How to Use Bitcoin to Play Decentralized Poker

Ranjit Kumaresan
MIT
ranjit@csail.mit.edu

Tal Moran
IDC Herzliya
talm@idc.ac.il

Iddo Bentov
Technion
idddo@cs.technion.ac.il

ABSTRACT

Back and Bentov (arXiv 2014) and Andrychowicz *et al.* (Security and Privacy 2014) introduced techniques to perform secure multiparty computations on Bitcoin. Among other things, these works constructed lottery protocols that ensure that any party that aborts after learning the outcome pays a monetary penalty to all other parties. Following this, Andrychowicz *et al.* (Bitcoin Workshop 2014) and concurrently Bentov and Kumaresan (Crypto 2014) extended the solution to arbitrary secure function evaluation while guaranteeing fairness in the following sense: any party that aborts after learning the output pays a monetary penalty to all parties that did not learn the output. Andrychowicz *et al.* (Bitcoin Workshop 2014) also suggested extending to scenarios where parties receive a payoff according to the output of a secure function evaluation, and outlined a 2-party protocol for the same that in addition satisfies the notion of fairness described above.

In this work, we formalize, generalize, and construct multiparty protocols for the primitive suggested by Andrychowicz *et al.* We call this primitive *secure cash distribution with penalties*. Our formulation of secure cash distribution with penalties poses it as a multistage *reactive* functionality (i.e., more general than secure function evaluation) that provides a way to securely implement smart contracts in a decentralized setting, and consequently suffices to capture a wide variety of stateful computations involving data and/or money, such as decentralized auctions, markets, and games such as poker, etc. Our protocol realizing secure cash distribution with penalties works in a hybrid model where parties have access to a claim-or-refund transaction functionality $\mathcal{F}_{\mathrm{CR}}^{\star}$ which can be efficiently realized in (a variant of) Bitcoin, and is otherwise independent of the Bitcoin ecosystem. We emphasize that our protocol is *dropout-tolerant* in the sense that any party that drops out during the protocol is forced to pay a monetary penalty to all other parties. Our formalization and construction generalize both *secure computation with penalties* of Bentov and Kumaresan (Crypto 2014), and *secure lottery with penalties* of Andrychowicz *et al.* (Security and Privacy 2014).

CCS'15, October 12-16, 2015, Denver, CO, USA.

http://dx.doi.org/10.1145/2810103.2813712.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

Keywords

Secure Computation; Bitcoin; Smart Contracts; Markets; Poker.

1. 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* (MPC), 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 [2]. Indeed mental poker has since been used as a metaphor for MPC [3]. Clearly, MPC 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.

The obvious problem is that secure computation as defined can only allow players to play mental poker (i.e., without involving real money). 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. 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. As [4] points out, an even more fundamental issue is to get parties to respect the outcome of the protocol and distribute the money as dictated by the output.

Tying payments to secure computation. More generally, there are many cases in which we would like to tie real-world payments to secure computation, e.g., decentralized fair exchange of digital goods or services for money in online marketplaces, decentralized multistage auctions, decentralized online gambling, etc. Currently, these tasks are delegated to a trusted third party (such as a bank, escrow service, or a court system). For "traditional" currency systems, any payment—whether or not it is based on secure computation— requires trust in a third party (as the currency itself is based on a trusted party, such as a central bank). However, the introduction of cryptocurrencies, such as Bitcoin [5], opens the possibility of handling payments in a decentralized manner [6, 7].

Indeed cryptocurrencies are a natural choice for combining MPC with "real money." Andrychowicz *et al.* [4] and Back and Bentov [8] introduced techniques to perform secure multiparty computations on Bitcoin. Among other things, these works constructed lottery protocols that ensure that any party that aborts after learning the outcome pays a monetary penalty to all other parties. Following this, Andrychowicz *et al.* [9] and concurrently Bentov and Kumaresan [10] extended the solution to arbitrary secure function evaluation while guaranteeing fairness in the following sense: any party that aborts after learning the output pays a monetary penalty to all parties that did not learn the output. Andrychowicz *et al.* [9] also suggested extending to scenarios where parties receive a payoff according to the output of a secure function evaluation, and outlined a 2-party protocol for the same that in addition satisfies the notion of fairness described above.

Our contributions in a nutshell. In this work, we formalize, generalize, and construct multiparty protocols for the primitive suggested by [9]. We call this primitive *secure cash distribution with penalties*. Our formulation of secure cash distribution with penalties poses it as a multistage *reactive* functionality (i.e., more general than secure function evaluation) that suffices to capture a wide variety of stateful computations involving data and/or money, such as *decentralized* auctions, games, markets, etc. Our protocol realizing secure cash distribution with penalties works in a hybrid model where parties have access to a claim-or-refund transaction functionality $\mathcal{F}_{\mathrm{CR}}^{\star}$ which can be efficiently realized in (a variant of) Bitcoin, and is otherwise independent of the Bitcoin ecosystem. We emphasize that our protocol is *dropout-tolerant* in the sense that any party that drops out during the protocol is forced to pay a monetary penalty to all other parties. Our formalization and construction simultaneously generalize *secure computation with penalties* of Bentov and Kumaresan [10], and *secure lottery with penalties* of Andrychowicz *et al.* [4]. Below we describe our contributions in more detail.

Defining SCD. We define SCD as a *bounded reactive* functionality, i.e., the computation proceeds in a finite number of stages. In an initial "deposit" stage, parties deposit sums of money. In each succeeding stage, parties provide inputs and obtain outputs for that stage. Then in the last stage, the money deposited by the parties is redistributed among them according to the output of the last stage. Any party that aborts during any stage of the computation will be forced to pay penalties to all parties. Thus SCD guarantees that honest parties either complete the entire computation or are compensated financially.

Implementing SCD. Note that while in the standard setting, reactive secure computation reduces to non-reactive secure computation by secret sharing the state between successive stages, a similar reduction does not carry over when we are in the penalty setting since a malicious party may abort *between* successive stages of a reactive computation and go unpenalized. We design a protocol that realizes SCD (i.e., with full *simulation security* [11]) in a hybrid model where parties have access to a claim-or-refund transaction functionality $\mathcal{F}_{\mathrm{CR}}^{\star}$. The main technical idea in our solution is the construction of a *see-saw transaction mechanism* which is a novel extension of the ladder transaction mechanism of [10]. Loosely speaking, the ladder mechanism implements fair exchange with penalties in the following sense: each party has their (digital) item at the beginning of the protocol, and at the end if one party receives all items, then it pays a penalty to parties which have not received all items. In contrast the see-saw mechanism implements the following variant of fair exchange with penalties: the exchange proceeds in multiple rounds, and in each round, parties can *adaptively* choose their input item they want exchanged based on the items put up for exchange by other parties in previous rounds. Penalties are now enforced across the entire exchange process. That is, if a party decides to terminate the exchange process, then it pays a penalty to all other parties. Note in particular that penalties are enforced even when no party receives all items. Contrast this with the ladder mechanism that enforces penalties to all parties only when some party received all items. See Section 5 for the implementation of the see-saw mechanism. Our protocol for secure cash distribution makes non-black-box use of an underlying MPC protocol (cf. Section 4).

Practical applications. Consider a group of servers that agree to carry out an intensive computation task that spans several days. Furthermore, assume that the computation requires multiple rounds of interactions and the full participation of all participating servers, and otherwise fails. Here, we would like to guarantee that the servers exchange information as agreed upon without defaulting. In such a setting, it is critical to ensure that the computation is carried out as intended, and that no server invests computational effort only to learn that a different server abruptly decided to not continue the computation any more. Observe that the problem description as is does not involve money. Still our formulation of SCD allows us to capture such a setting and offers a meaningful solution to this problem, namely that a defaulting server will be forced to pay a penalty to everyone else. Such a solution can be achieved by a straightforward use of a verifiable computation scheme in combination with our see-saw transaction mechanism.

Next, consider a group of agents who participate in a set of financial transactions over the internet. For example, these could be agents in a prediction market (possibly with dark pool trading capabilities) who place bets on the occurrence of sets of events, and may adaptively vary their choices depending on whether a previous event in the set happened or not. One must also consider what happens when a malicious agent stops participating during the process. A naïve solution would require that the agents make a deposit at the beginning of the protocol which they would forfeit when they abort. To make this idea work in a decentralized setting, one must develop a method to put the deposits in escrow, and make sure that in the event of an abort (1) honest agents can always retrieve their deposits from the escrow, and (2) honest agents obtain penalties from the escrow when a dishonest agent aborts. Implementing such a decentralized escrow when a majority of agents are dishonest is not straightforward. Our formulation of SCD exactly allows the capability to maintain a decentralized escrow across multiple stages of a computation and hence our protocol implementing SCD provides a solution to the prediction market problem described above.

