Amortizing Secure Computation with Penalties

Ranjit Kumaresan
MIT
Cambridge, Massachusetts
vranjit@mit.edu

Iddo Bentov
Cornell University
Ithaca, New York
iddobentov@cornell.edu

ABSTRACT

Motivated by the impossibility of achieving fairness in secure computation [Cleve, STOC 1986], recent works study a model of fairness in which an adversarial party that aborts on receiving output is forced to pay a mutually predefined monetary penalty to every other party that did not receive the output. These works show how to design protocols for *secure computation with penalties* that guarantees that either fairness is guaranteed or that each honest party obtains a monetary penalty from the adversary. Protocols for this task are typically designed in an hybrid model where parties have access to a "claim-or-refund" transaction functionality denote $\mathcal{F}_{\mathrm{CR}}^{*}$.

In this work, we obtain improvements on the efficiency of these constructions by amortizing the cost over multiple executions of secure computation with penalties. More precisely, for computational security parameter λ, we design a protocol that implements $\ell = \mathsf{poly}(\lambda)$ instances of secure computation with penalties where the total number of calls to $\mathcal{F}_{\mathrm{CR}}^{*}$ is independent of ℓ.

Keywords: Secure computation, fairness, Bitcoin, amortization.

1. INTRODUCTION

Protocols for secure multiparty computation [23, 10] allow a set of mutually distrusting parties to carry out a distributed computation without compromising on privacy of inputs or correctness of the end result. Despite being a powerful tool, it is known that secure computation protocols do not provide fairness or guaranteed output delivery when a majority of the parties are dishonest [7]. Addressing this deficiency is critical if secure computation is to be widely adopted in practice, especially given the current interest in practical secure computation. Several workarounds have been proposed in the literature to counter adversaries that may decide to abort, possibly depending on the outcome of the protocol. In this work, we are interested in the workaround proposed in [18, 17] where an adversarial party that aborts on receiving output is forced to pay a mutually predefined monetary penalty to every other part that did not receive the output. In practice, such mechanisms would be effective if the compensation amount is rightly defined. While the original works [18, 17] depended on e-cash systems, recent works [4, 2, 6, 14, 1, 15, 12] have shown how to use a decentralized digital currency (like Bitcoin) to design protocols for secure computation in the penalty model.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978424

In this work, we propose major improvements to the efficiency of protocols for secure computation with penalties by amortizing the cost over multiple executions (effectively making the amortized "on-chain" cost zero). To better explain our contributions, we first discuss the model, efficiency metrics, settings, and the efficiency of state-of-the-art protocols.

Claim-or-refund transaction functionality. In [6, 15], protocols for secure computation with penalties are designed in a hybrid model where parties have access to an ideal transaction functionality called the *claim-or-refund transaction functionality* [6, 5, 19]. This functionality, denoted as $\mathcal{F}_{\mathrm{CR}}^{*}$, takes care of handling "money/coins" and allows protocols to be designed independently of the Bitcoin ecosystem. In a nutshell, $\mathcal{F}_{\mathrm{CR}}^{*}$ implements the following functionality: (1) it accepts a deposit of $\mathsf{coins}(q)$, a Boolean circuit ϕ and a time-limit τ from a designated sender S; and (2) waits until time τ to get a witness w from a designated receiver R such that $\phi(w) = 1$; and (3) if such a witness was received within time τ transfers $\mathsf{coins}(q)$ to R; (4) else returns $\mathsf{coins}(q)$ back to S.

Three features of $\mathcal{F}_{\mathrm{CR}}^{*}$ explain its importance: (1) $\mathcal{F}_{\mathrm{CR}}^{*}$ can be very efficiently implemented in Bitcoin (see Appendix A), and (2) $\mathcal{F}_{\mathrm{CR}}^{*}$ provides an abstraction which makes protocols designed in the $\mathcal{F}_{\mathrm{CR}}^{*}$-hybrid model robust to changes in the Bitcoin architecture, and (3) $\mathcal{F}_{\mathrm{CR}}^{*}$ is "complete" for secure computation with penalties. Protocols for secure computation with penalties designed in the $\mathcal{F}_{\mathrm{CR}}^{*}$-hybrid model work as long as $\mathcal{F}_{\mathrm{CR}}^{*}$ is implemented. Such an implementation need not be tied to Bitcoin, i.e., banks and other financial institutions could, in principle, support $\mathcal{F}_{\mathrm{CR}}^{*}$ transactions.

