Instantaneous Decentralized Poker

Iddo Bentov[1(✉)], Ranjit Kumaresan[2], and Andrew Miller[3]

[1] Cornell University, Ithaca, USA
iddobentov@cornell.edu
[2] Microsoft Research, Redmond, USA
vranjit@gmail.com
[3] University of Illinois at Urbana-Champaign, Champaign, USA
soc1024@illinois.edu

Abstract. We present efficient protocols for *amortized* secure multiparty computation with penalties and secure cash distribution, of which poker is a prime example. Our protocols have an initial phase where the parties interact with a cryptocurrency network, that then enables them to interact only among themselves over the course of playing many poker games in which money changes hands.

The high efficiency of our protocols is achieved by harnessing the power of stateful contracts. Compared to the limited expressive power of Bitcoin scripts, stateful contracts enable richer forms of interaction between standard secure computation and a cryptocurrency.

We formalize the stateful contract model and the security notions that our protocols accomplish, and provide proofs in the simulation paradigm. Moreover, we provide a reference implementation in Ethereum/Solidity for the stateful contracts that our protocols are based on.

We also adapt our off-chain cash distribution protocols to the special case of stateful duplex micropayment channels, which are of independent interest. In comparison to Bitcoin based payment channels, our duplex channel implementation is more efficient and has additional features.

1 Introduction

As demonstrated by Cleve [13], fair multiparty computation without an honest majority is impossible in the standard model of communication. Hence, there have been numerous attempts to circumvent this theoretical impossibility result, in particular by relying on techniques such as gradual release (cf. [35] for a survey) and optimistic fair exchange [4]. With the introduction of Bitcoin [33], the academic study of decentralized cryptocurrencies gave rise to a line of research that seeks to impose fairness in secure multiparty computation (MPC) by means of monetary penalties [8]. In this model, the participating parties make security deposits, and the deposits of parties who deviate from the protocol are used to compensate the honest parties.

Still, interacting with a Proof-of-Work based decentralized network entails long waiting times due to the need to be secure against reversal of the ledger history. A recent work by Kumaresan and Bentov [27] showed a Bitcoin based

T. Takagi and T. Peyrin (Eds.): ASIACRYPT 2017, Part II, LNCS 10625, pp. 410–440, 2017.
https://doi.org/10.1007/978-3-319-70697-9_15

amortization scheme in which the parties run an initial setup phase requiring interaction with the cryptocurrency network, but thereafter they engage in many fair secure computation executions, communicating only among themselves for as long as all parties are honest.

1.1 Our Contributions

Asymptotic gains in amortized protocols. We present new protocols that rely on stateful contracts instead of Bitcoin transactions, and thereby improve upon the previous results in several ways. First, the setup phase of [27] requires the n parties to execute $O(n^2)$ PoW-based rounds of interaction with the cryptocurrency network, while our stateful protocols require $O(1)$ rounds. The protocols of [27] for secure MPC with penalties also require a security deposit of $O(n^2)$ coins per party, while our protocols require $O(n)$ coins per party. We use UC-style definitions [11] to formalize the security notions that are achieved by our amortized protocols, and provide proofs using the simulation paradigm.

Amortized SCD. Unlike the protocols in [27], our protocols support *secure cash distribution with penalties* (SCD), rather than only fair secure MPC with penalties. The distinction between SCD and fair MPC with penalties is that in SCD the inputs and outputs of the parties are comprised of both money and data, while fair MPC with penalties has only data for inputs and outputs (but uses money to compensate honest parties who did not learn the output).

Real poker. A canonical example of SCD is a mental poker game, where the outcome of the computation is not intrinsically useful, but rather determines how money should change hands. This means that following an on-chain setup phase, the parties can play any number of *instantaneous* poker games, for as long as no party has run out of money. Hence, while there is a large body of work on efficient mental poker schemes, to the best of our knowledge we are the first to provide a practical poker protocol with actual money transfers from the losers to the winners. Moreover, we accompany our poker protocol with an implementation for the Ethereum cryptocurrency.

Highly efficient payment channels. As a special case, our off-chain cash distribution protocols can also be used for stateful bi-directional payment channels. This use case does not require secure computation and yet it is particularly important. The reason for this is that micropayment channels can reduce the amount of transaction data that the decentralized cryptocurrency network maintains, and thus the long-term scalability pressures that a cryptocurrency faces can be relieved by well-functioning off-chain payment channels (see, e.g., [18] for further discussion). In comparison to Bitcoin based off-chain payment channels, our stateful approach yields better efficiency and extra features. Since micropayment channels are of independent interest, we provide a self-contained protocol and implementation of our stateful duplex off-chain channel.

1.2 Related Works

The first secure computation protocols that utilize Bitcoin to guarantee fairness are by Maxwell [30], Barber et al. [5], Andrychowicz et al. [2,3] and Bentov and Kumaresan [8]. Bitcoin based protocols for reactive cash distribution and poker were given by Kumaresan et al. [26]. The technique for amortized secure computation with penalties in the Bitcoin model was introduced by Kumaresan and Bentov [27]. Our protocols subsume and improve on these, providing both the amortization benefit of [27] with the cash distribution functionality of [26], and furthermore reduce the on-chain costs and the necessary amount of collateral. Several other works analyze fair protocols in rigorous models, in particular Kiayias et al. [24] and Kosba et al. [25] introduced formal cryptocurrency modeling and presented fair (non-amortized) protocols that improve upon the PoW-based round complexity and collateral requirements of the Bitcoin-based protocols in the prior works.

The cash distribution contract we present (see Sect. 2.3) is closely related to an ongoing proposal in the cryptocurrency community for "state channels" (cf. Coleman [14]), wherein a group of parties agrees on a sequence of "off-chain" state transitions, and resort to an on-chain reconciliation process only in the case that the off-chain communications break down. To our knowledge, no security definition has yet been provided for such applications. Furthermore, our application is much more expressive, since we can implement state transitions that depend on parties' private information, while still guaranteeing fairness.

The original mental poker protocol by Shamir et al. [36] relies on commutative encryption. However, their protocol was only for two parties and was found to have security vulnerabilities [15,29]. Following that, many different protocols for mental poker were proposed. For example, Crépeau presented secure poker protocols that are based on probabilistic encryption [16] and zero-knowledge proofs [17], but his constructions are rather inefficient. In 2003, a breakthrough by Barnett and Smart [6] gave a far more efficient poker protocol. Castellà-Roca et al. [12] utilized homomorphic encryption to construct a poker protocol that is similar to [6]. Bayer and Groth [7] later gave a secure and efficient shuffle procedure that can be integrated with [6].

The poker protocol that we integrate into our SCD implementation is by Wei and Wang [23], with a full version by Wei [22]. This protocol uses a proof of knowledge scheme that is slightly faster than [6,7,12], and provides a security proof using the simulation paradigm.

2 Overview

In Bitcoin, the full nodes maintain a data structure that is known as the "unspent transaction outputs set" (UTXO set), which represents the current state of the ledger. Each unspent output in the UTXO set incorporates a circuit (a.k.a. script or predicate), such that any party who can provide an input (a.k.a. witness) that satisfies the circuit can spend the coins amount of this output into a new

unspent output. Therefore, if there is only one party who knows the witness for the circuit, then this party is in effect the holder of the coins.

Standard Bitcoin transactions use a signature as the witness. The signature is applied on data that also references the new unspent output, thereby binding the transaction to the specific receiver of the coins and thus prevents a man-in-the-middle attack by the nodes in the decentralized Bitcoin network.

However, Bitcoin allows the use of more complex circuits as well. Such circuits allow us to support quite elaborate protocols in which money changes hands, as opposed to using Bitcoin only for simple money transfers between parties.

Specifically, protocols for fair secure computation and fair lottery can be implemented with a blackbox use of an $\mathcal{F}_{\text{CR}}^{\star}$ functionality [8,26,28]. Essentially, $\mathcal{F}_{\text{CR}}^{\star}$ specifies that a "sender" P_1 locks her coins in accordance with some circuit ϕ, such that a "receiver" P_2 can gain possession of these coins if she supplies a witness w that satisfies $\phi(w) = 1$ before some predefined timeout, otherwise P_1 can reclaim her coins. As shown in [8,27], the $\mathcal{F}_{\text{CR}}^{\star}$ functionality can be realized in Bitcoin, as long as the circuit ϕ can be expressed in the Bitcoin scripting language. In the aforementioned secure computation and lottery protocols [8,26,28], the particular circuit that is needed verifies a signature (just as in standard transactions) and a decommitment according to some arbitrary hardcoded value. Such a circuit can be realized by using a hash function for the commitment scheme (Bitcoin supports `SHA1`,`SHA256`,`RIPEMD160`). Since signature verification is an order of magnitude more complex than hash invocation, the complexity of an $\mathcal{F}_{\text{CR}}^{\star}$ transaction is only marginally higher than that of standard Bitcoin transactions.

Note that our underlying assumption is that an honest party can interact with the cryptocurrency network (within a bounded time limit) to ensure her monetary compensation. We also assume that the off-chain communication among the parties takes place in a separate point-to-point synchronous network. Given that the network is synchronous, our MPC protocols will be secure even if only one party is honest.

The $\mathcal{F}_{\text{CR}}^{\star}$ model can be regarded as a restricted version of the Bitcoin model, which is expressive enough for realizing multiparty functionalities that are impossible in the standard model. One may ask whether it is possible to design better protocols in a model that is more expressive than the Bitcoin model. In this work we will answer the question in the affirmative.

A possible extension to the Bitcoin transaction structure is covenants [32,34], where each unspent output specifies not only the conditions on who can spend the coins (i.e., the circuit ϕ), but also conditions on who is allowed to receive the coins. Indeed, as shown in [32], covenants can be used to implement certain tasks that the current Bitcoin specifications do not support (e.g., vaults that protect against coin theft).

Generalizing further, each unspent output can maintain a *state*. That is, an unspent output will be comprised of a circuit ϕ and state variables, and parties can update the state variables by carrying out transactions that satisfy ϕ in accord with the current values of the state variables. Additionally, parties can

deposit coins into the unspent output, and a party can withdraw some partial amount of the held coins by satisfying ϕ with respect to the state variables. This approach is used in Ethereum [10,38], where the notion of "outputs" is replaced with "user accounts" and automated "contract accounts".

With a slight abuse of terminology, the transaction format of the Bitcoin model can thus be described as "stateless". By this we mean that the coins of an unspent Bitcoin output are controlled by a hardcoded predicate that represents their current state, and anyone who can supply a witness that satisfies this predicate is able to spend these coins into an arbitrary new state.

Let us mention that the Bitcoin transaction format can still enable "smart contracts", in the sense of having coins that can be spent only if some other transaction took place (i.e., without relying on a third party). The technique for achieving this would generally involve multiple signed transactions that are prepared in advance and kept offline. Then, depending on the activity that occurs on the blockchain, some of the prepared transaction will become usable. However, in certain instances the amount of offline transactions may grow exponentially, as in the case of zero-collateral lotteries [31].

The protocols that we present in this work will be in a model that has stateful contracts. As described, this refers to unspent outputs that are controlled according to state variables. It should be emphasized that our protocols do not rely on a Turing-complete scripting language, as all the loops in the contracts that we design (in particular our poker contract) have a fixed number of iterations.

To justify our modeling choice, let us review the advantages of stateful contracts over stateless transactions. As a warmup, we begin by examining simple protocols for 2-party fair exchange.

2.1 Fair Exchange with Penalties Between Two Parties

Suppose that P_1, P_2 execute an unfair secure computation that generates secret shares of the output $x = x_1 \oplus x_2$ and commitments $T_1 = h(x_1), T_2 = h(x_2)$, and delivers (x_i, T_1, T_2) to party P_i. Consider the naive protocol for fair exchange of the shares x_1, x_2 via $\mathcal{F}_{\mathrm{CR}}^{\star}$ transactions:

$$P_1 \xrightarrow[q,\tau]{T_2} P_2 \tag{1}$$

$$P_2 \xrightarrow[q,\tau]{T_1} P_1 \tag{2}$$

An arrow denotes an $\mathcal{F}_{\mathrm{CR}}^{\star}$ transaction that lets P_i collect q coins from P_{3-i} before time τ, by revealing a decommitment y such that $h(y) = T_i$.

The above protocol is susceptible to an "abort" attack by a malicious P_2 that waits for P_1 to make the deposit transaction (i.e., Step 1), after which P_2 simply does not execute Step 2 to make a deposit transaction. Instead, P_2 claims the first transaction, and obtains q coins while P_1 obtains x_2. Now, P_2 simply aborts the protocol. In effect, this means that an honest P_1 paid q coins to learn P_2's share. Fairness with penalties requires that an honest party never loses any money, hence this naive approach does not work (cf. [8] for precise details).

The above vulnerability can be remedied via the following protocol:

$$P_1 \xrightarrow[q,\tau_2]{T_1 \wedge T_2} P_2 \tag{3}$$

$$P_2 \xrightarrow[q,\tau_1]{T_1} P_1 \tag{4}$$

For this improved protocol to be secure, two *sequential* PoW-based waiting periods are necessary. Otherwise, a corrupt P_2 may be able to reverse transaction (4) after P_1 claims it, so that P_1 would reveal her share and not be compensated.