More generally, since SCD models stateful reactive functionalities it provides a way to securely implement *smart contracts* in a decentralized setting, and consequently captures a wide variety of games, including poker (assuming that the strategy space of the players includes variables that cannot be clearly defined and may depend on side information that cannot be completely captured).

Limitations. Note that the plain model realizations of $\mathcal{F}_{\mathrm{CR}}^{\star}$ rely on Bitcoin scripts. While we explicitly specify the checks that the scripts need to perform, the current Bitcoin scripting language is quite conservative (many opcodes became blacklisted [12]), and therefore some of the required checks are not currently supported in Bitcoin. More concretely, our construction requires signature verification of arbitrary messages (i.e., not more burdensome than the supported signature verification for the entire transaction data). In addition it requires scripts to support calculations whose complexity depends on the specific application. For instance, in the application to poker, we require Bitcoin scripts to support simple

arithmetic calculations that verify whether a transcript of a poker protocol follows the rules of poker. In the most general setting, the *validation complexity* [13] (which corresponds to the complexity of script verification) equals the complexity of verifying validity of partial transcripts of an underlying secure computation protocol that realizes the reactive functionality. As suggested in [13], validation complexity may also accurately reflect additional transaction fees that may be levied to include "unordinary" transactions (i.e., transactions of the kind that our constructions need) into the blockchain. Currently, only a small fraction of miners (e.g., *Eligius* mining pool) accept transactions that make use of the entire Bitcoin scripting language. In any case, our constructions require that new opcodes be added to the Bitcoin scripting language (e.g., the opcode mentioned above for verifying signatures of arbitrary messages). While we expect Bitcoin to be less conservative in the scripts it supports in the future, our protocols can be deployed on alt-coins with Turing complete scripts. However, Turing complete scripts are an overkill for our constructions. This is because the number of rounds until the final cash distribution must be bounded (cf. Section 4), hence miners can levy a suitable transaction fee by easily assessing the verification complexity of a certain SCD (e.g., poker of some fixed number of rounds) script. By contrast, a full-fledged Turing-complete cryptocurrency (like the Ethereum project) has to resort to extra mechanisms in order to protect itself from DoS attacks [14].

Our main goal in this work is to show feasibility of realizing SCD. As mentioned earlier, our SCD protocol makes non-black-box use of an underlying MPC protocol and can be inefficient in practice. We note that this limitation can be removed for some applications. For example, in Section 6, we show how to obtain a protocol for decentralized poker with dropout tolerance that makes only black-box use of MPC.

Related work. The general problem of secure computation was solved in the 2-party setting by Yao [15], and in the multiparty setting in [3]. Besides not handling payments, none of the schemes above can guarantee fairness in the presence of a dishonest majority [16]. [4] designed a multiparty lottery protocol in the penalty model. [9] designed 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 [10]. [13] shows applications of Bitcoin to various other interesting cryptographic primitives. The work of [17] also shows how to enforce smart contracts with financial privacy but with different trust assumptions.

Relation to [9]. The extension to a setting where payoffs depend on the output of a secure function evaluation was proposed in [9, Section 6]. The authors then show how to modify the Bitcoin scripts that implements two-party secure function evaluation with penalties and obtain a solution for the extended setting with payoffs. We emphasize that [9] handles the two-party case, with a non-reactive functionality, and outlines a solution using ad-hoc Bitcoin transactions. In contrast, we provide formal definitions for secure cash distribution with penalties which we define as a stateful reactive functionality, then construct a *multiparty* protocol that securely realize this definition, and provide formal proofs. Furthermore, our protocol works in a clean hybrid model where parties have access to a claim-or-refund transaction functionality $\mathcal{F}_{\mathrm{CR}}^{\star}$ and is otherwise independent of the Bitcoin ecosystem. That is, our protocol can be easily adapted to any setting (e.g., alt-coins, PayPal) that can support an implementation of $\mathcal{F}_{\mathrm{CR}}^{\star}$. In a sense, our work shows that $\mathcal{F}_{\mathrm{CR}}^{\star}$ is a *complete primitive* for secure computations involving money. Finally, a technique that we use in our cash distribution mechanism was previously outlined in [9]. Specifically, we use the idea from [9] that allows the parties to transfer arbitrary amounts of money by dividing a large amount into "power of 2 fractions." Note that it is possible to replace this technique with a naïve mechanism and still obtain our feasibility results.

2. PRELIMINARIES

A function $\mu(\cdot)$ is negligible in λ if for every positive polynomial $p(\cdot)$ and all sufficiently large λ's it holds that $\mu(\lambda) < 1/p(\lambda)$. A probability ensemble $X = \{X(a,\lambda)\}_{a\in\{0,1\}^*,n\in\mathbb{N}}$ 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 $X \stackrel{c}{\equiv} Y$ 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 λ. We prove security in the "secure computation with coins" (SCC) model proposed in [10]. Note that the main difference from standard definitions of secure computation [18] is that now the view of $\mathcal{Z}$ contains the distribution of coins. Let $\mathrm{IDEAL}_{f,\mathcal{S},\mathcal{Z}}(\lambda, z)$ denote the output of environment $\mathcal{Z}$ initialized with input z after interacting in the ideal process with ideal process adversary $\mathcal{S}$ and (standard or special) ideal functionality $\mathcal{G}_f$ on security parameter λ. Recall that our protocols will be run in a hybrid model where parties will have access to a (standard or special) ideal functionality $\mathcal{G}_g$. We denote the output of $\mathcal{Z}$ after interacting in an execution of π in such a model with $\mathcal{A}$ by $\mathrm{HYBRID}^{g}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$, where z denotes $\mathcal{Z}$'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 π be a probabilistic polynomial-time n-party protocol and let $\mathcal{G}_f$ be a probabilistic polynomial-time n-party (standard or special) ideal functionality. We say that π SCC realizes $\mathcal{G}_f$ with abort in the $\mathcal{G}_g$-hybrid model (where $\mathcal{G}_g$ is a standard or a special ideal functionality) if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ attacking π there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model such that for every non-uniform probabilistic polynomial-time adversary $\mathcal{Z}$,

$$\{\mathrm{IDEAL}_{f,\mathcal{S},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} \stackrel{c}{\equiv} \{\mathrm{HYBRID}^{g}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}. \quad \diamondsuit$$

Ideal functionality $\mathcal{F}_{\mathrm{CR}}^{\star}$ [10, 19, 20]. This special ideal functionality has been employed in the design of multiparty fair secure computation and lottery protocols [10]. See Figure 1 for a formal description. At a high level, $\mathcal{F}_{\mathrm{CR}}^{\star}$ allows a sender P_s to *conditionally* send $\mathsf{coins}(x)$ to a receiver P_r. The condition is formalized as the revelation of a satisfying assignment (i.e., witness) for a sender-specified circuit $\phi_{s,r}(\,\cdot\,; z)$ (i.e., relation) that may depend on some public input z. Further, there is a "time" bound, formalized as a round number τ, within which P_r has to act in order to claim the coins. An important property that we wish to stress is that the satisfying witness is made *public* by $\mathcal{F}_{\mathrm{CR}}^{\star}$. In the Bitcoin realization of $\mathcal{F}_{\mathrm{CR}}^{\star}$, sending a message with $\mathsf{coins}(x)$ corresponds to broadcasting a transaction to the Bitcoin network, and waiting according to some time parameter until there is enough confidence that the transaction will not be reversed.

$\mathcal{F}^{\star}_{\mathrm{CR}}$ with session identifier sid, running with parties P_s and P_r, a parameter 1^{λ}, and adversary $\mathcal{S}$ proceeds as follows:

- *Deposit phase.* Upon receiving the tuple $(\mathsf{deposit}, sid, ssid, s, r, \phi_{s,r}, \tau, \mathsf{coins}(x))$ from P_s, record the message $(\mathsf{deposit}, sid, ssid, s, r, \phi_{s,r}, \tau, x)$ and send it to all parties. Ignore any future deposit messages with the same $ssid$ from P_s to P_r.
- *Claim phase.* In round τ, upon receiving $(\mathsf{claim}, sid, ssid, s, r, \phi_{s,r}, \tau, x, w)$ from P_r, check if (1) a tuple $(\mathsf{deposit}, sid, ssid, s, r, \phi_{s,r}, \tau, x)$ was recorded, and (2) if $\phi_{s,r}(w) = 1$. If both checks pass, send $(\mathsf{claim}, sid, ssid, s, r, \phi_{s,r}, \tau, x, w)$ to all parties, send $(\mathsf{claim}, sid, ssid, s, r, \phi_{s,r}, \tau, \mathsf{coins}(x))$ to P_r, and delete the record $(\mathsf{deposit}, sid, ssid, s, r, \phi_{s,r}, \tau, x)$.
- *Refund phase:* In round $\tau + 1$, if the record $(\mathsf{deposit}, sid, ssid, s, r, \phi_{s,r}, \tau, x)$ was not deleted, then send $(\mathsf{refund}, sid, ssid, s, r, \phi_{s,r}, \tau, \mathsf{coins}(x))$ to P_s, and delete the record $(\mathsf{deposit}, sid, ssid, s, r, \phi_{s,r}, \tau, x)$.