Capturing the cost of secure computation with penalties. A protocol for secure computation with penalties typically involves an *sequence* of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits. Thus the costs of such protocols can be captured in a variety of ways such as (1) the total number of calls to $\mathcal{F}_{\mathrm{CR}}^{*}$, (2) the maximum/total deposits made to $\mathcal{F}_{\mathrm{CR}}^{*}$ in the sequence of deposits, and the complexity of the parameters, specifically (3) the maximum/total size of Boolean circuits ϕ employed in the sequence and (4) the maximum value of time-limit τ used in the sequence. To realize $\mathcal{F}_{\mathrm{CR}}^{*}$ in Bitcoin, we need at least one Bitcoin transaction to be broadcasted by the sender (cf. Appendix A). Thus, the number of calls to $\mathcal{F}_{\mathrm{CR}}^{*}$ captures the number of Bitcoin transactions that need to be broadcasted to and supported by the Bitcoin network. The deposits made to $\mathcal{F}_{\mathrm{CR}}^{*}$ capture the amount of funds that need to be locked up in Bitcoin transactions during the course of the protocol. The size of the Boolean circuit ϕ used in an $\mathcal{F}_{\mathrm{CR}}^{*}$ transaction captures the amount of time miners need to spend to validate that $\mathcal{F}_{\mathrm{CR}}^{*}$ transaction, consequently captures the load on the network for verifying transactions. Additionally, Bitcoin transactions currently offer limited support for Bitcoin "scripts" (essen-

tially the circuit ϕ). While this is expected to improve in the future (and other alt-coins like Ethereum are already offer generous support for scripts), the size of the Boolean circuit does a good job in capturing the complexity of the scripts that Bitcoin needs to support secure computation with penalties. We denote the total size of the Boolean scripts (i.e., Bitcoin scripts) used in our protocols as the "script complexity" of the protocol. Finally, the maximum value of the time-limit used in the sequence of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits captures the "round complexity" of the protocol. Sometimes we make a distinction between "on-chain round complexity" and "off-chain round complexity." This distinction is expected to yield a tighter grip on the efficiency of the protocol. The "on-chain round complexity" refers to the number of sequential transactions that need to be made on the blockchain. Since the time taken to confirm a transaction on the blockchain today is about 1 hour, an "on-chain round complexity" of s implies that the protocol will take at least s hours to complete. The "off-chain round complexity" refers to the standard metric of round complexity used in traditional MPC protocols. Note that an off-chain round typically takes less than a few seconds to complete. Thus, we believe that for a fair comparison this distinction needs to be made.

Our contributions. We show how to amortize the cost of secure computation with penalties. Let λ be a computational security parameter. Then for $\ell = \mathrm{poly}(\lambda)$ we show how ℓ *sequential* instances of an n-party non-reactive (resp. reactive) secure computation penalties can be realized with the same "on-chain" cost of a *single execution* in [6] (resp. [15]). Since the on-chain latency is typically very high and the on-chain costs capture the load on the Bitcoin network, we believe that our results deliver major improvements to the efficiency of secure computation with penalties and make it more easy to envision practical implementations on the Bitcoin network (or other networks). Finally, in our protocols neither the parameter ℓ nor the sequence of possibly different functions that need to be evaluated need to be known in advance. (For the reactive case, an upper bound on the transcript and rounds need to be known in advance.)

Technical overview and differences from prior work. The main difference from prior work is that we *reuse* a single initial set of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits for multiple instances of secure computation with penalties. That is, parties make an initial set of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits first, then *locally* execute secure computation protocols, and whenever there is an abort in the local execution, they have recourse to the $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits in order to get penalties. That is, in an optimistic setting where all parties follow the protocol, the initial set of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits remain untouched throughout the ℓ local executions. In the general case, the initial set of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits will be claimed when an abort occurs in one of the local executions.

