consider the naive 3-party protocol enhanced with the locking mechanism (i.e., both second stage as well as the third stage deposits are locked). Now $P_i$ claims the first stage deposit, and after that $P_3$ simply aborts without claiming the second stage locked transaction. This will disallow $P_2$ from claiming the the third stage deposit as it remains locked. Thus, essentially $P_3$ aborted the protocol, and yet $P_2$ does not gain coins($q$) (in fact, it loses coins($q$) here).

The above attack naturally leads us to include a $\mathcal{F}_{CR}$ transaction to $P_i$ that can be claimed just by revealing $TT_i$, i.e., it is essentially an unlocked transaction.

![Diagram](image)

**Fig. 4.** Locked ladder mechanism.

At a high level, the protocol proceeds by getting a roof deposit from each of the parties to $P_n$ that can be claimed if $P_n$ produces $TT_n$. Next, we enter the ladder deposits for each $i = n - 1$ down to 2 (note the order is important), where party $P_i$ receives a deposit that is locked with token $U_{i,j}$ from each party $P_j$ for $j > i$ (these correspond to $S_{+i}$), and makes deposits to $P_j$ for $j > i$ that are locked with $U_{i,j}$ (these correspond to $S_{-i}$), and finally receives an unlocked deposit from $P_{i+1}$ that can be claimed if $P_i$ reveals $TT_i$. Note that deposits in $S_{-i}$ can be claimed with $TT_{i-1}$ (in addition to $U_{i,j}$), and that deposits in $S_{+i}$ can be claimed with $TT_i$ (in addition to $U_{i,j}$). Finally, we have the foot deposit (essentially foot of the ladder that involves $P_1$) where $P_2$ makes a deposit.
to $P_1$ that can be claimed with $T_1$.

As usual these deposits will be claimed in reverse. That is, $P_1$ first claims the floor deposit by revealing $T_1$. Then parties enter the ladder reveal phase. As in [26], the parties climb the ladder metaphorically as they take turns claiming the ladder deposits. The difference from [26] is that before climbing a rung of the ladder, parties first do a “rung lock” step, and after they climb the rung, they perform a “rung unlock” step (i.e., those corresponding to the round of the protocol). These instances are invoked sequentially, and thus the timelocks have to be set accordingly. Recall that at the end of the reveal phase of every instance of the locked ladder mechanism, parties have either already been compensated, or they learn all the protocol messages for the round, and are all evened out wrt deposits. Then to apply the see-saw idea, we need to introduce new “chain” deposits between successive instances of the locked ladder mechanism.

**Chain deposits.**
- For $j = 2$ to $n$:
  $$P_j \xrightarrow{TT_{r+1}, q} P_1 \quad (TX_{1,j}^{chain})$$
- For $j = 2$ to $n$:
  $$P_1 \xrightarrow{TT_{r}, q} P_j \quad (TX_{1,j}^{chain})$$

**Bootstraping.** Finally we focus on how to incentivize $P_1$ to start the protocol (i.e., reveal $T_{1,1}$) or otherwise pay penalty. To do this, we make use of “dummy tokens” $\{U_{1,j}\}_{j \in [n]}$. These dummy tokens are obtained by the parties via an initial secure computation step. In more detail, for all $j \in [n]$, the secure computation protocol:
- chooses $U_{1,j} \leftarrow \{0, 1\}^\lambda$ at random;
- computes $com_{boot}^{1,j} \leftarrow S(\lambda_j, U_{1,j}, \omega_{1,j}^{boot})$;
- outputs $com_{boot}^{1,j}$ and $U_{1,j}$ to each $P_j$,

where $S$ is the sender algorithm of a honest binding commitment scheme (c.f. Appendix A). Note that $com_{boot}^{1,j}$ is computed in order to allow parties to generate the verification circuit for transaction $TX_{1,j}^{boot}$ described below. Also, we stress that for $j ≠ 1$, the dummy token $U_{1,j}$ is unknown to $P_1$; it only knows the corresponding commitment $com_{1,j}^{boot}$. (We note that the above secure computation step can be combined with the secure computation step for handling the cash deposit step described at the end of Section IV as well as for generating the dummy tokens $\{U_{i,j}\}$ in the lock mechanism.) Consider the following set of deposit transactions where $\tau_1 > \tau_0$:

**Bootstrap deposits.**
- For $j = 2$ to $n$:
  $$P_j \xrightarrow{T_{i} \land U_{1,j}, q} P_1 \quad (TX_{j,1}^{boot})$$
- For $j = 2$ to $n$:
  $$P_1 \xrightarrow{U_{1,j}, q} P_j \quad (TX_{1,j}^{boot})$$