By contrast, consider the 2-party fair exchange protocol that is based on a stateful contract, as illustrated in Fig. 1. Here, both parties should deposit q coins each, concurrently. If the $2q$ coins were not deposited into the contract before the timeout is reached, then an honest party who deposited into the contract can claim her coins back. In the case that the $2q$ coins were deposited, it triggers the contract to switch to a new state, where each party P_i can claim her deposit by revealing the hardcoded decommitment T_i. In contrast to the $\mathcal{F}^{\star}_{\mathrm{CR}}$ protocol, the stateful contract requires only one PoW-based waiting period before the honest parties may reveal their shares.

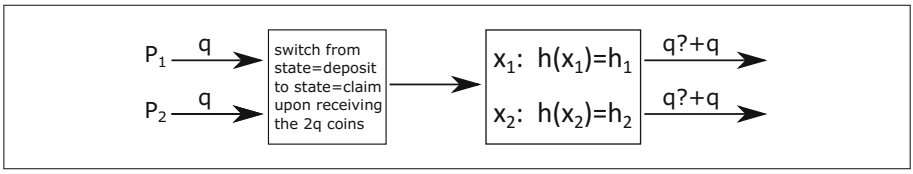

Fig. 1. Stateful contract for fair secure 2-party computation with penalties.

While the quantitative difference between stateful and stateless contracts in the above discussion may appear to be unimpressive, the distinction becomes more pronounced in the case of multiparty fair exchange (a.k.a. fair reconstruction [8]), and even more so in amortized protocols. Let us demonstrate the amortized multiparty case in the next section.

2.2 Amortized Multiparty Fair SFE with Penalties

We illustrate in Fig. 2 the stateful contract for n parties who wish to engage in amortized fair secure function evaluation (SFE) with penalties. The lifespan of this contract can be thought of as having three phases:

– *Deposit Phase:* All parties should deposit $q(n-1)$ coins each. If $nq(n-1)$ coins were deposited before the initial timeout is reached, then the contract switches into an "active" state. Otherwise, each honest party who deposited will claim her $q(n-1)$ coins back.

– *Execution Phase:* While the state is "active", the n parties will not interact with the contract at all. Instead, they will engage in multiple executions of SFE. In the i^{th} execution, the secure computation prepares secret shares $\{x_{i,j}\}_{j=1}^{n}$ of the output, as well as commitments $\{h_{i,j} = h(x_{i,j})\}_{j=1}^{n}$, and delivers $(x_{i,j}; h_{i,1}, h_{i,2}, \ldots, h_{i,n})$ to party P_j. Each party will then use her secret key (for which the corresponding public key is hardcoded in the contract) to create a signature $s_{i,j}$ for the tuple $(h_{i,1}, h_{i,2}, \ldots, h_{i,n})$, and send the signature $s_{i,j}$ to the other parties. Upon receiving all the signatures $\{s_{i,j}\}_{j=1}^{n}$, each honest P_j will send her secret share $x_{i,j}$ in the clear to the other parties.
– *Claim Phase:* In the case that a corrupt party P_c did not reveal her share $x_{i,c}$ during the execution phase, each honest party P_j will send $m_{i,j} = (x_{i,j}; s_{i,1}, s_{i,2}, \ldots, s_{i,n})$ to the contract, and thereby transition the contract into a "payout" state. The message $m_{i,j}$ also registers that P_j deserves to receive a compensation of q coins, in addition to her initial $q(n-1)$ coins deposit. Until a timeout, any party P_ℓ can avoid being penalized by sending $m_{i',\ell} = (x_{i',\ell}; s_{i',1}, s_{i',2}, \ldots, s_{i',n})$ with $i' \geq i$ to the contract. In case $i' > i$, this would invalidate the q coins compensation that was requested via $m_{i,j}$, and instead register that P_ℓ is owed q coins in compensation.

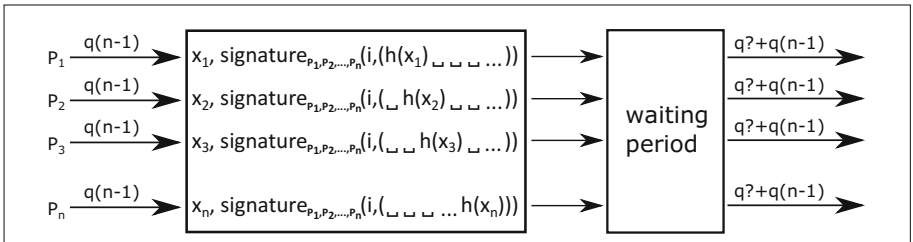

Fig. 2. Stateful contract for amortized multiparty fair SFE with penalties.

As can be observed, the n parties can engage in an unlimited amount of off-chain SFE executions (where the executions can compute different functions), and no interaction with the blockchain will take place as long as all parties are honest. When a corrupt party P_c deviates from this protocol, each honest party will receive q coins compensation, that is taken from P_c's initial security deposits of $q(n-1)$ coins. The actual protocol handles more technical issues, cf. Sect. 4.

By contrast, achieving the same guarantees in the $\mathcal{F}_{\text{CR}}^{\star}$ model is known to possible only via an intricate "see-saw" construction that requires $O(n^2)$ PoW-based rounds, and collateral of $O(qn^2)$ coins from each party [27]. Moreover, the stateless nature of Bitcoin transactions entail a global timeout after which the entire see-saw construction expires. Setting the global timeout to a high value enables many off-chain SFE executions, but also implies that a DoS attack by a corrupt party (who would abort before signing any secret shares of the output

of the first execution) will cause each honest party to wait for a long time before being able to regain possession of her $O(qn^2)$ coins deposit. Due to the time value of money, this is obviously undesirable. The stateful approach does not require a global timeout that is measured in *absolute* terms. Instead, the contract remains operational for as long as all the parties wish to engage in the off-chain protocol, and transitioning the contract into the "payout" state will trigger an event whose expiration is *relative* to the time at which the transition occurred.

We stress that a corrupt party can always pretend to be inactive and force honest parties to interact with the cryptocurrency network. Thus, for example, this protocol can be combined with a reputation system. Further, our implementation uses a technique that shares the on-chain transaction fees among all the parties equally, so that corrupt parties always have to pay the fee (cf. Sect. 6). An Ethereum implementation of this contract is provided in [9, Fig. 18].

Can stateful contracts provide even more benefits? As the next section shows, the answer is "yes".

2.3 Stateful Off-Chain Cash Distribution Protocols

Suppose that the parties P_1, P_2 wish to play a multiple-round lottery game, such that either of them is allowed to quit after each round. Thus, P_1 enters the lottery with m coins, P_2 enters with w coins, and in the first round P_1 picks a random secret x_1 and commits to $\mathsf{com}(x_1)$, P_2 picks x_2 and commits to $\mathsf{com}(x_2)$, and then they decommit x_1, x_2 so that the least significant bit of $x_1 \oplus x_2$ decides whether P_1's balance is incremented to $m+1$ and P_2's balance is decremented to $w-1$, or P_1's balance is decremented to $m-1$ and P_2's balance is incremented to $w+1$. If both of them wish to continue, then they will proceed to the next round and repeat this protocol.

Obviously, the parties must not be allowed to quit in the middle of a round without repercussions. That is, if P_1 reveals her decommitment x_1 and P_2 aborts, then P_1 should be compensated. Moreover, as in Sect. 2.2, it is better that the parties play each round without any on-chain interaction.

Therefore, in each round i, P_1 and P_2 will sign the current balance m_i, w_i together with the round index i and the commitments $\mathsf{com}(x_{i,1}), \mathsf{com}(x_{i,2})$. After the parties exchange these signed messages, they can safely send their decommitments $x_{i,1}, x_{i,2}$ in the clear. The logic of the stateful contract allows each party to send her decommitment along with the signed message, and thus finalize the game according to the current balances. If the other party does not reveal her decommitment during a waiting period, then the contract increments the balance of the honest party. If both parties reveal, then the contract computes $x_1 \oplus x_2$ to decide who won the last round, so that the balance of the winner is incremented and the balance of the loser is decremented. During the waiting period, an honest party can send a signed message with an index $i' > i$ and thereby invalidate the message that a corrupt party sent to the contract.

It should also be noted that an honest party should not continue to play after the balance of the other party reaches 0, since the contract cannot reward the winner more money than what was originally deposited.

We illustrate the contract in Fig. 3, and provide an Ethereum implementation in [9, Figs. 18–20]. The multiparty version of our lottery code is available at [1].

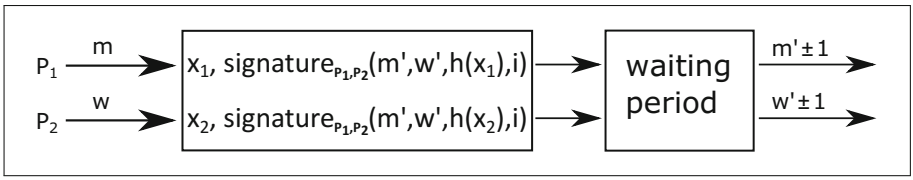

Fig. 3. Off-chain 2-party lottery.

Such a 2-party lottery is a very simple example of a *secure cash distribution with penalties* (SCD) functionality [26]. Another special case of SCD is a multiparty poker game in which money (i.e., coins of the cryptocurrency system that the parties hold) is transferred from losers to winners. In Sect. 3 and onwards we formulate the ideal functionalities and protocol for fair MPC and SCD, and in Sect. 6 we provide an efficient off-chain poker protocol with implementation.

As noted in Sect. 1, SCD can also be realized in the $\mathcal{F}_{\text{CR}}^{\star}$ model via a non-amortized (i.e., on-chain) protocol, though the construction requires a setup phase with $O(n^2)$ PoW-based rounds. To the best of our knowledge, there is no amortized SCD realization in the $\mathcal{F}_{\text{CR}}^{\star}$ model.

A slight variation can turn the above lottery contract into a bi-directional off-chain micropayment channel, see [9, Sect. 2.4].

3 Preliminaries

We say that a function $\mu(\cdot)$ is negligible in λ if for every polynomial $p(\cdot)$ and all sufficiently large λ's it holds that $\mu(\lambda) < 1/|p(\lambda)|$. A probability ensemble $X = \{X(t,\lambda)\}_{t\in\{0,1\}^*,n\in\mathbb{N}}$ is an infinite sequence of random variables indexed by a and $\lambda \in \mathbb{N}$. Two distribution ensembles $X = \{X(t,\lambda)\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y(t,\lambda)\}_{\lambda\in\mathbb{N}}$ 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 $t \in \{0,1\}^*$,

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

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 [8]. Note that the main difference from standard definitions of secure computation [21] is that now the view of $\mathcal{Z}$ contains the distribution of coins. Let $\text{IDEAL}_{f,\mathcal{S},\mathcal{Z}}(\lambda, z)$ 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 $\text{HYBRID}^{g}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$, 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 $\mathcal{Z}$,*

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

Definition 2. *Let π be a protocol and f be a multiparty functionality. We say that π* securely computes *f* with penalties *if π SCC-realizes the functionality $\mathcal{F}^{\star}_{f}$ according to Definition 1.*

Throughout this paper, we deal only with static adversaries and impose no restrictions on the number of parties that can be corrupted. Our schemes also make use of a digital signature scheme which we denote as (SigKeyGen, SigSign, SigVerify). Please see [8,26–28] for additional details on the model including the necessary modifications to UC. Please also see [24] which extensively treats these modifications, proposes alternative models, and uses protocol compilers different from GMW.

3.1 Ideal Functionalities

Secure computation with penalties—multiple executions. We now present the functionality $\mathcal{F}^{*}_{\text{MSFE}}$ which we wish to realize. Recall that secure computation with penalties guarantees the following.

- An honest party never has to pay any penalty.
- If a party aborts after learning the output and does not deliver output to honest parties, then *every* honest party is compensated.

See Fig. 4 for a formal description. Note that $\mathcal{F}^{*}_{\text{MSFE}}$ directly realizes multiple invocations of *non-reactive* secure computation with penalties. In the first phase referred to as the *deposit phase*, the functionality $\mathcal{F}^{*}_{\text{MSFE}}$ accepts safety deposits $\mathsf{coins}(d)$ from each honest party and penalty deposit $\mathsf{coins}(hq)$ from the adversary. Note that the penalty deposit suffices to compensate each honest party n the event of an abort. Once the deposits are made, parties enter the next phase referred to as the *execution phase* where parties can engage in unbounded number of secure function evaluations. In each execution, parties submit inputs and wait to receive outputs. As usual, the ideal adversary $\mathcal{S}$ gets to learn the output first

Notation: session identifier sid, parties $P_1, \ldots, P_n$, adversary $\mathcal{S}$ that corrupts $\{P_s\}_{s \in C}$, *safety deposit* d, *penalty amount* q. Set $H = [n] \setminus C$ and $h = |H|$.

DEPOSIT PHASE: Initialize $\mathsf{flg} = \perp$. Wait to get message $(\mathsf{setup}, sid, ssid, j, \mathsf{coins}(d))$ from P_j for all $j \in H$. Wait to get message $(\mathsf{setup}, sid, ssid, \mathsf{coins}(hq))$ from $\mathcal{S}$.