To make things simple, we divide the implementation of ℓ instances into three stages: (1) the master setup and deposit phase, (2.1) a local setup phase for each execution, (2.2) a local exchange phase for each execution, and (3) the master claim phase. In the master setup and deposit phase, parties run an unfair standard secure computation protocol that helps specify the Boolean circuits required for the initial $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits, following which parties make these $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits, referred to as the "master deposits." Note that parties do not yet know the inputs of any of the instances of secure computation with penalties and thus all they supply to the master setup phase is simply randomness. Consequently, the Boolean circuits in the $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits will also be independent of the inputs/outputs of the ℓ executions. This is one of the fundamental differences between the previous protocols [6, 15, 16] and ours. For example in the non-reactive protocols of [6, 16], the Boolean circuits in the $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits are commitments on the secret shares of the final output. That is, the function evaluation occurs first even before the $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits are made. On the other hand, in our case, there are multiple function evaluations and the master deposits are made before any of the function evaluations begin. Further, the master deposits will need to allow honest parties to obtain penalties in case *any* of the function evaluation instances are aborted by the adversary. Our approach can be applied to the setting of [6, 15, 16] by setting $\ell = 1$. By performing the master deposits before the function evaluation, our approach surprisingly makes the security analysis easier. In particular, we no longer need to worry about aborts that happen during the deposit phase. Even better, all the master deposits can be made simultaneously, i.e., in $O(1)$ on-chain rounds, unlike prior protocols where the deposit phase required $O(n)$ on-chain rounds. Also, in an optimistic setting where all parties behave honestly, the master claim phase (described later) can also be made simultaneously, i.e., in $O(1)$ on-chain rounds.

Once all the master deposits have been made, parties sequentially perform the local executions. At the beginning of each local execution, parties run a secure computation protocol specified in the *local setup phase*. The objective of this phase is to set things up in a way such that penalties can be obtained from the parties in case of aborts. Following this, parties enter the local exchange phase for that execution, where they exchange messages that reveal the output of that execution. Note that these phases do not involve $\mathcal{F}_{\mathrm{CR}}^{*}$. It is only when there is an abort in any of these phases, do parties enter the master claim phase where they try to claim these deposits. We describe the three phases in more detail.

Master setup and deposit phase. In this phase, parties run a secure computation protocol that implements the following functionality: (1) run the key generation algorithm of a signature scheme to generate (mvk, msk), (2) secret share msk among all parties, and output mvk to all parties. We refer to (mvk, msk) as the master keys. Note that msk is unknown to the adversary. Following the secure computation protocol, parties make a series of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits. These are the master deposits. The Boolean circuits used in these deposits perform two checks: first, they check for one or more messages each of which contain a signature that verifies against the master verification key mvk, and second, they check that the messages obeys a certain structural relation between them. The structural relation is necessary to ensure that the honest parties obtain penalties if a local execution was aborted. More on this later. Curiously, the sequence of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits in the master deposit phase is identical to the "see-saw" deposits of [15] in both the non-reactive and reactive cases. Of course, we will be using different (more complicated) scripts in each $\mathcal{F}_{\mathrm{CR}}^{*}$ deposit.

Local execution phase. In this overview, we will focus only on handling the non-reactive case. In the k-th local setup phase (for $k \in [\ell]$), parties run a secure computation protocol that evaluates the function f_k on inputs provided by the parties, and then secret shares the output among the parties. The secret shares of the outputs are authenticated *twice*: once under the msk and once under a *local signing key* that is generated inside this MPC. Note that to authenticate the output secret shares under msk, the parties will need to provide the secret shares of msk to the MPC. Neither the msk nor the local signing key will be revealed to the parties. Also, the messages that are signed aren't simply output secret shares but will include the execution number k and the identity of the party. That is, if s_i is the i-th output secret share, then signatures under msk and the local signing key will be computed on the message (i, k, s_i). Furthermore, the setup phase will also produce signatures under msk on messages of the form (j, i, k) where $j, i \in [n]$.[1]

[1]To avoid clutter in our presentation, we assume that the messages

These are referred to as the "lock witnesses." Another caveat is that we require that the MPC of the local setup phase to deliver outputs in a particular order. This specific ordering, the use of lock witnesses, and the structure of the messages containing the secret shares all will be important ideas that will ensure that the honest parties get compensated in the event of aborts.