Figure 1: The ideal functionality $\mathcal{F}^{\star}_{\mathrm{CR}}$.

3. 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. The deposited cash is then distributed back to the parties depending on (and along with) the output of the function evaluation. Any malicious party that aborts the protocol after learning output and/or receiving coins must pay a monetary penalty to all honest parties.

Clearly, such a primitive generalizes both secure computation with penalties [10, 9] and secure lottery with penalties [10, 4]. As it turns out, this informal definition of secure cash distribution is not strong enough to enable applications that we are interested in. 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 ρ 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 ℓ-th stage for $\ell \in [\rho]$, and let state_0 be initialized as $\mathsf{state}_0 := \mathsf{NULL}$. Then over the course of the computation, parties successively evaluate $f_\ell(x_\ell; \mathsf{state}_{\ell-1})$ to obtain $(z_\ell, \mathsf{state}_\ell)$ 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_ℓ represents the state saved for the $(\ell+1)$-th computation stage, and is kept private from the parties (via use of secret sharing).

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 the applications we are interested in.

We now proceed to the formal details. Let $d^{\star} = (d^{\star}_1, \ldots, d^{\star}_n)$ be the initial cash deposit from the parties, i.e., party P_i deposits $\mathsf{coins}(d^{\star}_i)$ into the computation. Then at the end of the protocol all the deposited coins, i.e., $\mathsf{coins}(\sum_{i\in[n]} d^{\star}_i)$, are distributed back to the parties according to the evaluation of the reactive functionality F on the parties' inputs. More precisely, let z_ρ denote the parties' output at the end of the last stage of the computation. We assume that z_ρ specifies how the coins are (re)distributed at the end of the entire computation. That is, we can parse $z_\rho = (z = (z_1, \ldots, z_n), z^{\star} = (z^{\star}_1, \ldots, z^{\star}_n))$ where z_i represents the parties' output, and $z^{\star}_i$ 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 $f_i : (\{0,1\}^*)^n \times \{0,1\}^* \to (\{0,1\}^*)^n \times \{0,1\}^*$ be a function. Let $d^{\star} = (d^{\star}_1, \ldots, d^{\star}_n) \in \mathbb{N}^n$ be a vector. We say that $(F = (f_1, \ldots, f_\rho), d^{\star})$ is a *bounded zero-sum reactive distribution* if $\forall\, x_1, \ldots, x_\rho \in (\{0,1\}^*)^n$ it holds that the value $z_\rho = ((z_1, \ldots, z_n), (z^{\star}_1, \ldots, z^{\star}_n)) \in (\{0,1\}^*)^n \times \mathbb{N}^n$ obtained from the sequence:

$$
\begin{aligned}
(z_1, \mathsf{state}_1) &\leftarrow f_1(x_1; \mathsf{NULL});\\
(z_2, \mathsf{state}_2) &\leftarrow f_2(x_2; \mathsf{state}_1);\\
&\vdots\\
(z_\rho, \mathsf{state}_\rho) &\leftarrow f_\rho(x_\rho; \mathsf{state}_{\rho-1}),
\end{aligned}
$$

satisfies $\sum_i z^{\star}_i = \sum_i d^{\star}_i$. ♢

Observation 1. The coins earned by P_i, namely $z^{\star}_i$ may be such that $z^{\star}_i > d^{\star}_i$ (e.g., when F represents the lottery functionality [4, 10]). To simplify exposition, we make use of a "helper" function g which on input $(d^{\star}, z^{\star})$ returns a matrix A whose (i,j)-th entry denoted $a_{i,j}$ specifies the amount of coins that need to be transferred from P_i to P_j. In particular, it must hold for all $i \in [n]$ that $\sum_{j\in[n]} a_{i,j} = d^{\star}_i$, and for all $j \in [n]$ that $\sum_{i\in[n]} a_{i,j} = z^{\star}_j$. Observe that it is easy to design g for a zero-sum distribution $(F, d^{\star})$.

Next we formally define $\mathcal{F}^{\star}_{F,d^{\star}}$ which idealizes SCD.

Ideal functionality $\mathcal{F}^{\star}_{F,d^{\star}}$. See Figure 2 for the formal defnition. In an initial cash deposit phase, the functionality $\mathcal{F}^{\star}_{F,d^{\star}}$ receives $\mathsf{coins}(d + d^{\star}_r)$ from each honest P_r, where d represents a parameterizable safety deposit and $d^{\star}_r$ represents the cash that will be stored in escrow. In addition, $\mathcal{F}^{\star}_{F,d^{\star}}$ allows the ideal world adversary $\mathcal{S}$ to deposit some coins which may be used to compensate honest parties if $\mathcal{S}$ aborts after receiving the outputs. If there is an abort at this stage, that is, $\mathcal{S}$ 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 ℓ-th stage of the computation phase, the honest parties supply their inputs to ℓ-th stage of the computation. The functionality then waits to receive corrupt parties' inputs for this stage. If $\mathcal{S}$ aborts at this stage, then the honest parties receive $\mathsf{coins}(q)$ penalty in addition to getting their deposit $\mathsf{coins}(d)$ back (and may also obtain some extra $\mathsf{coins}(q_r)$), and the computation phase is terminated. Now honest parties receive the amount that they deposited at the beginning of the protocol in the cash distribution phase. However, if $\mathcal{S}$ does continue (i.e., provide inputs to this stage), then the functionality computes the output of the ℓ-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.

Let $(F = (f_1, \ldots, f_n), d^\star = (d_1^\star, \ldots, d_n^\star))$ be a bounded zero-sum reactive distribution (cf. Definition 2). $\mathcal{F}^\star_{F,d^\star}$ with session identifier sid running with parties $P_1, \ldots, P_n$, a parameter 1^λ, and an ideal adversary $\mathcal{S}$ that corrupts parties $\{P_s\}_{s\in C}$ proceeds as follows: Let $H = [n] \setminus C$ and $h = |H|$. Let d represent the safety deposit, and let q denote the penalty amount. Initialize $\mathsf{state}_0 := \mathsf{NULL}$ and $\mathsf{flag} = 1$.

- *Deposit phase:* Wait to receive a message $(\mathsf{deposit}, sid, ssid, r, d^\star, \mathsf{coins}(d + d_r^\star))$ from P_r for all $r \in H$. Then wait to receive $(\mathsf{deposit}, sid, ssid, d^\star, \mathsf{coins}(hq + \sum_{s\in C} d_s^\star))$ from $\mathcal{S}$.
- *Computation phase:* For each $\ell = 1, \ldots, \rho$, do:
  - Wait to receive a message $(\mathsf{input}, sid, ssid, r, x_{\ell,r})$ from P_r for all $r \in H$.
  - If $\mathcal{S}$ sends $(\mathsf{abort}, sid, ssid, \{\mathsf{coins}(q_r)\}_{r\in H})$, send $(\mathsf{penalty}, sid, ssid, \mathsf{coins}(q + q_r))$ to P_r for all $r \in H$, send $(\mathsf{payback}, sid, ssid, \mathsf{coins}(\sum_{s\in C} d_s^\star - \sum_{r\in H} q_r))$ to $\mathcal{S}$, set $\mathsf{flag} = 0$, and terminate phase.
  - Else if $\mathcal{S}$ sends $(\mathsf{input}, sid, ssid, \{x_{\ell,s}\}_{s\in C})$, set $x_\ell = (x_{\ell,1}, \ldots, x_{\ell,n})$.
  - Compute $(z_\ell, \mathsf{state}_\ell) \leftarrow f_\ell(x_\ell; \mathsf{state}_{\ell-1})$, and parse z_ℓ to obtain $(z_{\ell,1}, \ldots, z_{\ell,n})$.
  - Send $(\mathsf{output}, sid, ssid, \{z_{s,\ell}\}_{s\in C})$ to $\mathcal{S}$.
  - If $\mathcal{S}$ returns $(\mathsf{continue}, sid, ssid)$, then send $(\mathsf{output}, sid, ssid, z_{\ell,r})$ to P_r for all $r \in H$.
  - Else if $\mathcal{S}$ sends $(\mathsf{abort}, sid, ssid, \{\mathsf{coins}(q_r)\}_{r\in H})$, send $(\mathsf{penalty}, sid, ssid, \mathsf{coins}(q + q_r))$ to P_r for all $r \in H$, send $(\mathsf{payback}, sid, ssid, \mathsf{coins}(\sum_{s\in C} d_s^\star - \sum_{r\in H} q_r))$ to $\mathcal{S}$, set $\mathsf{flag} = 0$, and terminate phase.
- *Distribution phase:* If $\mathsf{flag} = 0$, send $(\mathsf{return}, sid, ssid, \mathsf{coins}(d + d_r^\star))$ to P_r for all $r \in H$, and terminate. Else, parse z_ρ to obtain $z^\star = (z_1^\star, \ldots, z_n^\star)$, and send $(\mathsf{pay}, sid, ssid, z^\star, \mathsf{coins}(d + z_r^\star))$ to P_r for all $r \in H$, and send $(\mathsf{pay}, sid, ssid, z^\star, \mathsf{coins}(hq + \sum_{s\in C} z_s^\star))$ to $\mathcal{S}$.