EXECUTION PHASE: Set $\mathsf{flg} = 0$. For $\mathsf{id} = 1, 2, \cdots$, sequentially do:

- If a message (exit, sid) is received from P_j, then send (exit, sid, j) to all parties, and go to the claim phase.
- Wait to receive a message $(\mathsf{input}, sid, ssid \| \mathsf{id}, j, y_j^{(\mathsf{id})}, g^{(\mathsf{id})})$ from P_j for all $j \in H$. Then send $(\mathsf{function}, sid, ssid \| \mathsf{id}, g^{(\mathsf{id})})$ to all parties.
- Wait to receive a message $(\mathsf{input}, sid, ssid \| \mathsf{id}, \{y_s^{(\mathsf{id})}\}_{s \in C}, g^{(\mathsf{id})})$ from $\mathcal{S}$. If no such message was received, then go to the claim phase.
- Compute $z^{(\mathsf{id})} \leftarrow g^{(\mathsf{id})}(y_1^{(\mathsf{id})}, \ldots, y_n^{(\mathsf{id})})$ and send $(\mathsf{output}, sid, ssid \| \mathsf{id}, z^{(\mathsf{id})})$ to $\mathcal{S}$.
- If $\mathcal{S}$ returns $(\mathsf{continue}, sid, ssid \| \mathsf{id})$, send $(\mathsf{output}, sid, ssid \| \mathsf{id}, z^{(\mathsf{id})})$ to each P_j.
- Else if $\mathcal{S}$ returns $(\mathsf{abort}, sid, ssid)$, update $\mathsf{flg} = 1$, and go to the claim phase.

CLAIM PHASE:

- If $\mathsf{flg} = 0$ or $\perp$, send $(\mathsf{return}, sid, ssid, \mathsf{coins}(d))$ to P_j for $j \in H$. If $\mathsf{flg} = 0$, send $(\mathsf{return}, sid, ssid, \mathsf{coins}(hq))$ to $\mathcal{S}$.
- Else if $\mathsf{flg} = 1$, send $(\mathsf{penalty}, sid, ssid, \mathsf{coins}(d+q+q_i))$ to P_i for all $i \in H$ where $q_i = 0$ unless $\mathcal{S}$ sent a message $(\mathsf{extra}, sid, ssid, \{(i, q_i)\}_{i \in H}, \mathsf{coins}(\sum_{i \in H} q_i))$.

Fig. 4. Special ideal functionality $\mathcal{F}^*_{\text{MSFE}}$ for multiple sequential SFE with penalties.

and then decide whether to deliver the output to all parties. If $\mathcal{S}$ decides to abort, then no further executions are carried out, parties enter the *claim phase*, and honest parties get $\mathsf{coins}(d + q)$, i.e., their safety deposit plus the penalty amount. Now if $\mathcal{S}$ never aborts during a local execution, then the safety deposits are returned back to the honest parties, and $\mathcal{S}$ gets back its penalty deposit. Note that we explicitly allow an exit command that enables parties to exit the contract immediately. Prior works [26,27] required parties to wait for a pre-specified time out parameter before parties can reclaim their deposits.

Supporting cash distribution via $\mathcal{F}^*_{\text{MSCD}}$. See Fig. 5 for a formal definition for $\mathcal{F}^*_{\text{MSCD}}$. The definition for amortized secure cash distribution with penalties in the reactive setting $\mathcal{F}^*_{\text{MSCD}}$ is identical to $\mathcal{F}^*_{\text{MSFE}}$ except that we repeatedly evaluate reactive functions which is composed of multiple stage functions. Now, $\mathcal{S}$ can abort between different stages of a reactive function evaluation or within a single stage. In either case, the honest parties will be compensated via the penalty deposit $\mathsf{coins}(hq)$ submitted by $\mathcal{S}$ in the deposit phase. Furthermore, $\mathcal{F}^*_{\text{MSCD}}$ also supports cash distribution which makes it useful for many applications. In particular, we allow parties to dynamically add deposits. The reactive functions that are evaluated take into account the current balance which is maintained in the variable $\boldsymbol{b}$, and the output of the evaluations update $\boldsymbol{b}$ to reflect how the cash is redistributed. That is, the amount of coins are specified as input to the reactive

functions, and the output will influence how the coins are redistributed. Finally we note that unlike prior formulations [26,27], we do not require an apriori upper bound on the *number of stages* that is common to the reactive functions supported by $\mathcal{F}^*_{\text{MSCD}}$. Like most prior works (with the exception of [25]), we do not attempt to hide the (updated) balance vectors or the amount of coins.

3.2 Stateful Contract — Ideal Functionality

We now present the functionality $\mathcal{F}^*_{\text{StCon}}$ which we use to abstract the smart contracts functionality provided by cryptocurrencies. At a high level, $\mathcal{F}^*_{\text{StCon}}$ lets parties run a time-dependent stateful computation. In other words, the contract encodes a finite state computation where each transition could potentially be dependent on the *time of the transition*. Time-dependent transitions in our stateful contract functionality allow us to design protocols that can support early termination of contracts which could be a potentially critical feature in certain settings. Such features are not supported by claim-or-refund $\mathcal{F}^\star_{\text{CR}}$.

As is typical in the penalty model, we let parties make an initial deposit to the $\mathcal{F}^*_{\text{StCon}}$ functionality. Following this, parties together specify a finite state computation denoted Prog along with an initial state st. The functionality then simply waits for a state transition to be triggered by any of the parties. Upon a valid trigger w (i.e., for which Prog produces non-$\perp$ output), the contract runs Prog on the tuple $(j, w, t; \mathsf{st})$ where j is the index of party P_j who supplies the trigger w at time t, and where st is the current state of Prog. Prog then outputs the current state which will be stored in the variable st and the amount of money e that P_j is supposed to obtain. Both st and e are revealed to all parties (i.e., $\mathcal{F}^*_{\text{StCon}}$ has public state), and $\mathcal{F}^*_{\text{StCon}}$ will distribute $\mathsf{coins}(e)$ to party P_j. The formal description of $\mathcal{F}^*_{\text{StCon}}$ is given in Fig. 6. The functionality repeatedly accepts state transitions until it has distributed all the coins that were deposited to it in the initialization phase.

Analogous to the script complexity definition for $\mathcal{F}^\star_{\text{CR}}$-based protocols, we can define the *script complexity* (also referred to as "validation complexity") of a protocol in the $\mathcal{F}^*_{\text{StCon}}$-hybrid model. See [9, Sect. 3.2] for more details.

Definition 3 ($\mathcal{F}^*_{\text{StCon}}$ Script Complexity). *Let* Π *be a protocol among* $P_1, \ldots, P_n$ *in the* $\mathcal{F}^*_{\text{StCon}}$*-hybrid model where* $\mathcal{F}^*_{\text{StCon}}$ *is initialized with program* Prog *and initial state* st*. We say a trigger* $T = (j, w, t)$ *is valid iff* $\mathsf{Prog}(j, w, t; \mathsf{st}) \neq \perp$ *where* st *is the current state of* $\mathcal{F}^*_{\text{StCon}}$*. We say a state* st *is valid iff there exists a valid sequence of triggers starting from some initial state* st_0 *that result in* st *becoming the current state of* $\mathcal{F}^*_{\text{StCon}}$*. For a trigger* T *acting on input state* st*, we let* $C(\mathsf{Prog}, T, \mathsf{st})$ *denote the sum of the size of* T*, the size of the input states and the output states and the running time of* Prog *on input* $(T; \mathsf{st})$*. We define the* $\mathcal{F}^*_{\text{StCon}}$-validation complexity *of* Π*, or in short* transition validation complexity *of* Π *as the maximum value of* $C(\mathsf{Prog}, T, \mathsf{st})$ *maximized over all possible choices of a valid trigger* T *and a valid state* st*.* $\diamondsuit$

Functionality $\mathcal{F}^*_{\text{StCon}+}$. We also present another variant of $\mathcal{F}^*_{\text{StCon}}$, which we call $\mathcal{F}^*_{\text{StCon}+}$. The main difference is that $\mathcal{F}^*_{\text{StCon}+}$ accepts coin deposits, via

Notation: session identifier sid, parties $P_1, \ldots, P_n$, adversary $\mathcal{S}$ that corrupts $\{P_s\}_{s \in C}$, *safety deposit* d, *penalty amount* q, set $H = [n] \setminus C$.

DEPOSIT PHASE: Initialize $\mathsf{flg} = \bot$.

- Wait to receive a message $(\mathsf{setup}, sid, ssid, j, \mathsf{coins}(d))$ from P_j for all $j \in H$.
- Wait to receive a message $(\mathsf{setup}, sid, ssid, \mathsf{coins}(hq))$ from $\mathcal{S}$ where $h = |H|$.

EXECUTION PHASE: Initialize $\mathsf{flg} = 0$ and $\boldsymbol{b} \leftarrow \mathbf{0}$. For $\mathsf{id} = 1, 2, \ldots$, sequentially do:

- If a message $(\mathsf{exit}, sid, ssid)$ is received from P_j, then send $(\mathsf{exit}, sid, ssid, j)$ to all parties, and go to the claim phase.
- If a message $(\mathsf{addmoney}, sid, ssid \| \mathsf{id}, b_j, \mathsf{coins}(b_j))$ is received from some P_j, and a message $(\mathsf{addmoney}, sid, ssid \| \mathsf{id}, b_j)$ was received from every P_k with $k \neq j$, then send $(\mathsf{addmoney}, sid, ssid \| \mathsf{id}, b_j)$ to all parties. Update $\boldsymbol{b} \leftarrow \boldsymbol{b} + (\cdots, 0, b_j, 0, \cdots)$.
- Initialize $\mathsf{state} = \bot$. Wait to receive message $(\mathsf{function}, sid, ssid \| \mathsf{id}, g^{(\mathsf{id})})$ from P_j for all $j \in H$. Then send $(\mathsf{function}, sid, ssid \| \mathsf{id}, g^{(\mathsf{id})})$ to all parties.
- Parse $g^{(\mathsf{id})} = \{g_k^{(\mathsf{id})}\}_{k \in [\rho]}$. For $k = 1, \ldots, \rho$, sequentially do:
  - Wait to receive message $(\mathsf{input}, sid, ssid \| \mathsf{id} \| k, j, y'_j)$ from P_j for all $j \in H$.
  - Wait to receive a message $(\mathsf{input}, sid, ssid \| \mathsf{id} \| k, \{y'_s\}_{s \in C})$ from $\mathcal{S}$. If no such message was received, update $\mathsf{flg} = 1$ and go to the claim phase.
  - Compute $(z, \boldsymbol{b}', \mathsf{state}) \leftarrow g_k^{(\mathsf{id})}(y'_1, \ldots, y'_n; \mathsf{state}, \boldsymbol{b})$.
  - Send message $(\mathsf{output}, sid, ssid \| \mathsf{id} \| k, z, \boldsymbol{b}')$ to $\mathcal{S}$.
  - If $\mathcal{S}$ sends $(\mathsf{continue}, sid, ssid \| \mathsf{id} \| k)$, send $(\mathsf{out}, sid, ssid \| \mathsf{id} \| k, z, \boldsymbol{b}')$ to all P_i.
  - If $\mathcal{S}$ returns $(\mathsf{abort}, sid, ssid \| \mathsf{id} \| k)$, set $\mathsf{flg} = 1$, and go to the claim phase.
  - Update $\boldsymbol{b} \leftarrow \boldsymbol{b}'$.

CLAIM PHASE:

- If $\mathsf{flg} = 0$ or $\bot$, send $(\mathsf{return}, sid, ssid, \mathsf{coins}(d + \boldsymbol{b}_r))$ to all P_r for $r \in H$. If $\mathsf{flg} = 0$, send $(\mathsf{return}, sid, ssid, \mathsf{coins}(hq + \sum_{s \in C} \boldsymbol{b}_s))$ to $\mathcal{S}$.
- Else if $\mathsf{flg} = 1$, send $(\mathsf{penalty}, sid, ssid, \mathsf{coins}(d + q + \boldsymbol{b}_i + q_i))$ to P_i for all $i \in H$ where $q_i = 0$ unless $\mathcal{S}$ sent a message $(\mathsf{extra}, sid, ssid, \{q_i\}_{i \in H}, \mathsf{coins}(\sum_{i \in H} q_i))$. Send $(\mathsf{remaining}, sid, ssid, \mathsf{coins}(\sum_{s \in C} \boldsymbol{b}_s))$ to $\mathcal{S}$.

Fig. 5. Special ideal functionality $\mathcal{F}^*_{\mathrm{MSCD}}$ for multiple sequential MPC with penalties.

an update command even during the execution phase. We note that dynamic updates to Prog is supported by our definition (although we do not rely on this feature in our protocols). We note that $\mathcal{F}^*_{\mathrm{StCon+}}$ is also supported by Ethereum (Fig. 7).

Notation: session identifier sid, parties $P_1, \ldots, P_n$, *initial deposit vector* $(d_1, \ldots, d_n)$, program Prog and an initial state st. We assume that the initial deposit vector is specified in Prog.