Following the k-th local setup phase, parties enter the k-th local exchange phase in which parties broadcast the output shares along with the authentication under the local signing key to all parties. Again, we will require a specific ordering in which the parties perform broadcasts. If all parties behave honestly, then parties will obtain the output of the k-th local execution, and will proceed to the next execution, and so on. If there was an abort in either the local setup phase or the local exchange phase, parties enter the master claim phase and do not engage in any further local executions. Note that signatures under msk are never revealed during the local executions; they will be revealed only during the claim of the master deposits in the master claim phase.

Master claim phase. In this phase, parties take turns to claim master deposits. The objective of this phase is to ensure that if a local execution was aborted mid-way, then either this local execution is continued to its completion in this phase, or else guarantee that the honest parties obtain penalties. An important attack to defend against is one where the adversary *replays* messages from an older execution. Such an attack would end up allowing the adversary to claim all the master deposits it is required to claim thereby the adversary does not pay penalties to honest parties. Furthermore, it ensures that the most recent local execution remains aborted and only the adversary learns the output of that execution. Such attacks are taken care of (1) by the use of signatures under the master signing keys that will be revealed only in the master claim phase, and (2) by imposing certain conditions on the structural relations between the messages used in the claim of a master deposit. Claiming a master deposit involves revealing a partial transcript containing, say the first j output secret share messages that are of the form $(i, k, *)$ for all $i \in [j]$ and for some specific value $k \in [\ell]$.[2] The messages of this form alone are not sufficient to claim the deposit; one has to produce the corresponding signatures under msk as well. By imposing such conditions, namely that j signatures on messages $(1, k, *), \ldots, (j, k, *)$ are required to claim a deposit, we can ensure that the current local execution is continued. Signatures under msk on messages of the form $(i, *, *)$ will be revealed by honest P_i for a unique value k (typically the most recent local execution). This in turn will ensure that the k-th local execution is continued in the master claim phase. Of course, if the adversary were to abort in the master claim phase as well, we will show that this would result in all honest parties obtaining the necessary penalty.

Important caveats. Note that the penalties can be obtained only at the end of the master claim phase. The time-limits on the master deposits will typically be high in order to let all the ℓ executions finish. Suppose the very first execution was aborted by the adversary. Then the funds of the honest parties will remain locked up until the time-limit on the master deposit expires. We note that for the single instance case, i.e., $\ell = 1$, more efficient protocols are presented in [16]. Unfortunately, we were not able to take advantage of the techniques in their work. Finally, our protocols can also improve the efficiency of protocols for *secure cash distribution with penalties* considered in [2, 15]. While our protocols may be used to implement the protocol part of the construction in [15], the cash distribution part will require fresh deposits *per execution*. Still, we believe that the best venues for our results will be in applications such as poker where repeated executions among the same set of parties are likely.

Related work. The works of [4, 5] construct 2-party lottery protocols using Bitcoin scripts which essentially implement $\mathcal{F}_{\mathrm{CR}}^{*}$. Other notable works which are not in the $\mathcal{F}_{\mathrm{CR}}^{*}$ model include the works of [2, 1, 13, 12, 14]. The works of [13, 12] use a more powerful transaction functionality which implements a blockchain to implement "smart contracts" and fair secure computation (under the penalties notion). We wish to emphasize that protocols constructed in the $\mathcal{F}_{\mathrm{CR}}^{*}$-hybrid model can be easily cast into protocols in any of the above models. Also, we make an explicit distinction between the off-chain costs and the on-chain costs which is not always captured in other works. For instance, in Ethereum, the entire smart contract (or the function) is put on the blockchain, and in a naïve construction, every miner is involved in the computation of the function as well as the state changes associated with executing the contract. These are exactly the type of burdens on the miners that we are trying to relieve via use of (possibly expensive) off-chain mechanisms (e.g., secure computation). The works of [21, 8] are concerned with the establishment of a "payment channel" to allows parties to do an unbounded number of money transfers without burdening the blockchain.