That is, first each $P_j$ makes a deposit $TX_{j,1}^{boot}$ to $P_1$, and then $P_1$ makes deposits $TX_{1,j}^{boot}$ to each $P_j$. Then in the reveal phase, the claims are made in reverse: each $P_j$ first claims $TX_{j,1}^{boot}$ using the dummy token $U_{1,j}$. Now $P_1$ learns $U_{1,j}$, and since it already knows $T_1$, it can go ahead and claim each $TX_{j,1}^{boot}$. More importantly, note that once the bootstrap deposits are made, an honest $P_j$ will
always claim $Tx_{i,j}^{\text{boot}}$, and thus will be $\text{coins}(q)$ up. Thus the onus is on $P_1$ to deliver the first token (and to reclaim its $\text{coins}(q)$), failing which it effectively pays a penalty $\text{coins}(q)$ to each honest party. On the other hand if not all deposits are not made, then an honest $P_2$ does claim $Tx_{i,j}^{\text{boot}}$ (if made), which in turn ensures that $P_1$ does not yet know $U_{i,j}$, and cannot claim $Tx_{j,i}^{\text{boot}}$. In such a situation, the protocol either terminates (without even going past the deposit step—thus there is no requirement to penalize any party), or $P_1$ reveals $T_1$ by claiming some $Tx_{i,j}^{\text{boot}}$ where $P_j$ is also malicious. In the latter case, parties just move on to the first stage of the computation.

We note that this bootstrapping step is common to all our constructions in this section. Indeed the bootstrap deposits will be the last deposits to be made in the initial deposit phase, and will be the first deposits to be claimed in the reveal phase. Since the ideal oblivious transfer primitive $\mathcal{F}_{\text{OT}}$ is sufficient to obtain a common random string, we can then apply known constructions (e.g., [61]) to obtain the following theorem.

**Theorem 2.** Let $(F, d^*)$ be a bounded zero-sum reactive distribution as in Definition 2. Then assuming the existence of enhanced trapdoor permutations, there exists a protocol that SCC realizes (cf. Definition 1) $\mathcal{F}_{F,d^*}$ in the $(\mathcal{F}_{\text{OT}}, \mathcal{F}_{\text{CR}})$-hybrid model.

**Proof sketch.** The main idea behind the proof is that the witnesses used to claim $\mathcal{F}_{\text{CR}}$ transactions are simply successive messages of a secure computation protocol $\pi_F$ that realizes that standard reactive functionality $\mathcal{F}_F$. Since $\pi_F$ is secure by definition, we have that the computation also proceeds securely. To do the simulation, we make use of (1) the simulator for $\pi_F$, (2) the simulator for initial secure computation step (alternatively access to ideal unfair functionality) that generates the dummy tokens for the lock mechanism, the bootstrap deposits (i.e., the values $\{U_{i,j}\}$), and also for the cash deposits at the end (i.e., the values $\{T_{i,j,k}^{\text{fin}}\}$), and (3) the simulator algorithms for honest-binding commitments. Simulating the coins part of the protocol is more involved but closely follows the simulation of the ladder mechanism in [26]. We defer a full proof to the full version.

### VI. EFFICIENT PROTOCOLS FOR POKER

Recall that in Section III we showed how to obtain a protocol for poker using $\mathcal{F}_{F,d^*}$. Thus using the construction described in Sections IV and V one can readily construct a protocol for poker in the $\mathcal{F}_{\text{CR}}$-hybrid model. However these protocols can be quite expensive as they make white-box use of a protocol for secure computation. Additionally it might be the case that for some applications we may be able to apply specific optimizations. Indeed in this section we describe an optimized protocol for Texas hold ’em [31].

Our key observation is that in the protocol for poker outlined in Section III only player $P_1$ has an input in a stage of the computation that corresponds to player $P_2$’s $r$-th round move. Thus, there is hope to altogether avoid the secure computation step required for that stage. Indeed, we show how to do this by letting the $\mathcal{F}_{\text{CR}}$ script check the validity of the transcript of bets, thus making the secure computation step redundant. While this is the main idea, there are a number of additional difficulties that need to be handled including revealing hidden community cards at different stages of the hand. To counter these, we design the transactions a way that (1) forces a party to make a move during its turn or otherwise pay a penalty, and (2) allows the next party to obtain sufficient information that enables it to decide on its own move (this information includes unveiling a subset the community cards). To re-iterate, such a design is vital for two reasons: (a) it avoids use of expensive secure computation protocols (e.g., [61]) between moves, while (b) allowing us to make direct use of the see-saw transaction mechanism to incentivize participation in the protocol. An additional (generic) optimization involves skipping the “bootstrapping” deposits altogether since if party $P_1$ does not send the first message we are happy with an abort since no party could have learned anything.