Figure 2: Secure cash distribution with penalties $\mathcal{F}^\star_{F,d^\star}$.

On the other hand, if $\mathcal{S}$ decides to abort, then the honest parties get compensated as before, i.e., with $\mathsf{coins}(d+d_r^\star+q+q_r)$ in total, and the protocol is terminated. The computation phase terminates after the ρ-th stage ends. Note that upon successful completion of the ρ-th stage, all parties receive their final outputs. After this, parties enter the cash distribution phase; the cash is distributed according to the output of the ρ-th stage, i.e., z_ρ. The functionality parses z_ρ to obtain $z^\star$ which dictates how the cash is distributed among the parties. Using $z^\star$, the functionality distributes the cash among the parties, and returns their original deposits as well. In addition, the functionality also sends the value $z^\star$ to all parties, i.e., the way the cash gets distributed at the end is not private.

How to use $\mathcal{F}^\star_{F,d^\star}$ to implement poker. We now describe a naïve implementation of how to play poker hand via $\mathcal{F}^\star_{F,d^\star}$. (In Section 6 we provide an optimized poker protocol.) 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 $\mathcal{F}^\star_{F,d^\star}$. 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 $\mathcal{F}^\star_{F,d^\star}$. That is player P_i picks and sends $\tilde{r}_i$ to $\mathcal{F}^\star_{F,d^\star}$. Then $\mathcal{F}^\star_{F,d^\star}$ computes $f_1(\tilde{r}_1, \ldots, \tilde{r}_n)$ in the following way: first compute $\tilde{r} = \bigoplus_{i=1}^n \tilde{r}_i$ (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 is 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 $\mathsf{coins}(q)$. 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, $\mathcal{F}^\star_{F,d^\star}$ 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.

4. REALIZING SCD

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 (with no coins, and unfair abort), denoted $\mathcal{F}_F$, to set things up such that the *see-saw transaction mechanism* of Section 5 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., $\rho = 1$ and $F = f$. Essentially 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. The extension to multiple stages is straightforward and we describe it later.

First let us set up some notation. We say $(r,i) > (r',i')$ iff either (1) $r > r'$, or (2) $r = r'$ and $i > i'$. For (r,i), let $\mathsf{pred}(r,i)$ 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, $\mathsf{pred}(r,i)$ is the "predecessor" of (r,i). Let π_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*.[1] Let $\mathtt{TT}^{\pi_f}_{r,i}$ denote the transcript of protocol π_f up until party P_i's message in the r-th round. Let $\mathsf{nmf}_{r,i}$ denote the *next message function* for party P_i in round r. The function $\mathsf{nmf}_{r,i}$ takes as input the actual input x_i, the private randomness of party P_i, denoted ω_i, and the public transcript seen so far, i.e., $\mathtt{TT}^{\pi_f}_{\mathsf{pred}(r,i)}$. In other words, we have that $\mathtt{TT}^{\pi_f}_{r,i} \leftarrow \mathsf{nmf}_{r,i}(\mathtt{TT}^{\pi_f}_{\mathsf{pred}(r,i)}; (x_i, \omega_i))$ (i.e., $\mathsf{nmf}_{r,i}$ outputs the entire transcript so far). Also, since all messages are public broadcasts, there exists a function $\mathsf{tv}^{\pi_f}_{r,i}$ which checks if a given transcript $\mathtt{TT}_{r,i}$ (that contains all messages until and including party P_i's message in round r) is valid or not. By definition, we have that $\mathsf{tv}^{\pi_f}_{r,i}(\mathtt{TT}^{\pi_f}_{r,i}) = 1$. For simplicity and wlog, we assume that all messages (i.e., transcripts) broadcasted are signed by the sending party. This implies that the function tv that checks validity of the transcript also checks for 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_1 to either reveal its first round message $\mathtt{TT}^{\pi_f}_{1,1}$ to all parties, or pay a penalty to all parties. If P_1 revealed $\mathtt{TT}^{\pi_f}_{1,1}$, then P_2 can apply $\mathsf{nmf}_{1,2}(\mathtt{TT}^{\pi_f}_{1,1}; (x_2, \omega_2))$ to obtain $\mathtt{TT}^{\pi_f}_{1,2}$. Now we want P_2 to either reveal $\mathtt{TT}^{\pi_f}_{1,2}$ 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 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 be handled.) Designing a *transaction mechanism* for implementing the above strategy is one of the main contributions in this paper. We defer the presentation of the transaction mechanism to Section 5, and devote the rest of this section to handling other issues.

Handling multiple valid transcripts. In an actual implementation of the above strategy in the $\mathcal{F}^{\star}_{\mathrm{CR}}$-hybrid model, we will have parties receive multiple $\mathcal{F}^{\star}_{\mathrm{CR}}$ transactions from other parties that can be claimed if they produce a valid transcript. It is possible that a malicious party may claim a subset of these $\mathcal{F}^{\star}_{\mathrm{CR}}$ transactions using one valid transcript and a different subset using a different valid transcript. Such an "attack" may indeed be possible by varying the actual input and private randomness input to the next message function. Indeed 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). 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. Note that a "proof" of any such violation is readily obtained from the inconsistent transcripts. We ask each party P_i to make $\mathcal{F}^{\star}_{\mathrm{CR}}$ transactions to every other party that can be claimed by revealing a proof of violation: i.e., pair of transcripts $T^{\mathsf{vio}}_i = (\mathtt{TT}_i, \mathtt{TT}'_i)$ such that for some $r \in [m]$, it holds that $\mathsf{tv}^{\pi_f}_{r,i}(\mathtt{TT}_i) = \mathsf{tv}^{\pi_f}_{r,i}(\mathtt{TT}'_i)$ and yet $\mathtt{TT}_i \neq \mathtt{TT}'_i$. Since transcripts are signed, a proof of violation against an honest party can never be obtained (except with negligible probability). Following the notation in [10], we use $P_1 \xrightarrow[q,\tau]{T} P_2$ to denote an $\mathcal{F}^{\star}_{\mathrm{CR}}$ transaction for $\mathsf{coins}(q)$ made by P_1 that can be claimed by P_2 if P_2 produces witness T within time τ. Thus to safeguard against violations we ask each P_i to make the following set of transactions for each $j \in [n] \setminus \{i\}$:

$$P_i \xrightarrow[n\cdot q,\tau]{T^{\mathsf{vio}}_i} P_j \qquad (\mathsf{Tx}^{\mathsf{vio}}_{i,j})$$

Here τ is such that the transaction can be claimed until the end of the protocol. Note that the transaction if claimed will transfer $\mathsf{coins}(n \cdot q)$ 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 $\mathsf{Tx}^{\mathsf{vio}}_{i,j}$ 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 $\mathsf{coins}((n-1)q)$ to the remaining parties. Thus upon any violation by malicious parties, we ensure that each honest P_j will still be $\mathsf{coins}(q)$ up at the end of the protocol execution.