Initialization phase: Wait to get message $(\mathsf{init}, sid, ssid, \mathsf{Prog}, \mathsf{st}, \mathsf{coins}(d_j))$ from P_j for all $j \in [n]$. Initialize $Q \leftarrow \sum_{j \in [n]} d_j$.

Execution phase: Repeat until termination:

- Wait to receive a message $(\mathsf{trigger}, sid, ssid, w)$ from some party P_j at time t such that $\mathsf{Prog}(j, w, t; \mathsf{st}) \neq \bot$. Then, let $(\mathsf{st}, e) \leftarrow \mathsf{Prog}(j, w, t; \mathsf{st})$. If $e < Q$, then send $(\mathsf{output}, sid, ssid, j, w, t, \mathsf{st}, e)$ to each P_k and send $(\mathsf{pay}, sid, ssid, \mathsf{coins}(e))$ to P_j. Update $Q \leftarrow Q - e$. If $Q = 0$, then terminate.

Fig. 6. Special ideal functionality $\mathcal{F}^*_{\mathrm{StCon}}$ for stateful contracts.

Notation: session identifier sid, set of parties $\{P_1, \ldots, P_n\}$, *initial deposit vector* $d_1 = \cdots d_n = (n-1)q$, where q is the penalty amount, a program Prog, an initial state st, and a validation function for updates Update. During the execution phase, parties will be able to add coins via Update.

Initialization phase: Wait to get $(\mathsf{init}, sid, ssid, \mathsf{Prog}, \mathsf{Update}, \mathsf{st}, d_j, \mathsf{coins}(d_j))$ from each P_j. Initialize $Q \leftarrow \sum_{j \in [n]} d_j$.

Execution phase: Repeat until termination:

- If a message $(\mathsf{update}, sid, ssid, u, b, \mathsf{coins}(b))$ is received from P_j at time t such that $\mathsf{Update}(j, w, t; \mathsf{st}) \neq \bot$, then set $(\mathsf{Prog}, \mathsf{Update}, \mathsf{st}) \leftarrow \mathsf{Update}(j, u, t; \mathsf{Prog}, \mathsf{st})$, accept $\mathsf{coins}(b)$, update $Q \leftarrow Q + b_j$, and send $(\mathsf{update}, sid, ssid, j, u, t, \mathsf{st}, \mathsf{Prog}, \mathsf{Update})$ to all parties.
- If a message $(\mathsf{trigger}, sid, ssid, w)$ is received from P_j at time t such that $\mathsf{Prog}(j, w, t; \mathsf{st}) \neq \bot$. Then, let $(\mathsf{st}, e) \leftarrow \mathsf{Prog}(j, w, t; \mathsf{st})$. If $e < Q$, then send $(\mathsf{output}, sid, ssid, j, w, t, \mathsf{st}, e)$ to each P_k and send $(\mathsf{pay}, sid, ssid, \mathsf{coins}(e))$ to P_j. Update $Q \leftarrow Q - e$. If $Q = 0$, then terminate.

Fig. 7. Special ideal functionality $\mathcal{F}^*_{\mathrm{StCon+}}$ for stateful contracts.

4 Realizing $\mathcal{F}^*_{\mathbf{MSFE}}$ from $\mathcal{F}^*_{\mathbf{StCon}}$

In this section, we describe the protocols for amortized secure computation with penalties in the $\mathcal{F}^*_{\mathrm{StCon}}$-hybrid model. Due to lack of space we only give a brief overview. See [9, Appendix A] for full details.

The protocol for implementing $\mathcal{F}^*_{\mathrm{MSFE}}$ has three phases. In the first phase, parties interact with the on-chain stateful contract, i.e., the ideal functionality $\mathcal{F}^*_{\mathrm{StCon}}$. In particular, parties agree on setting contract parameters that fix the number of parties, the allowed state transitions of the contract, the time-out, and the compensation amounts to parties in case of an abort (cf. Fig. 9). Then in the

second phase (cf. Fig. 8), parties perform the actual computation. This is done off-chain via local MPC executions. In addition to performing the computation, these MPC executions also provide hooks to the on-chain contract (to handle aborts). We describe the local executions first because they will introduce new variables which will serve as hooks to the contract via the contract parameters. In the next phase, we describe the process which honest parties use in case an off-chain local execution was aborted. In particular, in this phase, parties will go on to the on-chain contract to either continue the aborted local execution or claim their compensation. This phase occurs immediately following an abort in the local executions phase. Figure 10 describes how parties handle notifications received from $\mathcal{F}^*_{\text{StCon}}$.

Local executions. The formal description of the local executions is in Fig. 8. In this section, we describe this phase in more detail. Suppose the parties are interested in computing a function $g^{(\mathsf{id})}$. At a high level, parties begin by running a standard secure computation protocol (that does not guarantee fairness) which computes $z = g^{(\mathsf{id})}(y_1, \ldots, y_n)$ where y_j represents the input of P_j. In addition this secure computation protocol also additively secret shares z into $z_1, \ldots, z_n$ and computes commitments h_j on each z_j using uniform randomness ω_j. Finally, the secure computation protocol outputs $x_j^{(\mathsf{id})} = (z_j; \omega_j)$ (i.e., the decommitment to h_j) and the value $\boldsymbol{h}^{(\mathsf{id})} = h_1 \| \cdots \| h_n$ to each party P_j. For the simulation to work, we need to use an honest-binding commitment scheme (cf. [9, Appendix D]).

Note that there could be aborts here and in every subsequent stage of the local execution. In each step, we assume that parties stop the protocol if they do not receive valid messages (i.e., including signatures) from any other party. Importantly, there is an additional (implicit) time-interval parameter δ which is used to detect aborts. In more detail, we say that a party aborted the protocol if (1) it is its turn to send a message, and (2) if the party does not send a valid protocol message within time-interval δ of the previous event. (We assume that all honest parties act immediately, i.e., within time-interval δ.) In the event of an abort (as defined above), parties go up to the on-chain contract for resolution (cf. Fig. 10). Once the local secure computation protocol ends, we ask each party P_j to compute a signature σ_j on the message $(\mathsf{id}, \boldsymbol{h}^{(\mathsf{id})})$ under its secret signing key sk_j and then broadcast it to all parties. In the next step, each party P_j broadcasts the decommitment $x_j^{(\mathsf{id})}$ to the value h_j contained in $\boldsymbol{h}^{(\mathsf{id})}$ (which in particular includes the secret share z_j of the output z). Once this step is completed, then all parties can recover the output of the id-th computation as $\bigoplus_{k \in [n]} z_k$. Please see [9, Appendix A] for more details.

Contract parameters. See Fig. 9 for a formal description. The state parameters are established in the following way. The variables $d_1, \ldots, d_n$ represent the amount $\mathsf{coins}((n-1)q)$ that the parties are expected to put in as the initial deposit to the contract.

State components and initialization. The variable st denotes the current state of the contract. The values $\boldsymbol{pk}$ and Δ are constant parameters to the contract and

1. (MPC step) Parties run a standard MPC protocol that
   - obtains inputs $y_1, \ldots, y_n$ from the parties; and
   - computes $z = g^{(\mathsf{id})}(y_1, \ldots, y_n)$; and
   - secret shares z into $z_1, \ldots, z_n$; and
   - computes $h_j = \mathsf{com}(x_j^{(\mathsf{id})} = (z_j; \omega_j))$; where each ω_j is chosen uniformly at random; and
   - outputs $(x_j^{(\mathsf{id})}, \boldsymbol{h}^{(\mathsf{id})} = h_1 \| \cdots \| h_n)$ to each P_j.
2. (Signature broadcast) Each P_j computes a signature σ_j on message $(\mathsf{id}, \boldsymbol{h}^{(\mathsf{id})})$ under the signing key sk_j and then broadcasts it. Let S_j denote the set of parties whose signature on message $(\mathsf{id}, \boldsymbol{h}^{(\mathsf{id})})$ was received by P_j. If $S_j = [n]$, then let $\boldsymbol{\sigma}^{(\mathsf{id})} = \sigma_1 \| \cdots \| \sigma_n$ and update $\mathsf{best}_j \leftarrow (\mathsf{id}, ((j, x_j^{(\mathsf{id})}), \boldsymbol{h}^{(\mathsf{id})}, \boldsymbol{\sigma}^{(\mathsf{id})}))$. Else, parties abort the local execution and go to the on-chain contract for resolution (cf. Figure 10).
3. (Share broadcast) Each P_j broadcasts $x_j^{(\mathsf{id})}$. Let $S_j^{(\mathsf{id})}$ denote the set of parties whose share (authenticated against $\boldsymbol{h}^{(\mathsf{id})}$) was received by P_j and let $X_j^{(\mathsf{id})}$ denote the corresponding set of decommitments received by P_j. Each P_j updates $\mathsf{best}_j \leftarrow (\mathsf{id}, (X_j, \boldsymbol{h}, \boldsymbol{\sigma}))$. If $|S_j^{(\mathsf{id})}| = [n]$, then P_j computes the output of the id-th local execution as $\bigoplus_{k=1}^{n} z_k$ where we parse $x_k^{(\mathsf{id})} \in X_j^{(\mathsf{id})}$ as $(z_k; \omega_k)$. Else, parties abort the local execution and go to the on-chain contract for resolution (cf. Figure 10).

Fig. 8. id-th off-chain local execution for implementing $\mathcal{F}^*_{\mathrm{MSFE}}$.

are always maintained as part of the state. The parameter $\boldsymbol{pk}$ represents the set of public keys of all the parties. The parameter Δ represents the length of the time interval within which parties need to act in order to keep the contract from defaulting. In addition, each state variable has five components: (1) $\mathsf{st.mode}$ represents the current mode in which the contract is in, and is one of {"init", "exec", "exit", "payout", "abort", "inactive"}; (2) $\mathsf{st.id}$ represents the id of the execution that is being continued currently on the on-chain contract; (3) $\mathsf{st.TT}$ represents the current transcript of the execution that is being continued on the chain; (4) $\mathsf{st}.t$ represents the time when the on-chain contract was triggered and either (a) was moved to "exit" or (b) resulted in a change of the variable $\mathsf{st.TT}$; (5) $\mathsf{st}.L$ is a boolean array that represents which parties have already withdrawn their deposits and compensations from $\mathcal{F}^*_{\mathrm{StCon}}$.

We represent the state variable st as a five tuple ($\mathsf{st.mode}$, $\mathsf{st.id}$, $\mathsf{st.TT}$, $\mathsf{st}.t$, $\mathsf{st}.L$). Also, $\mathsf{st.TT}$ is either $\perp$ (denoting the null transcript) or is a tuple of the form $(X, \boldsymbol{h}, \boldsymbol{\sigma})$. The initial state is st and its components are initialized in the following way: (1) $\mathsf{st.mode}$ = "init"; (2) $\mathsf{st.id} = -1$; (3) $\mathsf{st.TT} = \mathsf{NULL}$; (4) $\mathsf{st}.t = -1$; and (5) $\mathsf{st}.L = (1, \ldots, 1)$. Recall that st also contains the list of all public keys of the participants $\boldsymbol{pk}$, and the global time-out parameter Δ.

Triggering state transitions. During course of the execution, the state of the contract would either (1) remain in the initial state with $\mathsf{st.mode}$ = "init"; or

(2) be in exit mode, i.e., with $\mathsf{st.mode}$ = "exit", where contract participants are trying to get their initial deposit out of the contract (and terminate the contract); or (3) be trying to continue an incomplete off-chain local execution by keeping track of the current state of the local execution computation, i.e., with $\mathsf{st.mode}$ = "exec"; or (4) be in payout mode with $\mathsf{st.mode}$ = "payout" where parties have successfully completed all executions so far and are waiting to get their initial deposits out of $\mathcal{F}^*_{\mathrm{StCon}}$; or (5) be in "abort" mode where an execution was aborted and honest parties are waiting to get their initial deposits and compensation out of $\mathcal{F}^*_{\mathrm{StCon}}$; or (6) be in "inactive" mode where $\mathcal{F}^*_{\mathrm{StCon}}$ no longer accepts any further state transitions and in particular, has given out all the money that was deposited to it.

Transitions to different states will be triggered by a witness (j, w, t). Here j represents the party triggering the contract, i.e., party P_j. The value t represents the time at which the contract is triggered. Note that when $\mathsf{st.mode}$ = "payout"/"abort"/"inactive", the triggering witness w is simply the token value exit. As we will see later, the transitions from these states depends only on $\mathsf{st}.L$ and the triggering time t and $\mathsf{st}.t$. The more interesting case is when $\mathsf{st.mode}$ = "init"/"exit"/"exec". In this case, the triggering witness w provides the most recent state of the current local execution. We will use a separate subroutine pred to determine the validity of a trigger (j, w, t) when the witness w represents a transcript of an execution.