We now describe our optimized protocol in more detail. Let $(S, R)$ be a non-interactive honest binding commitment (cf. Definition 3). In an initial secure computation step, parties interact with an unfair ideal functionality $\mathcal{F}_f$ (alternatively run a secure computation protocol realizing $\mathcal{F}_f$) that does the following:

- selects hands $h_i$ (i.e., consisting of two cards) uniformly at random for each party $P_i$, as well as the five community cards $y_1, \ldots, y_5$;
- performs an $n$-out-of-$n$ secret sharing of each hand $h_i$ to obtain $\{h_{i,j}\}_{j \in [n]}$, and a $n$-out-of-$n$ secret sharing of each of the five cards $y_k$ to obtain $\{y_{k,j}\}_{j \in [n]}$;
- applies the sender algorithm of an honest-binding commitment using random $\omega_{i,j}$ to secret share $h_{i,j}$ to obtain $\text{com}^{h_{i,j}}$ and $\text{Tag}^{h_{i,j}}_{i,j}$ and $\text{Token}^{h_{i,j}}_{i,j} = (h_{i,j}, \omega_{i,j})$;
- applies the sender algorithm of an honest-binding commitment using random $\omega_{i,j}$ to secret share $y_{i,j}$ to obtain $\text{com}^{y_{i,j}}$ and $\text{Tag}^{y_{i,j}}_{i,j}$ and $\text{Token}^{y_{i,j}}_{i,j} = (y_{i,j}, \omega_{i,j})$;
- sets $\text{AllTags} = \{\text{Tag}^{h_{i,j}}_{i,j}, \text{Tag}^{y_{i,j}}_{i,j}\}_{i,j \in [n]}$; and
- for each $i \in [n]$, delivers $\text{AllTags}$, $\{\text{Token}^{h_{i,j}}_{i,j}, \text{Token}^{y_{i,j}}_{i,j}\}_{j \in [n]}$ to $P_i$.

That is, in the first step, parties run a standard secure computation (i.e., with unfair aborts) that generates the private hands for each party as well as the community cards. However none of these cards are delivered to the parties. Instead all of these cards (including each party’s
hands) are simply secret shared among the parties. In addition, parties also receive (honest-binding) commitments on all the shares, and the decommitments to the shares held by them. These are given so that parties can later verify if each party indeed reveals the correct shares by sending the decommitments corresponding to the public commitments.

Once this is done, parties make a series of deposits as in the see-saw (alternatively, locked ladder) mechanism. We defer the description of the $\phi_{i,j}$ for these deposits, and first focus on the structure of the protocol. Each party $P_j$ is first required to reveal $H_j = \{h_{i,j}\}_{i \in [n] \setminus \{j\}}$, i.e., the secret shares of other party’s hands. This is so that each party learns its private hands. Here we will make use of the see-saw mechanism to ensure that each party $P_j$ either reveals $H_j$ or pays a penalty to all other parties. The verification circuits for the $\mathcal{F}_{CR}$ transactions will depend on $\text{com}^i_{k,j}$ generated in the initial secure computation step.

Next parties enter a round of (pre-flop) betting. Here we assume a bound on maximum number of stages of betting (this is so that we can ensure that parties make all the necessary $\mathcal{F}_{CR}$ deposits in the see-saw mechanism). To place a bet, party $P_i$ simply sends the entire transcript of bets made so far in this round along with its new bet. Note that each party signs its bet when it makes one, and thus when parties send a transcript containing the bets, they must also contain the necessary signatures. We assume that there is a well-defined function $\text{tv}_{i,j} (\text{tv stands for “transcript validity”})$ that takes the transcript of the protocol so far (including bets made so far, and the new bet made by party $P_i$ in round $r$), and verifies if it is a valid bet. Note that a bet $b_i$ made by $P_i$ simply specifies the additional amount of coins it is willing to bet during its turn in pre-flop betting round. (Similarly to fold, $P_i$ simply sends a signed “fold” message.) We wish to stress that no actual coins related to the bet amounts are transferred in this phase. (These will all be transferred at the very end of the protocol.)

Now note that once this round of betting ends, the flops needs to be revealed to all the parties. We adopt the same strategy that we used to reveal each party’s hands. That is, each party $P_j$ is required to reveal $Y^3_j = \{y_{1,j}, y_{2,j}, y_{3,j}\}$, i.e., the secret shares of the flop. Once again we will make use of the see-saw mechanism to ensure that each party $P_j$ either reveals $Y^3_j$ or pays a penalty to all other parties.