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. 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 themselves. Adopting an idea from [9], we ask parties to commit to money transfers for all powers of 2 up to the maximum possible sum. In more detail, let $d^{\star} = (d^{\star}_1, \ldots, d^{\star}_n)$, and for each $i \in [n]$, let $m_i = \lceil \log(d^{\star}_i) \rceil$. 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 $\mathcal{F}^{\star}_{\mathrm{CR}}$ transaction as follows:

$$P_i \xrightarrow[2^k,\tau_{\mathsf{fin}}]{T^{\mathsf{fin}}_{i,j,k}} P_j \qquad (\mathsf{Tx}^{\mathsf{fin}}_{i,j,k})$$

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. That is, suppose the output of the reactive computation is $z_\rho = (z, z^{\star})$ with $z^{\star} = (z^{\star}_1, \ldots, z^{\star}_n)$, then we want P_j to obtain $\mathsf{coins}(z^{\star}_j)$ at the end of the protocol. In other words, we need to provide P_j with the right subset of $\{T^{\mathsf{fin}}_{i,j,k}\}_{i,k}$ that will allow it to claim exactly $\mathsf{coins}(z^{\star}_j)$. This subset will obviously need to be transferred in the last computation stage f_ρ. To make sure the deposits are made at the very beginning, the parties need to know the corresponding verification circuits $\phi^{\mathsf{fin}}_{i,j,k}$ at the beginning as well. To design the verification circuits, we employ *honest binding commitments* [22, 10] (See also Appendix A). Let (S, R) be a honest binding commitment scheme. (Note that such commitment schemes can be realized by cryptographic hash functions in the programmable random oracle

[1] That is, all messages exchanged in the protocol are simply broadcasts. Protocols secure against dishonest majority typically fall under this category. For an explicity example, see the main construction in [21, 3]. See also the discussion in [22].

model.) More precisely we require parties to execute a standard, secure-with-abort MPC protocol at the very beginning that for all $i \in [n], j \in [n] \setminus \{i\}, k \in [m_i]$:

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

The secret sharing is done so that parties can reconstruct the $T^{\mathsf{fin}}_{i,j,k}$ 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 $\phi^{\mathsf{fin}}_{i,j,k}$ to make the transaction $\mathsf{Tx}^{\mathsf{fin}}_{i,j,k}$. Next we describe the modification to the last stage. Instead of realizing f_ρ in the last stage, parties realize f'_ρ which:

- computes $z^\star = (z^\star_1, \ldots, z^\star_n)$ by invoking f_ρ;
- computes $A = g(d^\star, z^\star)$ (cf. Observation 1), let $a_{i,j}$ denote the (i,j)-th entry of matrix A, and let $b^\star_{i,j,1}, ..., b^\star_{i,j,m_i}$ be the binary representation of $a_{i,j}$;
- for all $i \in [n], j \in [n] \setminus \{i\}, k \in [m_i]$:
  - reconstructs $T^{\mathsf{fin}}_{i,j,k}$ (from $\mathsf{state}_{\rho-1}$);
  - outputs $T^{\mathsf{fin}}_{i,j,k}$ if $b^\star_{i,j,k} = 1$, else outputs 0.

Given the above it is easy to see that the set of transactions $\{\mathsf{Tx}^{\mathsf{fin}}_{i,j,k}\}$ transfer the right amounts of money according to the output $z^\star$. Next we show how to design the see-saw transaction mechanism that implements our strategy of forcing parties to send the next message of the protocol realizing $\mathcal{F}_F$.

5. SEE-SAW 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" $T_{1,1} = \mathrm{TT}^{\pi_f}_{1,1}$, then party P_2 computes (using $T_{1,1}$) and reveals token $T_{1,2} = \mathrm{TT}^{\pi_f}_{1,2}$, and so on until party P_n computes and reveals token $T_{1,n} = \mathrm{TT}^{\pi_f}_{1,n}$. (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 $T_{m,n} = \mathrm{TT}^{\pi_f}_{m,n}$. 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 $P_1, \ldots, P_{i-1}$, 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*. Any party that deviates during its turn in any of the m rounds in the reveal phase has to pay a penalty to all the remaining parties. Contrast this with the "ladder mechanism" of [10], where a party that aborts without learning the final output may not necessarily pay penalties to all parties.

Honest parties' strategy. As mentioned earlier, our protocol will be an ordered sequence of claim-or-refund transactions. In an honest execution of our protocol, all deposits will be made first before any of them is claimed. Also, the sequence deposits will be claimed in the reverse order in which they are made. Note that a malicious party may abort the protocol either (1) by not making a deposit it was supposed to make, or (2) by not claiming a deposit it could have claimed. The following two rules of thumb may be kept in mind to understand how honest parties behave in the event of such aborts.

1. When it's an honest party's turn to make a deposit, it makes the deposit if and only if all the deposits that were supposed to made before its deposit were made. That is, if a malicious party does not make a deposit during its turn, then no honest party makes any subsequent deposit in the protocol.
2. When it's an honest party's turn to make a claim, it makes the claim if it possesses all the witnesses necessary for making the claim. That is, an honest party may go ahead and claim a deposit even if (1) some deposits were not made, and (2) some claims were not made.

Two simplifying assumptions. The first is that our constructions will try to penalize deviations of party P_i in round r only when $(r,i) \neq (1,1)$. Later in this section, we show how to handle the "bootstrapping" step of forcing P_1 to start the protocol. The second is that we assume parties can use only unique witnesses to claim $\mathcal{F}^\star_{\mathrm{CR}}$ 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.

We construct our final protocol in a step-by-step manner. We start with $n = 2$ and $m = 1$.

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

$$P_1 \xrightarrow[q,\tau_2]{T_1 \wedge T_2} P_2 \qquad (\mathsf{Tx}_2)$$

$$P_2 \xrightarrow[q,\tau_1]{T_1} P_1 \qquad (\mathsf{Tx}_1)$$

Note that the verification circuits for these transactions are simply the corresponding transcript checking functions $\mathsf{tv}^{\pi_f}_{r,i}$, 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 Tx_1. Using T_1 revealed by P_1, party P_2 is able to claim Tx_2. We first consider aborts during the initial deposit phase. If P_1 aborts without making Tx_2, then clearly no money changes hands and we are good. Now if P_2 aborts without making Tx_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 Tx_2, and thus no money changes hands, and we are good. These attacks imply that we do not even get past the initial deposit phase (meaning that we are not required to penalize any party).

Next, we consider aborts during the reveal phase. Recall that once we enter the reveal phase, then we must penalize P_2 if P_1 revealed T_1 but P_2 did not reveal T_2. First suppose P_1 aborts, i.e., does not claim Tx_1. Then note that Tx_1 gets refunded back to P_2, and no party is penalized. Note that if P_1 does claim Tx_1, then P_2 is able to claim Tx_2, and the parties even out as well as obtain both T_1 and T_2. Next, we consider the case when P_2 aborts the protocol, i.e., does not claim Tx_2. In this case, Tx_2 gets refunded back to P_1. Also, P_1 would have already gained $\mathsf{coins}(q)$ after claiming Tx_1 and hence is compensated at the end of the protocol.

We use the following notation to simplify the presentation: for $r \in [m]$, let $\mathrm{TT}_r = \wedge_{s=1}^{r}(T_{s,1} \wedge \cdots \wedge T_{s,n})$, and for $i \in [n]$ and $r \in [m]$, let $\mathrm{TT}_{r,i} = \mathrm{TT}_{r-1} \wedge (\wedge_{j=1}^{i} T_{r,j})$. (Here TT stands for "transcript.") Also, let "$(r',i') > (r,i)$" if either (1) $r' > r$, or (2) $r' = r$ and $i' > i$.

ROOF DEPOSIT.

$$P_1 \xrightarrow[q,\tau_{m,2}]{\mathrm{TT}_{m,2}} P_2 \qquad (\mathsf{Tx}_{m,2})$$

SEE-SAW DEPOSITS. For $r = m-1$ to 1:

$$P_2 \xrightarrow[2q,\tau_{r+1,1}]{\mathrm{TT}_{r+1,1}} P_1 \qquad (\mathsf{Tx}_{r+1,1})$$

$$P_1 \xrightarrow[2q,\tau_{r,2}]{\mathrm{TT}_{r,2}} P_2 \qquad (\mathsf{Tx}_{r,2})$$

FLOOR DEPOSIT.

$$P_2 \xrightarrow[q,\tau_{1,1}]{\mathrm{TT}_{1,1}} P_1 \qquad (\mathsf{Tx}_{1,1})$$

Figure 3: Multi-round two party see-saw mechanism.