Subroutine pred**.** The predicate pred essentially decides if a trigger to the contract is a valid continuation of the computation on the chain. Now, pred takes a trigger (j, w, t) and examines it in conjunction with the current state of the contract. First, pred parses the witness w as $(\mathsf{id}, \mathrm{TT} = (X, \boldsymbol{h}, \boldsymbol{\sigma}))$ where id represents the (off-chain) execution that is being attempted to be continued on the chain by party P_j. The value $\mathrm{TT} = (X, \boldsymbol{h}, \boldsymbol{\sigma})$ essentially provide (along with a proof) the most recent state of a computation (typically the last incomplete off-chain computation). In particular and in the context of non-reactive functionalities, the value X maintains the set of parties who have completed their step of the computation on the chain along with their broadcasted decommitments to the secret share of the final output. The values $\boldsymbol{h}$ and $\boldsymbol{\sigma}$ essentially authenticate to the contract that values in X are legitimate values corresponding to the id-th off-chain computation. In more detail, $\boldsymbol{h} = h_1 \| \cdots \| h_n$ is the set of commitments (that is public to all parties). The value $\boldsymbol{h}$ should be consistent with the broadcast values X in the sense that $\mathsf{com}(X[k]) = h_k$ for all k such that $X[k] \neq \perp$. Likewise, the commitment values $\boldsymbol{h}$ need to be accompanied with $\boldsymbol{\sigma} = \sigma_1 \| \cdots \| \sigma_n$ where σ_i is the signature of party P_i attesting to the correctness of $\boldsymbol{h}$. Note that the signatures also tie the value of $\boldsymbol{h}$ to id.

Clearly, pred should output 1 if the witness is valid and if (j, w, t) happens to be the very first trigger to the contract. On the other hand, if (j, w, t) is not the first trigger to the contract, then we have to ensure that the trigger (j, w, t) provides a valid update to the contract state. Now the contract state could be in exit mode, i.e., $\mathsf{st.mode}$ = "exit", and in this case the trigger (j, w, t) with a valid witness w could be by an honest party to continue an incomplete

Notation. The variable st denotes the current state of the contract. We represent the state variable st as a five tuple $(\mathsf{st.mode},\ \mathsf{st.id},\ \mathsf{st.}\textsc{tt},\ \mathsf{st.}t,\ \mathsf{st.}L)$. We omit $\boldsymbol{pk}$ and Δ from the state to keep the presentation simple. The initial deposits are $d_1 = \cdots = d_n = (n-1)q$. The initial state is ("init", -1, $\perp$, -1, $\mathbf{1}$).

Subroutine pred. Let $\mathsf{pred}(j, w, t; \mathsf{st}) = 1$ if

- w is parsed as $(\mathsf{id}, \textsc{tt} = (X, \boldsymbol{h}, \boldsymbol{\sigma}))$ with $\boldsymbol{h} = h_1 \| \cdots \| h_n$, $\boldsymbol{\sigma} = \sigma_1 \| \cdots \| \sigma_n$; and
- for each $k \in [n]$ it holds that $\mathsf{SigVerify}((\mathsf{id}, \boldsymbol{h}), \sigma_k; pk_k) = 1$; and
- for each k such that $X[k] \neq \mathsf{NULL}$ it holds that $h_k = \mathsf{com}(X[k])$; and
- either (1) $\mathsf{st.mode}$ = "init"; or (2) $\mathsf{st.mode}$ = "exec"/"exit" and $t \leq \mathsf{st.}t + \Delta$ and either (2.1) $\mathsf{id} > \mathsf{st.id}$ or (2.2) $\mathsf{id} = \mathsf{st.id}$ and $X \not\subseteq \mathsf{st.}\textsc{tt}.X$.

State transitions. $\mathsf{Prog}(j, w, t; \mathsf{st})$: Initialize $e \leftarrow 0$.

- If $w = (\mathsf{id}, \textsc{tt})$ and $\mathsf{pred}(j, w, t; \mathsf{st}) = 1$: If $\mathsf{st.id} = \mathsf{id}$, then update $\mathsf{st.}\textsc{tt}.X \leftarrow \mathsf{st.}\textsc{tt}.X \cup \textsc{tt}.X$, else set $\mathsf{st.}\textsc{tt} \leftarrow \textsc{tt}$. Set $\mathsf{st} \leftarrow$ ("exec", id, $\mathsf{st.}\textsc{tt}$, t, $\mathsf{st.}L$).
- Else if $w = \mathsf{exit}$:
  - If (1) $\mathsf{st.mode}$ = "init" or (2) $\mathsf{st.mode}$ = "exec" and $|\mathsf{st.}\textsc{tt}.X| = n$: Set $\mathsf{st.mode} \leftarrow$ "exit" and $\mathsf{st.}t \leftarrow t$.
  - If $\mathsf{st.mode}$ = "exec" or "abort", and $t > \mathsf{st.}t + \Delta$ and $\mathsf{st.}L[j] = 1$ and $|\mathsf{st.}\textsc{tt}.X| \neq n$: Then update $\mathsf{st.}L[j] \leftarrow 0$ and $\mathsf{st.mode} \leftarrow$ "abort" and $\mathsf{st.}L[k] \leftarrow 0$ for all k such that $\mathsf{st.}\textsc{tt}.X[k] = \perp$. Further, if $\mathsf{st.}\textsc{tt}.X[j] \neq \perp$, then set $e \leftarrow n(n-1)q/|\mathsf{st.}\textsc{tt}.X|$.
  - If $\mathsf{st.mode}$ = "exit" or "payout", and $t > \mathsf{st.}t + \Delta$ and $\mathsf{st.}L[j] = 1$: Set $e \leftarrow (n-1)q$ and update $\mathsf{st.mode} \leftarrow$ "payout" and $\mathsf{st.}L[j] \leftarrow 0$.

If $\mathsf{st.}L[k] = 0$, for all $k \in [n]$ then we update $\mathsf{st.mode} \leftarrow$ "inactive".

Fig. 9. $\mathcal{F}^{*}_{\mathrm{StCon}}$ parameters for $\mathcal{F}^{*}_{\mathrm{MSFE}}$.

off-chain execution. This is to ensure that a malicious party cannot subvert the continuation of the off-chain execution on the on-chain contract by trying to exit prematurely (i.e., when $\mathsf{st.mode}$ = "init"). Likewise a malicious party might also submit an old completed execution (even while the current off-chain execution has not yet completed). Thus, we must have pred output 1 when the new id present in w is greater than $\mathsf{st.id}$.

Now the contract could be in exec mode, i.e., $\mathsf{st.mode}$ = "exec", in which case the contract is typically waiting for the on-chain execution to be completed. There are essentially two cases: (1) the current state does not correspond to or continue the most recent off-chain execution; in this case, the id in the new trigger must satisfy $\mathsf{id} > \mathsf{st.id}$ (i.e., the contract is essentially reset to the "correct last computation"), and (2) the new trigger continues the current state of the contract and for this $\mathsf{id} = \mathsf{st.id}$ must hold and also we need X to contain at least one value which is not in $\mathsf{st.}\textsc{tt}.X$, i.e., there is some $k \in [n]$ such that $X[k] \neq \perp = \mathsf{st.}\textsc{tt}.X$. In either case, the new trigger must appear within the time interval Δ of the previous trigger (i.e., before time $\mathsf{st.}t + \Delta$).

State transitions. The state transition function Prog takes as input the trigger (j, w, t) and the current state st. First, we check if the witness provided corresponds to an execution transcript. In this case, we invoke the predicate pred and if pred outputs 1, then we update $\mathsf{st}.\text{TT}$ depending on whether (1) $\mathsf{st.id} = \mathsf{id}$, in which case we update $\mathsf{st}.\text{TT}.X$ to include decommitments specified in X; or (2) $\mathsf{st.id} < \mathsf{id}$, in which case we update $\mathsf{st}.\text{TT} \leftarrow \text{TT}$. If w does not correspond to an execution transcript, then we assume that it is a token value exit. There are effectively three cases to handle:

- If st.mode equals "init" or equals "exec" with a fully completed transcript, then we change st.mode to "exit" and store the triggering time t in $\mathsf{st}.t$. This transition is provided to ensure that honest parties' deposits are not "locked in" and to enable them to withdraw their deposits from $\mathcal{F}^*_{\text{StCon}}$.
- If $\mathsf{st.mode} =$ "exec"/"abort", then we check if $t > \mathsf{st}.t + \Delta$ and if $|\mathsf{st}.\text{TT}.X| \neq n$ to confirm that the execution has indeed been aborted. In this case, if $\mathsf{st}.L[j] = 1$, then we will allow P_j to take money out of $\mathcal{F}^*_{\text{StCon}}$. We further need to check whether P_j was a malicious party that did not contribute to completing the execution. We do this by checking if $\mathsf{st}.\text{TT}.X[j] \neq \perp$. If all checks pass, we let P_j to withdraw its initial deposit plus compensation, i.e., a total of $n(n-1)q/|\mathsf{st}.\text{TT}.X|$ from $\mathcal{F}^*_{\text{StCon}}$.
- If $\mathsf{st.mode} =$ "exit"/"payout", then we check if $t > \mathsf{st}.t + \Delta$. This is to prevent situations where a malicious party tries to subvert continuing the off-chain aborted execution on the chain. (That is, honest parties get an additional time Δ to get $\mathcal{F}^*_{\text{StCon}}$ out of the exit mode.) If $t > \mathsf{st}.t + \Delta$ indeed holds, then we allow party P_j to take its initial deposit $(n-1)q$ out of the contract if it was not already paid before (i.e., $\mathsf{st}.L[j] = 1$).

Main protocol. The formal description can be found in Fig. 10. In this section we will describe the main protocol that makes use of the local execution subprotocol of Fig. 8 and also how parties interact with $\mathcal{F}^*_{\text{StCon}}$ according to the parameters described in Fig. 9. Parties basically start by initializing the $\mathcal{F}^*_{\text{StCon}}$ parameters as in Fig. 9. Following this, they begin off-chain local executions. Recall that each party P_j maintains a variable best_j which denotes the transcript corresponding to the latest *active* execution (both on-chain and off-chain) *according to the local view of party* P_j (see Fig. 8). This value will provide the necessary hook to the on-chain contract to handle off-chain aborts. In particular, the value best_j will be submitted by party P_j in order to recover from aborted off-chain executions.

In our main protocol, we essentially deal with three different scenarios: (1) when parties want to exit the contract and get back their deposits and compensation, (2) when parties want to continue an aborted off-chain execution on the chain, and (3) when parties are notified of state changes in $\mathcal{F}^*_{\text{StCon}}$.

Exiting the contract. First, we deal with the situation when parties would like to terminate the protocol and retrieve their initial deposits from the contract. To do so, we simply let parties submit a token value $w = \mathsf{exit}$ to trigger and put

Parties initialize the parameters as in Figure 9. Then for $\mathsf{id} = 1, 2, \cdots$, parties run the local execution prescribed in Figure 8. Recall that each party P_j maintains a variable best_j during the local executions which is initialized as $\perp$.

1. If a party P_j wants to exit the contract and reclaim its initial deposit, then it sends $w = \mathsf{exit}$ to $\mathcal{F}^*_{\mathrm{StCon}}$.
2. If there is an abort during a local off-chain execution, then each party P_j triggers $\mathcal{F}^*_{\mathrm{StCon}}$ with the value best_j.
3. Each party P_j waits and responds to state changes in $\mathcal{F}^*_{\mathrm{StCon}}$ depending on the current state st:
   (a) If $\mathsf{st.mode}$ = "payout"/"abort" and $\mathsf{st}.L[j] = 1$, send $w = \mathsf{exit}$ to $\mathcal{F}^*_{\mathrm{StCon}}$.
   (b) If (1) $\mathsf{st.id} < \mathsf{best}_j.\mathsf{id}$, or (2) $\mathsf{st.id} = \mathsf{best}_j.\mathsf{id}$ and $\mathsf{st}.\mathrm{TT}.X[j] = \perp$, then submit best_j to $\mathcal{F}^*_{\mathrm{StCon}}$.
   (c) If $\mathsf{st.id} > \mathsf{best}_j.\mathsf{id}$, then submit $\mathsf{best}_j \leftarrow (\mathsf{st.id}, ((j, x_j^{(\mathsf{st.id})}), \mathsf{st}.\mathrm{TT}.\boldsymbol{h}, \mathsf{st}.\mathrm{TT}.\boldsymbol{\sigma}))$ to $\mathcal{F}^*_{\mathrm{StCon}}$.

Finally, parties also keep track of whether $\mathsf{st.mode}$ = "exit" or "exec" and the current time t is such that $t > \mathsf{st}.t + \Delta$. In this case, parties send $w = \mathsf{exit}$ in order to claim their payout or compensation.

Fig. 10. Main protocol for implementing $\mathcal{F}^*_{\mathrm{MSFE}}$.

the contract into exit mode. Note that malicious parties might revert $\mathcal{F}^*_{\mathrm{StCon}}$ to go into exec mode. In this case, $\mathcal{F}^*_{\mathrm{StCon}}$ will notify honest parties of the change. Honest parties will be able to recover from this and put the contract back into exit mode. This will be described when we discuss how parties react to notifications from $\mathcal{F}^*_{\mathrm{StCon}}$.