Two additional rounds of betting take place before revealing the turn and the river. These are handled exactly like the pre-flop betting. Once all the community cards are revealed, parties that wish to claim the pot start revealing their cards. That is, parties execute an additional stage where they take turns to reveal their cards, i.e., reveal their share $h_{i,j}$ (which reveals their hand). Once all parties complete the showdown round, and the entire transcript $\text{TT}_{m,n}$ is available, then the pot winner can easily be determined. Note that we run only one secure computations step at the very beginning, and $\text{AllTags}$ generated in this step is sufficient to design the verification circuits for all $\mathcal{F}_{CR}$ deposits in the see-saw mechanism. Since the see-saw mechanism now applies, any party that aborts the protocol before the winner has been determined will pay a penalty to all other parties.

The above description turns out to be sufficient to realize “mental poker” $\Pi$, but is not sufficient to realize standard poker (i.e., poker with money). This is because we still haven’t let the winner(s) take the pot. Next we describe the cash distribution stage. Let $d^*) = (d_1^*, \ldots, d_n^*)$, and for each $i \in [n]$, let $m_i = [\log(d_i^*)]$. As in Section IV, for every ordered pair $(i, j)$ with $i, j \in [n]$ and $i \neq j$, and for each $k \in [m_i]$, we let $P_i$ make an $\mathcal{F}_{CR}$ transaction as follows (we slightly abuse the $\mathcal{F}_{CR}$ notation and use the verification circuit instead of the verifying witness):

$$P_i \xrightarrow{\phi_{i,j,k}^{\text{fin}}} P_j \quad (\text{TV}_{m,n}^\text{fin})$$

where verification circuit $\phi_{i,j,k}^{\text{fin}}$ takes $\text{TT}_{m,n}$ as input and:

- outputs 0 and terminates if $\text{tv}_{m,n}(\text{TT}_{m,n}) = 0$;
- computes $z^* = (z_1^*, \ldots, z_n^*)$ using $\text{TT}_{m,n}$, where $z_i^*$ represents the amount which party $P_i$ is supposed to get at the end of the protocol;
- computes $A = g(d^*, z^*)$ (cf. Observation $\Pi$), lets $a_{i,j}$ denote the $(i,j)$-th entry of matrix $A$, and lets $b_{i,j,1}^*, \ldots, b_{i,j,m_i}^*$ be the binary representation of $a_{i,j}$;
- outputs 1 if $b_{i,j,m_i}^* = 1$, else outputs 0.

**Efficiency.** Note that each party $P_i$ makes $(n - 1) \cdot m_i$ calls to $\mathcal{F}_{CR}$ and deposits a total of $(n - 1) \cdot d_i^*$ coins. For implementing the see-saw mechanism we require $O(n^2 m)$ calls to $\mathcal{F}_{CR}$ and each party to make a maximum deposit of $O(n m)$ where $m$ represents the bound on the maximum number of betting rounds in a hand. Note that we can preprocess both the secure computation, as well as the initial deposit phase (thus managing the long waiting times for transaction confirmation offline). Other than this, note that the messages in our secure poker protocol are mostly signed messages indicating the player’s move, and thus not very different from the messages in an insecure poker protocol.

**VII. Conclusions**

In this paper, we introduced and designed efficient protocols for secure cash distribution, a powerful primitive that provides a strong notion of dropout tolerance, and suffices to capture multiplayer games such as poker. We leave a full-scale implementation of our optimized protocol for future work.
Equivocation implies the standard hiding property, namely, that for all PPT algorithms $A$ (that maintain state throughout their execution) the following is negligible:

$$\Pr \left[ \left( m_0, m_1 \right) \leftarrow A(1^\lambda); \ b \leftarrow \{0, 1\}; \ com \leftarrow S(1^\lambda, m_b) : A(com) = b \right) \right] - \frac{1}{2}.$$ 

We observe that if $(com, \omega)$ are generated by $(\tilde{S}_1, \tilde{S}_2)$ for some message $m$ as in the definition above, then binding still holds: namely, no PPT adversary can find $(m', \omega')$ with $m' \neq m$ such that $R(m', com, \omega') = 1$.