Multi-round two-party see-saw mechanism. The sequence of transactions is 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 $\mathsf{Tx}_{1,1}$. Using $\mathrm{TT}_{1,1} = T_{1,1}$ revealed by P_1, party P_2 is now able to claim $\mathsf{Tx}_{1,2}$ by revealing $\mathrm{TT}_{1,2} = T_{1,1} \wedge T_{1,2}$. Likewise parties P_1 and P_2 take turns claiming each others' $\mathcal{F}^{\star}_{\mathrm{CR}}$ transactions.

We first consider aborts during the initial deposit phase. Suppose P_i aborts without making $\mathsf{Tx}_{r,j}$ for $j \neq i$ and some r. First, this ensures that (1) P_j does not make $\mathsf{Tx}_{r',i}$ for $(r',i) < (r,j)$, and (2) P_j will never reveal $T_{r,j}$ (since $T_{r,j}$ needs to be revealed only to claim $\mathsf{Tx}_{r',i'}$ for $(r',i') \geq (r,j)$), and (3) no party can claim $\mathsf{Tx}_{r',i'}$ for $(r',i') \geq (r,j)$ (since $T_{r,j}$ is necessary to claim $\mathsf{Tx}_{r',i'}$), and (4) all the deposits $\mathsf{Tx}_{r',i'}$ for $(r',i') > (r,i)$ (i.e., those that were made so far) will get 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 $\mathsf{Tx}_{1,1}$. 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 cannot claim $\mathsf{Tx}_{1,2}$. Now note that all deposits $\mathsf{Tx}_{r,i}$ 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 $\mathsf{Tx}_{1,1}$ (i.e., the reveal phase has started). Then in this case, P_2 is down $\mathsf{coins}(q)$ while P_1 is up $\mathsf{coins}(q)$. If P_2 aborts at this stage, then essentially P_2 has compensated P_1 with $\mathsf{coins}(q)$. On the other hand if P_2 claims $\mathsf{Tx}_{1,2}$, then note that it gets $\mathsf{coins}(2q)$ from that claim. Thus, it is now $\mathsf{coins}(q)$ up while P_1 is down $\mathsf{coins}(q)$. It is easy to see that as the remaining claims are made, parties take turns going up and down $\mathsf{coins}(q)$ (hence the name "see-saw"). Thus we have the property that whenever a party P_i claims $\mathsf{Tx}_{r,i}$ (except for $(r,i) = (m,2)$), it gains $\mathsf{coins}(q)$ while the other party loses $\mathsf{coins}(q)$. This incentivizes the other party to go ahead and claim $\mathcal{F}^{\star}_{\mathrm{CR}}$ transaction immediately above $\mathsf{Tx}_{r,i}$, say $\mathsf{Tx}_{r',i'}$. Indeed if the other party does not make the claim, then we have that the honest party (i.e., P_i) is compensated with $\mathsf{coins}(q)$ at the end of the protocol. This is because if $\mathsf{Tx}_{r',i'}$ is not claimed, then either (1) $(r',i') = (m,2)$, and this case P_i does not lose coins from this transaction, and simply ends the protocol with $\mathsf{coins}(q)$ as compensation, or (2) $(r',i') \neq (m,2)$, in which case P_i will never reveal $T_{r+1,i}$ thus making it impossible for any $\mathsf{Tx}_{r'',i''}$ to be claimed for any $(r'',i'') \geq (r+1,i)$, essentially ensuring that no further money transfers happen, and that P_i can end the protocol with $\mathsf{coins}(q)$ as compensation. Finally, in an honest execution, when P_2 claims the last transaction $\mathsf{Tx}_{m,2}$ it gets only $\mathsf{coins}(q)$ from that claim, and thus in this case both parties even out.

Multiparty locked ladder mechanism. Generalizing the two party solution is 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:

ROOF DEPOSITS. For $j \in \{1,2\}$:

$$P_j \xrightarrow[q,\tau_{3,j}]{\mathrm{TT}_3} P_3$$

THIRD STAGE DEPOSITS.

$$P_3 \xrightarrow[3q,\tau_{2,3}]{\mathrm{TT}_2} P_2$$

SECOND STAGE DEPOSITS.

$$P_2 \xrightarrow[q,\tau_{3,2}]{\mathrm{TT}_1} P_3$$

FIRST STAGE DEPOSITS.

$$P_2 \xrightarrow[q,\tau_{1,2}]{\mathrm{TT}_1} P_1$$

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 $\mathsf{coins}(q)$, i.e., P_2 has lost $\mathsf{coins}(2q)$. Then, in the third stage, it claims $\mathsf{coins}(3q)$ from P_3 by revealing T_2. This is effectively equivalent to P_3 compensating P_2 with $\mathsf{coins}(q)$, 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 $\mathsf{coins}(q)$ by P_3. Then, in the roof stage, P_3 claims back $\mathsf{coins}(q)$ 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. 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 $\mathsf{coins}(3q)$. While P_3 can use TT_2 to claim the roof deposits, and learn all the tokens, it does so at an expense of $\mathsf{coins}(q)$ (i.e., it claims $\mathsf{coins}(2q)$ from the roof deposits but has lost $\mathsf{coins}(3q)$ in the third stage deposits). This is clearly an undesirable situation as the honest party has lost $\mathsf{coins}(q)$.

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 [10] which allows transitions between different stages of the protocol. We explain these two ideas below.

Locking mechanism. Recall that the high level idea in our naïve 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 $\mathsf{coins}(q)$ each from a set of parties, followed by a set of transactions S_{-i} where the same set of parties each claim $\mathsf{coins}(q)$ 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 naïve 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 token via an initial MPC 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}, party P_i needs 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 naïve 3-party protocol enhanced with the locking mechanism (i.e., both second stage as well as the third stage deposits are locked). Now P_1 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 third stage deposit as it remains locked. Thus, essentially P_3 aborted the protocol, and yet P_2 does not gain $\mathsf{coins}(q)$ (in fact, it loses $\mathsf{coins}(q)$ here).

The above attack naturally leads us to include a $\mathcal{F}_{\mathrm{CR}}^{\star}$ transaction to P_i that can be claimed just by revealing TT_i, i.e., it is essentially an unlocked transaction. What the above would ensure is that P_i will never be stranded in a situation where it wishes to reveal its token, and yet is unable to claim any transactions. While the above is true, unfortunately if we include unlocked $\mathcal{F}_{\mathrm{CR}}^{\star}$ transactions from each P_j to P_i (i.e., those that can be claimed just using TT_i), then we have negated the locking mechanism, and are back to square one. Thus, what we want to do is to give a chance to P_i to avoid the "locked-out attack" while at the same time preventing the "coalition attack." To do this, we let only P_{i+1} make an unlocked $\mathcal{F}_{\mathrm{CR}}^{\star}$ transaction to P_i that can be claimed by revealing just TT_i. In some sense, this breaks the symmetry of the protocol, but it also gives us a chance to make use of the ladder mechanism of [10]. That is, following [10], we let P_{i+1} make an unlocked $\mathcal{F}_{\mathrm{CR}}^{\star}$ transaction to P_i for $\mathsf{coins}(i \cdot q)$ that can be claimed by revealing TT_i. We present our protocol in Figure 4.

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 $j > i$

ROOF DEPOSITS. For each $j \in [n-1]$:

$$P_j \xrightarrow[q,\tau_{2n-2}]{\mathrm{TT}_n} P_n$$

LADDER DEPOSITS. For $i = n-1$ down to 2:

- Rung unlock: For $j = n$ down to $i+1$:

$$P_j \xrightarrow[q,\tau_{2i-1}]{\mathrm{TT}_i \wedge U_{i,j}} P_i$$

- Rung climb:

$$P_{i+1} \xrightarrow[i \cdot q,\tau_{2i-2}]{\mathrm{TT}_i} P_i$$

- Rung lock: For each $j = n$ down to $i+1$:

$$P_i \xrightarrow[q,\tau_{2i-2}]{\mathrm{TT}_{i-1} \wedge U_{i,j}} P_j$$

FOOT DEPOSIT.

$$P_2 \xrightarrow[q,\tau_1]{\mathrm{TT}_1} P_1$$

Figure 4: Locked ladder mechanism.

(these correspond to S_{+i}), an unlocked deposit from P_{i+1} that can be claimed if P_i reveals TT_i, and makes deposits to P_j for $j > i$ that are locked with $U_{i,j}$ (these correspond to S_{-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 [10], the parties metaphorically climb the ladder as they take turns claiming the ladder deposits. The difference from [10] 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. Hence, while the protocol is being executed, P_i first pays the parties above it (who haven't "played" yet), but P_i will then immediately be able to "play" by extending TT_{i-1} and thereby reclaim these coins that it paid, thus avoiding the locked-out and coalition attacks.