Continuing an aborted off-chain execution. This is where the value best_j comes in handy as it stores the most recent state of the off-chain executions. We instruct parties to trigger $\mathcal{F}^*_{\mathrm{StCon}}$ with the value best_j which includes P_j's decommitment $x_j^{(\mathsf{id})}$ which in turn ensures (by the logic in Fig. 9) that P_j will not be penalized.

Responding to notifications from $\mathcal{F}^*_{\mathrm{StCon}}$. First, if $\mathsf{st.mode}$ = "payout"/"abort", then parties send a token value $w = \mathsf{exit}$ to get their deposits out of $\mathcal{F}^*_{\mathrm{StCon}}$. In addition, if $\mathsf{st.mode}$ = "abort", then parties would also get compensation from $\mathcal{F}^*_{\mathrm{StCon}}$. Second, if the on-chain execution does not correspond to the most recent execution, then we ask parties to submit best_j to the contract. (This will also handle the case when honest parties try to exit the contract but a malicious party feeds an older execution to $\mathcal{F}^*_{\mathrm{StCon}}$.) Checking if the on-chain execution corresponds to the most recent execution is handled by checking first if $\mathsf{st.id} < \mathsf{best}_j.\mathsf{id}$ and then if $\mathsf{st.id} = \mathsf{best}_j.\mathsf{id}$ but $\mathsf{st}.\mathrm{TT}.X[j] = \perp$ (i.e., party P_j's decommitment is not yet part of the on-chain execution transcript). Finally, we also need to handle the corner case when $\mathsf{st.id} > \mathsf{best}_j.\mathsf{id}$. This scenario is actually possible when party P_j is honest but only when $\mathsf{st.id} = \mathsf{best}_j.\mathsf{id} + 1$. We now describe the sequence of events which lead to this case. Suppose in Step 2 of

Fig. 8 some malicious party did not broadcast its signature on $\boldsymbol{h}^{(\mathsf{id})}$, then party P_j will not update best_j. Thus $\mathsf{best}_j.\mathsf{id} = \mathsf{id} - 1$ where id is the execution id of the current execution. Note that each honest P_j would have submitted its signature on $\boldsymbol{h}^{(\mathsf{id})}$ in Step 2. Therefore, malicious parties would possess a valid $\boldsymbol{h}^{(\mathsf{id})}$ and $\boldsymbol{\sigma}^{(\mathsf{id})}$ for execution id. That is, a malicious party P_k is able to trigger $\mathcal{F}^*_{\mathrm{StCon}}$ with a witness $w = (\mathsf{id}, \mathrm{TT} = ((k, x_k^{(\mathsf{id})}), \boldsymbol{h}^{(\mathsf{id})}, \boldsymbol{\sigma}^{(\mathsf{id})})$ which will result in $\mathsf{pred}(k, w, t) = 1$. Thus, we need a mechanism to allow honest parties to continue the id-th execution (i.e., continue TT) and ensure that they don't get penalized. This is indeed possible since honest parties already obtain $x_j^{(\mathsf{st.id})}$ from Step 1 of Fig. 8. That is, we let honest parties submit $w = (\mathsf{st.id}, \mathrm{TT}_j = ((j, x_j^{(\mathsf{st.id})}), \boldsymbol{h}^{(\mathsf{id})} = \mathsf{st}.\mathrm{TT}.\boldsymbol{h}, \boldsymbol{\sigma}^{(\mathsf{id})} = \mathsf{st}.\mathrm{TT}.\boldsymbol{\sigma}))$ to $\mathcal{F}^*_{\mathrm{StCon}}$.

Due to lack of space, we prove the following theorem in [9, Appendix A].

Theorem 1. *Let λ be a computational security parameter. Assume the existence of one-way functions. Then there exists a protocol that* SCC-realizes *(cf. Definition 1) $\mathcal{F}^*_{\mathrm{MSFE}}$ in the ($\mathcal{F}^*_{\mathrm{StCon}}$,$\mathcal{F}_{\mathrm{OT}}$)-hybrid model whose script complexity (cf. Definition 3) is independent of the number of secure function evaluations and depends only on the length of outputs of the functions evaluated in $\mathcal{F}^*_{\mathrm{MSFE}}$ and is otherwise independent of them. Furthermore, in the optimistic case when all parties are honest, there are a total of $O(n)$ state transitions each having complexity $O(n\lambda)$.*

5 Realizing $\mathcal{F}^*_{\mathrm{MSCD}}$ from $\mathcal{F}^*_{\mathrm{StCon+}}$

We now discuss how to implement $\mathcal{F}^*_{\mathrm{MSCD}}$. Since we are now dealing with reactive functionalities, we make use of an MPC protocol that handles reactive functionalities (say GMW). Since we are dealing with cash distribution, we will let the MPC protocol take, in addition to regular inputs, values that represent the current balance of each party. Note that this balance could change at the end of each stage of the reactive function evaluation. We stress that the reactive functions take only strings as inputs/outputs (and do not handle coins), and the amount of coins and current balance are merely specified as strings. We assume that the updated balance vectors can be obtained directly from the transcript of the protocol implementing the reactive function.

The overall protocol structure closely mimics our protocol for implementing $\mathcal{F}^*_{\mathrm{MSFE}}$. We give a high level overview of the protocol and detail the main differences from the implementation of $\mathcal{F}^*_{\mathrm{MSFE}}$.

Local executions. See Fig. 11 for a formal description. The local executions begin by allowing parties to add coins to their deposit (which will be redistributed depending on the output of the stage functions of the reactive function) but only between different reactive function evaluations. In order to synchronize (in order to make the simulation go through) and ensure that coins are not added while a reactive function is being evaluated, we ask parties to obtain signatures from all parties. (Then we design $\mathcal{F}^*_{\mathrm{StCon+}}$ such that it accepts coins only when

the submitting party has signatures from all participants.) Then, in the next step, we ask parties to agree on the transcript validation function for the reactive protocol π implementing $g^{(\mathsf{id})}$ that they are going to execute. That is, in the id-th local execution, parties agree on $\mathsf{tv}^{(\mathsf{id})}$ and each party signs this value under its signing key and broadcasts it to all parties. This is different from the previous case where while implementing $\mathcal{F}^*_{\text{MSFE}}$, we needed parties to sign on $\boldsymbol{h}^{(\mathsf{id})}$ in the id-th execution but *after* the secure computation protocol was run. Note that we also need parties to agree on the updated balance vector $\boldsymbol{b}$ before beginning the id-th local execution.

Once the signatures are done, parties begin evaluating each stage of the reactive computation in sequence. Parties then run a secure computation protocol for each stage sequentially until the entire reactive protocol completes. Note that a typical protocol for a reactive computation maintains state across different stages by secret sharing this value among the participants. That is, when parties are ready to begin the secure computation protocol for the next stage, they supply along with the inputs for the new stage also an authenticated secret share of the previous state. Note that the balance vectors are supplied as input to the reactive MPC (in order to calculate the updated balance). We abstract these details and assume that the authenticated secret shares of intermediate states corresponding to party P_j is part of its input y_j for this stage. The next message function nmf takes the current available transcript as input, along with the parties' input for this stage, the randomness, the current balance vector, and also the secret signing key of this party. Note that nmf and tv are such that for every partial transcript TT' such that $|\text{TT}'| = (j-1) \bmod n$ and $\mathsf{tv}(\text{TT}') = 1$, we have that $(m, \sigma) \leftarrow \mathsf{nmf}(\text{TT}'; (y_j, \omega_j, \boldsymbol{b}, sk_j))$ satisfies $\mathsf{tv}(\text{TT} \| (m, \sigma)) = 1$. Observe that nmf produces signed messages that continue that transcript, and tv checks whether messages are signed appropriately, and if the newly extended transcript is valid according to the underlying reactive MPC. Such modifications to the underlying reactive MPC protocol (namely, adding signatures in nmf and verifying them in tv, and getting updated balance vectors) were also present in previous protocols that dealt with the reactive case [26,27]. Like in the implementation of $\mathcal{F}^*_{\text{MSFE}}$, here too we ask each party P_j to maintain a value best_j which essentially maintains the transcript corresponding to the current execution. Note that best_j contains both $\mathsf{tv}^{(\mathsf{id})}$ as well signatures on it from all parties denoted by $\boldsymbol{\sigma}^{(\mathsf{id})}$.

$\mathcal{F}^*_{\text{StCon+}}$ parameters. See Fig. 12 for a formal description. The overall structure is similar to the $\mathcal{F}^*_{\text{StCon}}$ parameters for $\mathcal{F}^*_{\text{MSFE}}$, and in particular we interpret the state st has having multiple components which keep track of the current mode of the state, the current transcript, the current execution id, time of last exec mode transition, and which parties have already withdrawn money from $\mathcal{F}^*_{\text{StCon+}}$. The main addition is now we also explicitly keep track of the transcript validation function of the current execution as part of the state. We denote this variable by $\mathsf{st.tv}$. We also keep track of how much each party deposited in the variable $\mathsf{st}.B$ and the latest redistribution of cash (i.e., before $\mathsf{st.id}$-th execution began) in the variable $\mathsf{st}.\boldsymbol{b}$. As with $\mathcal{F}^*_{\text{StCon}}$ parameters for $\mathcal{F}^*_{\text{MSFE}}$, here too we make use of a subroutine pred that effectively determines if the trigger witness w extends the

1. (Adding new coins) If some party P_j wants to add $\mathsf{coins}(b_j)$ to $\mathcal{F}^*_{\mathrm{StCon+}}$, then it sends $\boldsymbol{b}' \leftarrow \boldsymbol{b} + (\cdots, 0, b_j, 0, \cdots)$ to all parties. Each party P_k then generates a signature $\psi'_k \leftarrow \mathsf{Sign}(\mathsf{id}, \boldsymbol{b}')$ and broadcasts it. If P_j receives signatures from all parties, then P_j sends $u = (\mathsf{update}, \boldsymbol{b}', \boldsymbol{\psi}, \mathsf{coins}(b_j))$ to $\mathcal{F}^*_{\mathrm{StCon+}}$. Other parties wait to receive notification from $\mathcal{F}^*_{\mathrm{StCon+}}$ of the updated balance. If either (1) P_j did not obtain signatures from other parties; or (2) the remaining parties did not receive notification from $\mathcal{F}^*_{\mathrm{StCon+}}$, then all honest parties go to the on-chain contract with the intention of exiting the contract. See Figure 13. On the other hand, if the above step was successfully completed, then parties update $\boldsymbol{b} \leftarrow \boldsymbol{b}'$ and execute the next step.
2. (Parameter agreement) Initialize transcript $\mathrm{TT} = \bot$. Parties agree on the reactive function $g^{(\mathsf{id})} = \{g_k^{(\mathsf{id})}\}_k$ to be executed next. Parties also agree on a specific MPC protocol π using which they will securely compute $g^{(\mathsf{id})}$. Let the transcript verification predicate for this reactive MPC protocol be denoted by $\mathsf{tv}^{(\mathsf{id})}$. We use $|\mathsf{tv}^{(\mathsf{id})}|$ to denote the number of messages in a valid transcript that corresponds to a *completed* execution of $g^{(\mathsf{id})}$. Once they agree on $\mathsf{tv}^{(\mathsf{id})}$, each P_j computes $\sigma'_j \leftarrow \mathsf{Sign}((\mathsf{id}, \mathsf{tv}^{(\mathsf{id})}, \boldsymbol{b}); sk_j)$ and broadcasts this value to all parties. Each party sets $\boldsymbol{\sigma}^{(\mathsf{id})} = (\sigma'_1, \ldots, \sigma'_n)$. If not all signatures were obtained, then parties stop the local execution and go to the on-chain contract for resolution. Else each party P_j updates $\mathsf{best}_j \leftarrow (\mathsf{transcript}, \mathsf{id}, \bot, \mathsf{tv}^{(\mathsf{id})}, \boldsymbol{b}, \boldsymbol{\sigma}^{(\mathsf{id})})$
3. (Reactive MPC execution) Parse $g^{(\mathsf{id})} = \{g_k^{(\mathsf{id})}\}_{k\in[\rho]}$. For $k = 1, \ldots, \rho$:
   - Let $r_k^{(\mathsf{id})}$ denote the number of rounds in a standard (unfair) MPC protocol that implements $g_k^{(\mathsf{id})}$. We denote party P_j's inputs to $g_k^{(\mathsf{id})}$ by y_j and the corresponding randomness by ω_j. For $r = 1, \ldots, r_k^{(\mathsf{id})}$ sequentially:
     - Let $j = r \bmod n$. Party P_j computes its next message $(m_r, \sigma_r) \leftarrow \mathsf{nmf}(\mathrm{TT}; (y_j, \omega_j, \boldsymbol{b}, sk_j))$, and broadcasts (m_r, σ_r) to all parties.
     - If no message was received, then parties abort the local execution and go to the on-chain contract for resolution. Else, each P_j updates the transcript $\mathrm{TT} \leftarrow \mathrm{TT} \| (m_r, \sigma_r)$ and sets $\mathsf{best}_j \leftarrow (\mathsf{transcript}, \mathsf{id}, \mathrm{TT}, \mathsf{tv}^{(\mathsf{id})}, \boldsymbol{b}', \boldsymbol{\sigma})$.
   - Parties compute the output of $g_k^{(\mathsf{id})}$ using the completed transcript TT. Note that this output specifies the new balance vector $\boldsymbol{b}'$. Parties update $\boldsymbol{b} \leftarrow \boldsymbol{b}'$.