As observed in [26], we can construct highly efficient heuristically secure honest binding commitment schemes in the programable random oracle model. In the following let Hash be a programmable hash function, and let $\omega \in \{0, 1\}^\lambda$. We only describe the algorithms $S, R$. The algorithms $\tilde{S}_1, \tilde{S}_2$ are obtained by standard programming techniques.

$$S(1^\lambda, m; \omega)$$

$$\text{return com := Hash}(m||\omega);$$

$$R(m, com, \omega)$$

- If $com = \text{Hash}(m||\omega)$
  - return 1;
- else return 0;

Next we provide more details on the ideal functionality for secure computation with penalties denoted $F^*_f$.

**Ideal functionality $F^*_f$ [26], [27]**. In the first phase, the functionality $F^*_f$ receives inputs for $f$ from all parties. In addition, $F^*_f$ allows the ideal world adversary $S$ to deposit some coins which may be used to compensate honest parties if $S$ aborts after receiving the outputs. Note that an honest party makes a fixed deposit $d$ in the input phase. Then, in the output phase, $F^*_f$ returns the deposit made by honest parties back to them. If $S$ deposited sufficient number of coins, then it gets a chance to look at the output and then decide to continue delivering output to all parties, or just abort, in which case all honest parties are compensated using the penalty deposited by $S$. We note that the version of $F^*_f$ in [27] varies slightly from the one proposed in [26]. While [26] allowed $S$ to deposit insufficient number of coins (i.e., less than $hd$), [27] do not. On the other hand, [27] do allow $S$ to send extra coins to honest parties when it aborts. For our purposes we can use either variant (and make the corresponding changes) since we will be using $F^*_f$ as a black-box. Just to keep things concrete, we follow the definition given in [27] which we reproduce in Figure 5.

**B. Secure Cash Distribution With an Honest Majority**

Boiled down, the difficulty in constructing protocols for secure cash distribution is the issue of *fairness*: 

---

A. More Preliminaries

**Honest binding commitments.** We include the definition from [22].

**Definition 3.** A (non-interactive) commitment scheme for message space $\{M_\lambda\}$ is a pair of PPT algorithms $S, R$ such that for all $\lambda \in \mathbb{N}$, all messages $m \in M_\lambda$, and all random coins $\omega$ it holds that $R(m, S(1^\lambda, m; \omega), \omega) = 1$.

A commitment scheme for message space $\{M_\lambda\}$ is honest-binding if it satisfies the following:

**Binding (for an honest sender)** For all PPT algorithms $A$ (that maintain state throughout their execution), the following is negligible in $\lambda$:

$$\Pr \left[ \left( m \leftarrow A(1^\lambda); \ \omega \leftarrow \{0, 1\}^*; \ com \leftarrow S(1^\lambda, m; \omega); \ (m', \omega') \leftarrow A(\omega, com') : \ R(m', com', \omega') = 1 \land m' \neq m \right) \right]$$

**Equivocation** There is an algorithm $\tilde{S} = (\tilde{S}_1, \tilde{S}_2)$ such that for all PPT $A$ (that maintain state throughout their execution) the following is negligible:

$$\Pr \left[ \left( m \leftarrow A(1^\lambda); \ \omega \leftarrow \{0, 1\}^*; \ com \leftarrow S(1^\lambda, m; \omega); \ A(1^\lambda, com, \omega) = 1 \right) \right]$$

$$- \Pr \left[ \left( com, st \leftarrow S(1^\lambda); \ m \leftarrow A(1^\lambda); \ \omega \leftarrow \tilde{S}_2(st, m); \ A(1^\lambda, com, \omega) = 1 \right) \right]$$
### Security Proof Sketch

The proof of security is straightforward, and directly relies on the guaranteed output delivery of the secure computation protocol and the threshold properties of the fund controlling mechanism and the secret-sharing scheme:

1. **If any party aborts during the deposit phase, the remaining honest parties suffice to control the funds committed so far, hence they can issue refunds even without the cooperation of the malicious parties.**

2. **If the deposit phase completed successfully, all parties have committed the required funds. A party that aborts in any of the subsequent phases can no longer prevent the computation from proceeding:** the honest parties hold $t + 1$ shares of the state at every step in the computation phase, so the secure
computation protocol can always reconstruct the state correctly, and the secure computation protocol itself can always execute to completion due to the guaranteed output delivery property.

- Finally, in the distribution phase the honest parties suffice to control the committed funds, so the cooperation of the malicious parties is not required.

D. Extensions

We note that in this model, the protocol can easily be extended to support an a-priori unknown number of rounds (the output of the function $f_r$ can specify whether or not to continue to the next round), as well as coin distribution in intermediate rounds, rather than just at the end (as long as a bound on the total number of distributed coins is known—the initial deposit must be larger than this bound).