As in the ladder mechanism of [10], once the i-th ladder deposit is claimed, parties P_1 through P_i become "inactive" in the sense that they no longer claim any deposits and nor are any of their ladder deposits remain unclaimed. (In fact their only unclaimed deposits are those that are part of the roof deposits.) It is easy to see that the "inactive" parties are always $\mathsf{coins}(q)$ up after the i-th ladder rung unlock deposits are claimed, and that they remain $\mathsf{coins}(q)$ up until the beginning of roof claims. As it turns out, the lock and ladder mechanisms are sufficient to deal with aborts in the deposit and reveal phases, respectively.

Multiparty see-saw mechanism. Our idea is to mimic the two-party see-saw mechanism. That is, all we need to do is to ensure that the end of each of the m rounds, party P_1 has already com-

pensated $\mathsf{coins}(q)$ to every other party, and is thus incentivized to send the first token for the next round. This is quite straightforward to implement. For every round we invoke an instance of the single-round locked ladder mechanism (with the transcript verification circuits 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 w.r.t. 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:

$$P_j \xrightarrow[q,\tau_{r+1,1}]{\mathrm{TT}_{r+1,1}} P_1 \qquad (\mathsf{Tx}^{\mathsf{chain}}_{j,1})$$

- For $j = 2$ to n:

$$P_1 \xrightarrow[q,\tau_{r+1,1}]{\mathrm{TT}_{r,n}} P_j \qquad (\mathsf{Tx}^{\mathsf{chain}}_{1,j})$$

Remark. Note that solving the single-round multiparty case yields a solution to the multi-round multiparty case as well. To see why, let us denote the problem for the m-round n-party case by $\mathsf{LL}_{m,n}$. Then the problem $\mathsf{LL}_{m,n}$ is obtained by simply "folding" $\mathsf{LL}_{1,nm}$. That is, for $i \in [nm]$, interpret P_i's move in $\mathsf{LL}_{1,nm}$ as $P_{i \bmod n}$'s move in the $(\lfloor i/n \rfloor + 1)$-th round of $\mathsf{LL}_{m,n}$. The key observation behind why this transformation is secure is that any protocol π solving $\mathsf{LL}_{1,nm}$ is resistant to malicious coalitions of any subset of nm parties, and therefore, the "folded" m-round n-party protocol obtained from π is also resistant to any coalitions of subset of the n parties. Since the protocol in Figure 4 solves the single-round multiparty case, we trivially obtain the multi-round multiparty solution.

However note that the efficiency of such a solution obtained for the m-round two-party case i.e., for $\mathsf{LL}_{m,2}$ by using the $2m$-party locked ladder mechanism, has worse efficiency than the two-party see-saw protocol from Figure 3. While the see-saw protocol requires parties to each deposit $\mathsf{coins}(2mq)$, the amount deposited in the ladder grows as $O(m^2 q)$.

Bootstrapping. 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" $\{U_{1,j}\}_{j\in\{2,\ldots,n\}}$. These dummy tokens are obtained by the parties via an initial secure computation step. In more detail, for all $j \in [n] \setminus \{1\}$, the secure computation protocol:

- chooses $U_{1,j} \leftarrow \{0,1\}^\lambda$ and $\omega^{\mathsf{boot}}_{1,j} \leftarrow \{0,1\}^\lambda$ at random;
- computes $\mathsf{com}^{\mathsf{boot}}_{1,j} \leftarrow S(1^\lambda, U_{1,j}, \omega^{\mathsf{boot}}_{1,j})$;
- outputs $\mathsf{com}^{\mathsf{boot}}_{1,j}$ to all parties and $(U_{1,j}, \omega^{\mathsf{boot}}_{1,j})$ to P_j,

where S is the sender algorithm of a honest binding commitment scheme. Note that $\mathsf{com}^{\mathsf{boot}}_{1,j}$ is computed in order to allow parties to generate the verification circuit for transaction $\mathsf{Tx}^{\mathsf{boot}}_{1,j}$ and $\mathsf{Tx}^{\mathsf{boot}}_{j,1}$ described below. Also, we stress that for $j \neq 1$, the dummy token $U_{1,j}$ is unknown to P_1; it only knows the corresponding commitment $\mathsf{com}^{\mathsf{boot}}_{1,j}$. (We note that the above MPC step can be combined with the MPC step for handling the cash distribution step (cf. Section 4) 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:

$$P_j \xrightarrow[q,\tau_1]{T_1 \wedge U_{1,j}} P_1 \qquad (\mathsf{Tx}^{\mathsf{boot}}_{j,1})$$

- For $j = 2$ to n:

$$P_1 \xrightarrow[q,\tau_0]{U_{1,j}} P_j \qquad (\mathsf{Tx}^{\mathsf{boot}}_{1,j})$$

That is, first each P_j makes a deposit $\mathsf{Tx}^{\mathsf{boot}}_{j,1}$ to P_1, and then P_1 makes deposits $\mathsf{Tx}^{\mathsf{boot}}_{1,j}$ to each P_j. Then in the reveal phase, the claims are made in reverse: each P_j first claims $\mathsf{Tx}^{\mathsf{boot}}_{1,j}$ 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 $\mathsf{Tx}^{\mathsf{boot}}_{j,1}$. More importantly, note that once the bootstrap deposits are made, an honest P_j will always claim $\mathsf{Tx}^{\mathsf{boot}}_{1,j}$, and thus will be $\mathsf{coins}(q)$ up. Thus the onus is on P_1 to deliver the first token (and to reclaim its $\mathsf{coins}(q)$), failing which it effectively pays a penalty $\mathsf{coins}(q)$ to each honest party. 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.

We are now ready to state our main theorem. Since the ideal oblivious transfer primitive $\mathcal{F}_{\mathrm{OT}}$ is suffficient to obtain a common random string, we can then apply e.g., [21] to obtain:

Theorem 2. *Let $(F, d^\star)$ 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) $\mathcal{F}^\star_{F,d^\star}$ in the $(\mathcal{F}_{\mathrm{OT}}, \mathcal{F}^\star_{\mathrm{CR}})$-hybrid model.*

Proof sketch. The main idea behind the proof is that the witnesses used to claim $\mathcal{F}^\star_{\mathrm{CR}}$ transactions are simply successive messages of a secure computation protocol π_F that realizes that standard reactive functionality $\mathcal{F}_F$. Since π_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 π_F, (2) the simulator for initial MPC step (alternatively access to ideal unfair functionality realized in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model) 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 $\{T^{\mathsf{fin}}_{i,j,k}\}$), 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 [10]. We defer further details to the full version. □

6. EFFICIENT POKER PROTOCOL

In this section we describe an optimized protocol for Texas hold 'em poker that avoids non-black-box use of a secure computation protocol. Our key observation is that in each stage, only player P_i has an input in a stage of the computation that corresponds to player P_i's r-th round move. Let (S, R) be a non-interactive honest binding commitment (cf. Definition 3, Appendix A). Parties run a secure computation protocol that does the following:

- selects hands h_i 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 $\omega^h_{i,j}$ to secret share $h_{i,j}$ to obtain $\mathsf{com}^h_{i,j}$ and set $\mathsf{Tag}^h_{i,j} = \mathsf{com}^h_{i,j}$ and $\mathsf{Token}^h_{i,j} = (h_{i,j}, \omega^h_{i,j})$;
- applies the sender algorithm of an honest-binding commitment using random $\omega^y_{i,j}$ to secret share $y_{i,j}$ to obtain $\mathsf{com}^y_{i,j}$ and set $\mathsf{Tag}^y_{i,j} = \mathsf{com}^y_{i,j}$ and $\mathsf{Token}^y_{i,j} = (y_{i,j}, \omega^y_{i,j})$;
- sets $\mathsf{AllTags} = \{\mathsf{Tag}^h_{i,j}, \mathsf{Tag}^y_{i,j}\}_{i,j\in[n]}$; and
- delivers $\mathsf{AllTags}, \{\mathsf{Token}^h_{i,j}, \mathsf{Token}^y_{i,j}\}_{j\in[n]}$ to each P_i.

Note that at the end of this step, 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 commimtments.