Fig. 11. The id-th off-chain local executions for implementing $\mathcal{F}^*_{\mathrm{MSCD}}$.

state of the current/latest off-chain execution. For the sake of presentation, we allow parties to submit a trigger witness w which extends $\mathsf{st}.\mathrm{TT}$. Alternatively, and in cases where $\mathsf{st}.\mathsf{id}$ does not correspond to the latest execution, we also let parties submit a trigger witness w which provides the entire transcript of an off-chain execution. In this case, pred outputs 1 if the trigger was submitted at time $t \leq \mathsf{st}.t + \Delta$ and if $\mathsf{id} > \mathsf{st}.\mathsf{id}$ or $\mathsf{id} = \mathsf{st}.\mathsf{id}$ but TT is a longer transcript than the (partial) transcript $\mathsf{st}.\mathrm{TT}$.

Parameters. The variable st denotes the current state of the contract. The values $\boldsymbol{pk}$ and Δ are constant parameters to the contract and are always maintained as part of the state. We represent the state variable st as a seven tuple $(\mathsf{st.mode}, \mathsf{st.id}, \mathsf{st.TT}, \mathsf{st}.t, \mathsf{st}.L, \mathsf{st.tv}, \mathsf{st}.\boldsymbol{b}, \mathsf{st}.B)$. We omit $\boldsymbol{pk}$ and Δ and the total deposits so far from the state to keep the presentation simple. The initial deposits are $d_1 = \cdots = d_n = (n-1)q$. The initial state is ("init", -1, $\perp$, -1, $\mathbf{1}$, $\perp$, $\mathbf{0}$, $\mathbf{0}$). We also use a function cash (specified as part of $\mathsf{st.tv}$) that takes a valid transcript TT and j as input, and outputs the amount of coins that need to be given to P_j.

Subroutine pred**.** Let $\mathsf{pred}(j, w, t; \mathsf{st}) = 1$ if

- w is parsed as $(\mathsf{message}, \mathsf{id}, (m, \sigma))$ with $\mathsf{id} = \mathsf{st.id}$, $\mathsf{st.tv}(\mathsf{st.TT} \| (m, \sigma)) = 1$; or
- w is parsed as $(\mathsf{transcript}, \mathsf{id}, \mathrm{TT}, \mathsf{tv}, \boldsymbol{b}, \boldsymbol{\sigma})$ with (1) $\mathsf{tv}(\mathrm{TT}) = 1$; and (2) $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$ and for each $k \in [n]$ it holds that $\mathsf{SigVerify}((\mathsf{id}, \mathsf{tv}, \boldsymbol{b}), \sigma_k; pk_k) = 1$ and either (1) $\mathsf{st.mode} =$ "init"; or (2) $\mathsf{st.mode} =$ "exec"/"exit" and $t \leq \mathsf{st}.t + \Delta$ and either (2.1) $\mathsf{id} > \mathsf{st.id}$ or (2.2) $\mathsf{id} = \mathsf{st.id}$ and $|\mathrm{TT}| > |\mathsf{st.TT}|$.

Adding money. $\mathsf{Update}(j, u, t; \mathsf{Prog}, \mathsf{st})$ is defined as follows: If $u = (\boldsymbol{b}', \boldsymbol{\psi}, \mathsf{coins}(b_j))$, then parse $\boldsymbol{\psi}$ as $(\psi', \ldots, \psi'_n)$ and verify whether $b_j \neq 0$ and $\mathsf{SigVerify}((j, \boldsymbol{b}'), \psi_k; pk_k) = 1$ for all $k \in [n]$. Then check if $\sum_{k\in[n]} \boldsymbol{b}'_k = b_j + \sum_{k\in[n]} \mathsf{st}.B[k]$. If all checks pass and if $\mathsf{st.mode} \notin \{$"exit", "abort", "payout"$\}$, then update $\mathsf{st}.B[j] \leftarrow \mathsf{st}.B[j] + b_j$. Else output $\perp$.

State transitions. $\mathsf{Prog}(j, w, t; \mathsf{st})$ is defined as follows: Initialize $e \leftarrow 0$.

- If $w = (\mathsf{transcript}, \mathsf{id}, \mathrm{TT}, \mathsf{tv}, \boldsymbol{b}, \boldsymbol{\sigma})$ and $\mathsf{pred}(j, w, t; \mathsf{st}) = 1$: Set $\mathsf{st} \leftarrow$ ("exec", id, TT, t, $\mathsf{st}.L$, tv, $\boldsymbol{b}$, $\mathsf{st}.B$).
- Else if $w = (\mathsf{message}, \mathsf{id}, (m, \sigma))$ and $\mathsf{pred}(j, w, t; \mathsf{st}) = 1$: Update $\mathsf{st.TT} \leftarrow \mathsf{st.TT} \| (m, \sigma)$.
- Else if $w = \mathsf{exit}$:
  - If (1) $\mathsf{st.mode} =$ "init", or (2) $\mathsf{st.mode} =$ "exec" and $|\mathsf{st.TT}| = |\mathsf{st.tv}|$: Set $\mathsf{st.mode} =$ "exit", $\mathsf{st}.t \leftarrow t$.
  - If $\mathsf{st.mode} =$ "exec"/"abort", $t > \mathsf{st}.t + \Delta$, $\mathsf{st}.L[j] = 1$, $|\mathsf{st.TT}| \neq |\mathsf{st.tv}|$ and $j \neq j_a = 1 + |\mathsf{st.TT}| \bmod n$: Set $e \leftarrow nq + \mathsf{st}.\boldsymbol{b}_j$, $\mathsf{st}.L[j] \leftarrow 0$, $\mathsf{st}.L[j_a] \leftarrow 0$ and $\mathsf{st.mode} \leftarrow$ "abort".
  - If $\mathsf{st.mode} =$ "exit"/"payout", and $t > \mathsf{st}.t + \Delta$ and $\mathsf{st}.L[j] = 1$: Set $e \leftarrow (n-1)q + \mathsf{cash}(j, \mathsf{st.TT})$, $\mathsf{st.mode} \leftarrow$ "payout", and $\mathsf{st}.L[j] \leftarrow 0$.

If at the end of a transition, it holds that $\mathsf{st}.L[k] = 0$ for all $k \in [n]$, then we update $\mathsf{st.mode} \leftarrow$ "inactive".

Fig. 12. $\mathcal{F}^*_{\mathrm{StCon+}}$ parameters for $\mathcal{F}^*_{\mathrm{MSCD}}$.

The state transition function Prog will make use of pred described above. If pred outputs 1 then, the contract moves into exec mode and records the trigger time and updates the transcript with the transcript contained in the trigger. If the trigger is a token value exit, then if the current mode is either "init" or "exec" and $\mathsf{st.TT}$ is a completed transcript (we check this by checking if $|\mathsf{st.TT}| = |\mathsf{st.tv}|$), then we put the contract into exit mode and record the time. The rest of the

contract specification is quite similar to the $\mathcal{F}^*_{\text{MSFE}}$ case. In particular, when the trigger is a token value $w = \mathsf{exit}$ and if $t > \mathsf{st}.t + \Delta$ and the triggering party has not yet withdrawn money from $\mathcal{F}^*_{\text{StCon+}}$, we move the contract into "payout" or "abort" (depending on the current mode) and refund the deposit (plus current balance plus compensation if applicable) to the triggering party as long as it had not contributed to an off-chain/on-chain aborted execution. To detect whether the triggering party P_j aborted an execution, we simply check if $j = 1+|\mathsf{st}.\text{TT}| \bmod n$ and $|\mathsf{st}.\text{TT}| \neq |\mathsf{st}.\mathsf{tv}|$ holds. In this case, we penalize the party by not giving its deposit back but instead distributing it among the remaining parties. This is slightly different from the $\mathcal{F}^*_{\text{MSFE}}$ case, where we could potentially penalize multiple corrupt parties depending on whether they contributed their output secret share to the execution. Here, on the other hand, we only penalize one party $j_a = 1 + |\mathsf{st}.\text{TT}| \bmod n$. Note that we also redistribute the deposits made by the parties depending on the output of the latest complete execution. If the execution was completed on-chain, then we use the function cash_j (which we assume is a part of tv for simplicity) applied on $\mathsf{st}.\text{TT}$ to determine how much money party P_j is supposed to obtain. On the other hand, if the on-chain execution was also aborted, then (in addition to paying compensation to the honest parties) we distribute the initial deposits depending on the latest balance prior to this execution (which is stored in variable $\mathsf{st}.\boldsymbol{b}_j$).

Finally, the Update function provides an interface for parties to add coins between different reactive MPCs. Upon receiving a witness $u = (\boldsymbol{b}', \boldsymbol{\psi}, \mathsf{coins}(b_j))$, it checks if the provided coins b_j plus the coins already deposited with $\mathcal{F}^*_{\text{StCon+}}$ (i.e., sum of elements in $\mathsf{st}.B$) matches the amount specified in the balance vector (i.e., sum of elements in $\boldsymbol{b}'$). Note that the signatures also include the party index j (to avoid situations where a different party abuses the broadcasted signatures). Also, note that the deposits $\mathsf{st}.B$ only increase which ensures that there are no replay attacks. This is because once $\mathcal{F}^*_{\text{StCon+}}$ starts giving back coins (i.e., $\mathsf{st}.\mathsf{mode} \in \{$"payout", "abort"$\}$), then it does not accept any more coins.

Main protocol: handling aborts and notifications from $\mathcal{F}^*_{\text{StCon+}}$. The formal description can be found in Fig. 13. As with implementing $\mathcal{F}^*_{\text{MSFE}}$, here too parties begin by initializing $\mathcal{F}^*_{\text{StCon+}}$ with the parameters as in Fig. 12, then continue executing off-chain as in Fig. 11. Each party P_j maintains a local variable best_j which represents the most recent transcript of the current off-chain execution. This value will be helpful while recovering from an aborted off-chain execution (cf. Step 2 of Fig. 13). While in the $\mathcal{F}^*_{\text{MSFE}}$ case, party P_j triggered $\mathcal{F}^*_{\text{StCon+}}$ with best_j when its decommitment did not appear in the transcript in $\mathcal{F}^*_{\text{StCon+}}$, here in the $\mathcal{F}^*_{\text{MSCD}}$ case, party P_j triggers $\mathcal{F}^*_{\text{StCon+}}$ with best_j when best_j contains a longer transcript than the one that is current on the contract. Like in the $\mathcal{F}^*_{\text{MSFE}}$ case, we need to handle the corner case when $\mathsf{st}.\mathsf{id} = \mathsf{best}_j.\mathsf{id} + 1$. This happens when honest parties have completed Step 1 of the $\mathsf{st}.\mathsf{id}$-th local execution phase but did not receive signatures on $\mathsf{tv}^{(\mathsf{st}.\mathsf{id})}$ from all corrupt parties. In this case, each party P_j will choose new input and fresh randomness and continue the protocol from the transcript $\mathsf{st}.\text{TT}$. Note that P_j responds only when $j = 1 + |\mathsf{st}.\text{TT}| \bmod n$ (i.e., it is its turn) and when the execution has

not already completed (i.e., $|\mathsf{st}.\mathrm{TT}| \neq |\mathsf{st}.\mathsf{tv}|$). Finally, there is one other case which is unique to $\mathcal{F}^*_{\mathrm{MSCD}}$ implementation. Unlike the $\mathcal{F}^*_{\mathrm{MSFE}}$ case, each party might have to send out multiple messages (corresponding to the reactive MPC protocol) within a single execution. In particular, once the aborted off-chain execution goes on-chain, it remains on-chain (i.e., parties have to respond within time Δ of the previous step in order to avoid paying a penalty) and needs to be completed by the parties.[1] This brings us to the final case where $\mathsf{st}.\mathsf{id} = \mathsf{best}_j.\mathsf{id}$ where in Step 3(d) of Fig. 13, honest party P_j uses the next message function in order to continue the transcript $\mathsf{st}.\mathrm{TT}$. We prove the following theorem in [9, Appendix C].

Theorem 2. *Let λ be a computational security parameter. Assume the existence of enhanced trapdoor permutations. Then there exists a protocol that* SCC-realizes *(cf. Definition 1) $\mathcal{F}^*_{\mathrm{MSCD}}$ in the $(\mathcal{F}^*_{\mathrm{StCon+}}, \mathcal{F}_{\mathrm{OT}})$-hybrid model whose script complexity (cf. Definition 3) is independent of the number of secure computations. Furthermore, in the optimistic case (i.e., all parties are honest), there are a total of $O(n)$ state transitions (i.e., discounting updates) each of complexity $O(n\lambda)$.*

6 Efficient Poker Protocol

A tailor-made protocol for a poker game in which money changes hands was presented by Kumaresan et al. in [26, Sect. 6]. However, that protocol is not efficient enough in practice, due to two distinct reasons. The first reason is that [26, Sect. 6] was designed to work in the $\mathcal{F}^\star_{\mathrm{CR}}$ model, which incurs an expensive setup procedure and does not support off-chain amortization in its SCD variant (cf. Sect. 2). Furthermore, the $\mathcal{F}^\star_{\mathrm{CR}}$ verification circuits that [26, Sect. 6] uses are quite complex and not supported by the current Bitcoin scripting language. The second reason is that preprocessing shuffle protocol that [26, Sect. 6] employs is a generic secure MPC that delivers hash-based commitments to shares of the shuffled cards. It would be impractical to run a general-purpose secure MPC protocol (typically among 3 to 9 parties in a poker game) that performs the shuffle and the hash invocations, and indeed there are special-purpose poker protocols that perform much better.

See Sect. 1.2 for a survey of the various poker protocols. Our implementation uses the poker protocol of Wei and Wang [22,23], which improves an earlier work of Castellà-Roca et al. [12] by using a zero-knowledge proof of knowledge scheme instead of homomorphic encryption.

A potential disadvantage of special-purpose poker protocols is the on-chain verification cost: the generic secure MPC approach would allow us to define an

[1] Alternatively, when a contract goes on-chain, it is possible to make it come back off-chain right after getting the next message from the party that aborted the off-chain execution. Our protocol does not do this but can be easily modified to behave as described above. Note that this modification does not change our theorem statements.

Parties initialize the parameters as in Figure 12. Then for $\mathsf{id} = 1, 2, \cdots$: Parties run the local execution prescribed in Figure 11. Recall that each party P_j maintains a variable best_j which is initialized as $\perp$.

1. If a party P_j wants to exit the contract, then it sends $w = \mathsf{exit}$ to $\mathcal{F}^*_{\mathrm{StCon+}}$.
2. If there is an abort at any stage during a local off-chain execution, then parties do not continue any more local executions and instead trigger $\mathcal{F}^*_{\mathrm{StCon+}}$ with the value best_j if it is non-null.
3. Each party P_j waits and responds to state changes in $\mathcal{F}^*_{\mathrm{StCon+}}$ depending on the current state st:
   (a) If $\mathsf{st.mode}$ = "payout"/"abort" and $\mathsf{st}.L[j] = 1$, send $w = \mathsf{exit}$ to $\mathcal{F}^*_{\mathrm{StCon+}}$.
   (b) If (1) $\mathsf{st.id} < \mathsf{best}_j.\mathsf{id}$, or (2) $\mathsf{st.id} = \mathsf{best}_j.\mathsf{id}$ and $|\mathsf{best}_j.\mathrm{TT}| > |\mathsf{st}.\mathrm{TT}|$, then submit best_j to $\mathcal{F}^*_{\mathrm{StCon+}}$.
   (c) If $\mathsf{st.id} = \mathsf{best}_j.\mathsf{id} + 1$ and $j = 1 + |\mathsf{st}.\mathrm{TT}| \bmod n$, then choose input y_j and use fresh randomness ω_j and compute $(m, \sigma) \leftarrow \mathsf{nmf}(\mathsf{st}.\mathrm{TT}; (y_j, \omega_j, \boldsymbol{b}, sk_j))$ and send $(\mathsf{message}, \mathsf{st.id}, (m, \sigma))$ to $\mathcal{F}^*_{\mathrm{StCon+}}$ and update $\mathsf{best}_j \leftarrow (\mathsf{transcript}, \mathsf{st.id}, \mathsf{st}.\mathrm{TT} \| (m, \sigma), \mathsf{st.tv}, \mathsf{st}.\boldsymbol{b}, \mathsf{st}.\boldsymbol{\sigma})$.
   (d) If $\mathsf{st.id} = \mathsf{best}_j.\mathsf{id}$ and if $j = 1 + |\mathsf{st}.\mathrm{TT}| \bmod n$ and if $|\mathsf{st}.\mathrm{TT}| \neq |\mathsf{st.tv}|$, then compute $(m, \sigma) \leftarrow \mathsf{nmf}(\mathsf{st}.\mathrm{TT}; (y_j, \omega_j, \boldsymbol{b}, sk_j))$ where y_j, ω_j are inputs to the current stage. Send $\mathsf{best}_j \leftarrow (\mathsf{message}, \mathsf{st.id}, (m, \sigma))$ to $\mathcal{F}^*_{\mathrm{StCon+}}$ and update $\mathsf{best}_j \leftarrow (\mathsf{transcript}, \mathsf{st.id}, \mathsf{st}.\mathrm{TT} \| (m, \sigma), \mathsf{st.tv}, \mathsf{st}.\boldsymbol{b}, \mathsf{st}.\boldsymbol{\sigma})$.

Finally, parties also keep track of whether $\mathsf{st.mode}$ = "exit" or "exec" and the current time t is such that $t > \mathsf{st}.t + \Delta$. In this case, parties send $w = \mathsf{exit}$ in order to claim their payout or compensation and do not participate in any further local executions.

Fig. 13. Main protocol implementing $\mathcal{F}^*_{\mathrm{MSCD}}$.

on-chain predicate that verifies that a pre-image (corresponding to a share of a shuffled card) hashes to the commitment value, and penalize a corrupt party who would not supply the correct pre-image. By contrast, the efficient poker protocols rely on constructions that are algebraic in nature, which implies that the on-chain verification predicate would be significantly more complex.

In the case that all parties are honest, on-chain verification will never occur. In the case that corrupt parties deviate, they can force an honest party to send a transaction containing a witness that satisfies the complex on-chain predicate. The on-chain fallback procedure introduces an additional cost in the form of a transaction fee the party supplying the witness pays (to the miners). While the on-chain fallback also introduces a delay that all of the parties would bear, a malicious party may still cause an honest party to pay the fee. Fortunately, the cost of transaction fees is quite minor (cf. [9, Sect. 6]). Still, our implementation provides an improvement by employing a technique that shares the transaction fee across all parties. In Ethereum this is not straightforward, as the initiator of the transaction must provide all of the transaction fees up front; however,

our technique compensates this party by paying funds collected from all of the parties in advance.

By using the efficient scheme of Wei and Wang, the main steps of our SCD poker protocol are as follows:

1. The parties will execute a deck preparation and shuffle protocols, that output group elements (cf. [23, Protocols 1, 3, 4]).
2. The parties will sign an off-chain message that commits to these group elements (cf. [9, Sect. 3.3]). This signed message could later be sent to the on-chain contract, in case that a corrupt party deviates from the protocol.
3. The parties will run the poker game according to the predefined rules, where in each round a specific party performs a valid action (e.g., raise/call/fold).
4. After each round, all the parties will sign an off-chain message that encodes the state of the poker table.
5. When a party draws a private card from the deck, the parties will execute the card drawing protocol of [23, Protocol 6].

Per the above discussion regarding the complexity of the on-chain predicate, it can be seen that the verification procedure for drawing a private card is dominated by a zero-knowledge proof of knowledge of equality of discrete logarithms (cf. [6,37]). To reduce the round complexity and avoid the HVZK concern, in Step 5 the parties will use a non-interactive proof of knowledge. While there are provably secure methods to obtain the NIZK (see [19]), our efficient implementation uses the Fiat-Shamir heuristic.

Our Ethereum implementation of the NIZK verifier is based on the `Secp256k1` ellitpic curve group, the same used in Ethereum and in Bitcoin for digital signatures. The Ethereum language does not provide opcodes for working with elliptic curve points (though such native support is planned [20]). The current Ethereum scripting language features a dedicated opcode for `secp256k1` signature verification, but this opcode is signature-specific and cannot (to our knowledge) be repurposed for the NIZK scheme. Thus we are forced to implement our NIZK the "hard way," making use of an `Secp256k1` elliptic curve library (due to Andreas Olofsson) built from low-level Ethereum opcodes that implement the elliptic curve exponentiation (`_mul`) operation. Our Ethereum implementation, which is shown in [9, Fig. 22], bears an obvious resemblance to the high-level proof of knowledge of discrete logarithms protocol [6,37].

Using the pyethereum simulator framework, we found that the total gas cost of the NIZK verifier is 1.3 M, whereas an Ethereum block has a gas limit of 4.7 M. The cost of the verifier is dominated by the cost of four scalar multiplications. In contrast, the signature verification opcode costs only 3000 gas (a hundred times cheaper), despite performing a scalar multiplication anyway. Thus if Ethereum were modified to support more general elliptic curve arithmetic, we would anticipate a hundred-fold improvement with respect to the transaction fees.

Note that the off-chain signatures in Step 4 include only the current state and not the transcript history, because the proof of knowledge NIZKs do not branch. That is, at a specific round a party will need to provide a NIZK that depends

	NIZK verify	Scalar multiplication	Built-in instruction
Gas cost	1287858	303401	$\leq$3000
Transactions per block	3	15	$\geq$1500

only on public values: the intermediate result of the card drawing protocol, and the group elements that the parties committed in the first step.

The poker protocol of Wei and Wang that we deploy supports all the requirements that were suggested by Crépeau [16]. For example, complete confidentiality of strategy is supported, since the proof of knowledge verification would not reveal the cards at the end of the game. Thus, our implementation enables poker variants such as Texas hold 'em and five-card draw, where private cards are drawn after the game is already in progress. See Wei [22] for benchmarks that measure the running time of the initial shuffling (which is done off-chain).

Our poker implementation is available at [1].

References

1. https://github.com/amiller/instant-poker
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Fair two-party computations via the bitcoin deposits. In: First Bitcoin Workshop, FC (2014)
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: IEEE Security and Privacy (2014)
4. Asokan, N., Shoup, V., Waidner, M.: Optimistic protocols for fair exchange. In: ACM CCS (1997)
5. Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to better — how to make bitcoin a better currency. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 399–414. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32946-3_29
6. Barnett, A., Smart, N.P.: Mental poker revisited. In: Paterson, K.G. (ed.) Cryptography and Coding 2003. LNCS, vol. 2898, pp. 370–383. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40974-8_29
7. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 263–280. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_17
8. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
9. Bentov, I., Kumaresan, R., Miller, A.: Instantaneous decentralized poker. Technical report available at https://arxiv.org/abs/1701.06726 (2017)
10. Buterin, V.: (2013). https://github.com/ethereum/wiki/wiki/White-Paper
11. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS (2001)
12. Castellà-Roca, J., Domingo-Ferrer, J., Riera, A., Borrell, J.: Practical mental poker without a TTP based on homomorphic encryption. In: Johansson, T., Maitra, S. (eds.) INDOCRYPT 2003. LNCS, vol. 2904, pp. 280–294. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-24582-7_21

13. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: STOC, pp. 364–369 (1986)
14. Coleman, J.: State channels. http://www.jeffcoleman.ca/state-channels/
15. Coppersmith, D.: Cheating at mental poker. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 104–107. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_10
16. Crépeau, C.: A secure poker protocol that minimizes the effect of player coalitions. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 73–86. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_8
17. Crépeau, C.: A zero-knowledge Poker protocol that achieves confidentiality of the players' strategy *or* How to achieve an electronic Poker face. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 239–247. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_18
18. Croman, K., et al.: On scaling decentralized blockchains. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 106–125. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_8
19. Damgård, I., Fazio, N., Nicolosi, A.: Non-interactive zero-knowledge from homomorphic encryption. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 41–59. Springer, Heidelberg (2006). https://doi.org/10.1007/11681878_3
20. Ethereum EIP. https://github.com/ethereum/EIPs/pull/213
21. Goldreich, O.: Foundations of Cryptography, vol. 2. Cambridge University Press, Cambridge (2004)
22. Wei, T.J.: Secure and practical constant round mental poker. Inf. Sci. (2014)
23. Wei, T.J., Wang, L.-C.: A fast mental poker protocol. J. Math. Cryptology **6**(1), 39–68 (2012)
24. Kiayias, A., Zhou, H.-S., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 705–734. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_25
25. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE S&P (2016)
26. Kumaresan, R., Moran, T., Bentov, I.: How to use bitcoin to play decentralized poker. In: CCS (2015)
27. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: CCS (2016)
28. Kumaresan, R., Vaikuntanathan, V., Vasudevan, P.N.: Improvements to secure computation with penalties. In: CCS (2016)
29. Lipton, R.: How to cheat at mental poker. In: AMS Short Course on Crypto (1981)
30. Maxwell, G.: (2011). https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment
31. Miller, A., Bentov, I.: Zero-collateral lotteries in bitcoin and ethereum. In: IEEE S&B Workshop (2017). https://arxiv.org/abs/1612.05390
32. Möser, M., Eyal, I., Gün Sirer, E.: Bitcoin covenants. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 126–141. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_9
33. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
34. O'Connor, R., Piekarska, M.: Enhancing bitcoin transactions with covenants. In: Financial Cryptography Bitcoin Workshop (2017)

35. Pinkas, B.: Fair secure two-party computation. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 87–105. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_6
36. Shamir, A., Rivest, R., Adleman, L.: Mental poker. In: The Mathematical Gardener (1981)
37. Shoup, V., Alwen, J.: (2007). http://cs.nyu.edu/courses/spring07/G22.3220-001/lec3.pdf
38. Wood, G.: Ethereum: A secure decentralized transaction ledger (2014). http://gavwood.com/paper.pdf