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 $\phi_{i,r}$ 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 $\mathcal{F}^\star_{\mathrm{CR}}$ transactions will depend on $\mathsf{com}^h_{i,j}$ 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 $\mathcal{F}^\star_{\mathrm{CR}}$ 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 hand 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 $\mathsf{tv}_{r,i}$ (tv stands for "transcript validity") that takes the transcript of the poker game 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 $Y^1_j = \{y_{1,j}, y_{2,j}, y_{3,j}\}$, 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^1_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 $h_{i,i}$ (which reveals their hand). Once all parties complete the showdown round, and the entire transcript $\mathrm{TT}_{m,n}$ is available, then the pot winner can be determined. Note that we run only one MPC at the very beginning, and $\mathsf{AllTags}$ generated in this step is sufficient to design the verification circuits for all $\mathcal{F}^\star_{\mathrm{CR}}$ 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" [1], 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 $d^\star = (d^\star_1, \ldots, d^\star_n)$, and for each $i \in [n]$, let $m_i = \lceil \log(d^\star_i) \rceil$. As in Section 4, 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 $\mathcal{F}^\star_{\mathrm{CR}}$ transaction as follows (we slightly abuse the $\mathcal{F}^\star_{\mathrm{CR}}$ notation and use the verification circuit instead of the verifying witness):

$$P_i \xrightarrow[2^k, \tau_0]{\phi^{\mathsf{fin}}_{i,j,k}} P_j \qquad (\mathsf{Tx}^{\mathsf{fin}}_{i,j,k})$$

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

- outputs 0 and terminates if $\mathsf{tv}_{m,n}(\mathrm{TT}_{m,n}) = 0$;
- computes $z^\star = (z^\star_1, \ldots, z^\star_n)$ using $\mathrm{TT}_{m,n}$, where $z^\star_i$ represents the amount which party P_i is supposed to get at the end of the protocol;
- computes $A = g(d^\star, z^\star)$ (cf. Observation 1), lets $a_{i,j}$ denote the (i, j)-th entry of matrix A, and lets $b^\star_{i,j,1}, ..., b^\star_{i,j,m_i}$ be the binary representation of $a_{i,j}$;
- outputs 1 if $b^\star_{i,j,k} = 1$, else outputs 0.

Efficiency. Note that each party P_i makes $(n-1) \cdot m_i$ calls to $\mathcal{F}^\star_{\mathrm{CR}}$ and deposits a total of $(n-1) \cdot d^\star_i$ coins. For implementing the see-saw mechanism we require $O(n^2 m)$ calls to $\mathcal{F}^\star_{\mathrm{CR}}$ and each party to make a maximum deposit of $O(nm)$ 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.

7. CONCLUSIONS

In this paper, we presented formal definitions for *secure cash distribution with penalties* (SCD), a primitive that allows stateful computations involving data and/or money, and guarantees a strong notion of dropout tolerance. We then constructed a protocol for SCD that only makes use of a claim-or-refund transaction functionality $\mathcal{F}^\star_{\mathrm{CR}}$ (which can be implemented in a variant of Bitcoin) and is otherwise independent of the Bitcoin ecosystem. Our SCD protocol may be improved in a number of ways, including improvements to round complexity, validation complexity of $\mathcal{F}^\star_{\mathrm{CR}}$ transactions, as well as alternate constructions that make only black-box use of MPC.

Acknowledgments

We would like to thank Ananth Raghunathan, Yuval Ishai, and Stefan Dziembowski for useful discussions. The work of the first author was supported by funding from NSF grants CNS-1350619, CNS-1414119, and CNS-1413920, and funding from Qatar Computing Research Institute, and funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 259426. The work of the second author was supported by funding from ISF grant no. 1790/13 and by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 293843. The work of the third author was supported by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258.

8. REFERENCES

[1] A. Shamir, R. Rivest, and L. Adleman, "Mental poker." *The Mathematical Gardener.*, pp. 37–43, 1981.

[2] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, Nov. 1982, pp. 160–164.

[3] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," in *19th Annual ACM Symposium on Theory of Computing (STOC)*, A. Aho, Ed. ACM Press, 1987.

[4] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin." in *IEEE Security and Privacy*, 2014.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, http://bitcoin.org/bitcoin.pdf.

[6] M. Green, "Poker is hard, especially for cryptographers," http://blog.cryptographyengineering.com/2012/04/poker-is-hard-especially-for.html, 2013.

[7] M. Jakobsson, D. Pointcheval, and A. Young, "Secure mobile gambling," in *Cryptographers' Track — RSA 2001*, ser. LNCS, D. Naccache, Ed., vol. 2020. Springer, Apr. 2001, pp. 110–125.

[8] A. Back and I. Bentov, "Note on fair coin toss via bitcoin," http://arxiv.org/abs/1402.3698, 2013.

[9] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Fair two-party computations via the bitcoin deposits." in *First Workshop on Bitcoin Research, FC*, 2014.

[10] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols." in *Crypto (2)*, 2014, pp. 421–439.

[11] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, Oct. 2001.

[12] "Bitcoin CVEs," https://en.bitcoin.it/wiki/CVEs#CVE-2010-5141.

[13] R. Kumaresan and I. Bentov, "How to use bitcoin to incentivize correct computations." in *CCS*, 2014.

[14] G. Andresen, "Turing complete language vs non-turing complete." https://bitcointalk.org/index.php?topic=431513.20#msg4882293.

[15] A. Yao, "How to generate and exchange secrets (extended abstract)," in *FOCS*, 1986, pp. 162–167.

[16] R. Cleve, "Limits on the security of coin flips when half the processors are faulty (extended abstract)." in *STOC*, 1986, pp. 364–369.

[17] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," Cryptology ePrint Archive, Report 2015/675, 2015, http://eprint.iacr.org/2015/675.

[18] O. Goldreich, "Foundations of cryptography - vol. 2," 2004.

[19] S. Barber, X. Boyen, E. Shi, and E. Uzun, "Bitter to better - how to make bitcoin a better currency." in *FC*, 2012.

[20] G. Maxwell, "Zero knowledge contingent payment. 2011," https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.

[21] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *34th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, May 2002, pp. 494–503.

[22] J. A. Garay, J. Katz, R. Kumaresan, and H.-S. Zhou, "Adaptively secure broadcast, revisited." ACM Press, 2011, pp. 179–186.

APPENDIX

A. FORMAL DEFINITIONS

Definition 3 (Honest binding commitments [22])**.** A (non-interactive) *commitment scheme* for message space $\{\mathcal{M}_\lambda\}$ is a pair of PPT algorithms S, R such that for all $\lambda \in \mathbb{N}$, all messages $m \in \mathcal{M}_\lambda$, and all random coins ω it holds that $R(m, S(1^\lambda, m; \omega), \omega) = 1$. A commitment scheme for message space $\{\mathcal{M}_\lambda\}$ is *honest-binding* if:

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

$$\Pr\left[\begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); \\ \omega \leftarrow \{0,1\}^*;\ \mathsf{com} \leftarrow S(1^\lambda, m; \omega); \\ (m', \omega') \leftarrow \mathcal{A}(\mathsf{com}, \omega) : \\ \qquad R(m', \mathsf{com}, \omega') = 1 \bigwedge m' \neq m \end{array}\right]$$

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

$$\left|\begin{array}{l} \Pr\left[\begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); \\ \omega \leftarrow \{0,1\}^*;\ \mathsf{com} \leftarrow S(1^\lambda, m; \omega) : \\ \qquad \mathcal{A}(1^\lambda, \mathsf{com}, \omega) = 1 \end{array}\right] \\ \qquad - \Pr\left[\begin{array}{l} (\mathsf{com}, \mathsf{st}) \leftarrow \tilde{S}_1(1^\lambda); \\ m \leftarrow \mathcal{A}(1^\lambda);\ \omega \leftarrow \tilde{S}_2(\mathsf{st}, m) : \\ \qquad \mathcal{A}(1^\lambda, \mathsf{com}, \omega) = 1 \end{array}\right] \end{array}\right|$$

Equivocation implies the standard hiding property. Also, observe that binding holds for commitments generated by $(\tilde{S}_1, \tilde{S}_2)$. As observed in [10], we can construct highly efficient heuristically secure honest binding commitment schemes in the *programmable random oracle* model. In the following let Hash be a programmable hash function, and let $\omega \in \{0,1\}^\lambda$. We describe the algorithms S, R (algorithms $\tilde{S}_1, \tilde{S}_2$ are obtained by standard oracle programming techniques).

$\underline{S(1^k, m; \omega)}$
  return $\mathsf{com} := \mathsf{Hash}(m \| \omega)$;

$\underline{R(m, \mathsf{com}, \omega)}$
  If $\mathsf{com} \stackrel{?}{=} \mathsf{Hash}(m \| \omega)$
    return 1;
  else return 0;