2. PRELIMINARIES

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

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

All parties are assumed to run in time polynomial in the security parameter λ. We prove security in the "secure computation with coins" (SCC) model proposed in [6]. Note that the main difference from standard definitions of secure computation [9] 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 $\mathcal{Z}$,*

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

of the form (i, k, s_i) and (j, i, k) are padded appropriately so that signatures on messages of one form cannot be trivially used to forge signatures on messages of the other form.

[2] We often use '*' as the wildcard character.

Notation: session identifier sid, an n-input, n'-output function f, a hard-coded ordering of outputs $\chi = (\chi_1, \ldots, \chi_{n'})$, parties $P_1, \ldots, P_n$, adversary $\mathcal{S}$ that corrupts parties $\{P_s\}_{s \in C}$, set $H = [n] \setminus C$.

INPUT PHASE:

- Wait to receive a message $(\mathsf{input}, sid, ssid, r, y_r)$ from P_r for all $r \in H$.
- Wait to receive a message $(\mathsf{input}, sid, ssid, s, \{y_s\}_{s \in C})$ from $\mathcal{S}$.

OUTPUT DELIVERY:

- Compute $((\chi_1, z_1), \ldots, (\chi_{n'}, z_{n'})) \leftarrow f(y_1, \ldots, y_n)$.
- For $j \in [n']$, sequentially do: send $(\mathsf{output}, sid, ssid, z_j)$ to P_{χ_j}. If $\chi_j \in C$, then:
  - If $\mathcal{S}$ sends $(\mathsf{abort}, sid, ssid)$, send $(\mathsf{output}, sid, ssid, \perp)$ to P_r for $r \in H$.

Figure 1: The ideal functionality $\mathcal{F}_f^{\mathsf{ord}}$ enforcing ordered delivery of output.

$\mathcal{F}_{\mathrm{CR}}^{\star}$ with session identifier sid, running with parties $P_1, \ldots, P_n$, a parameter 1^λ, and an ideal adversary $\mathcal{S}$ proceeds as follows:

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

Figure 2: The special ideal functionality $\mathcal{F}_{\mathrm{CR}}^{\star}$.

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 $\mathcal{F}_f^{\star}$ 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 $(\mathsf{SigKeyGen}, \mathsf{SigSign}, \mathsf{SigVerify})$.

2.1 Ideal Functionalities

Secure function evaluation with ordered output delivery. In our protocols, we ask parties to run secure computation protocols that deliver output in a certain order. (Note that standard secure computation protocols do not guarantee fairness in the presence of a dishonest majority.) Such protocols can be obtained easily by tweaking existing MPC protocols in the following way. First, the function is evaluated on the inputs to produce, say n outputs $z_1, \ldots, z_n$. Each z_i is then secret shared among the parties. Once the outputs are delivered to the parties, they then proceed to reconstruct the actual outputs in order. That is, in the first reconstruction phase, all parties broadcast their shares of z_1. At the end of this phase, P_1 obtains z_1. Then parties broadcast their shares of z_2 in the next phase and so on. Our protocols typically involve sending the outputs in reverse order. The actual order is slightly more complicated, but the idea above can be trivially generalized to accommodate our needs. For clarity, we present the generalized definition of the functionality in Figure 1. The values χ_j specify the index of the party that is supposed to receive the output in the j-th phase. That is, in phase j, party P_{χ_j} receives the output z_j. Note that $n' > n$ is possible. In our protocols we will need $n' = O(n^2)$ but simple round reduction techniques can be applied to implement the desired functionality in $O(n)$ rounds. Note that the protocol realizing $\mathcal{F}_f^{\mathsf{ord}}$ does not guarantee fairness. Also note that $\mathcal{F}_f^{\mathsf{ord}}$ can be realized in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model.

Claim-or-refund transaction functionality $\mathcal{F}_{\mathrm{CR}}^{*}$ [6, 5, 19]. At a high level, $\mathcal{F}_{\mathrm{CR}}^{\star}$ allows a sender P_s to *conditionally* send $\mathsf{coins}(x)$ to a receiver P_r. The condition is formalized as the revelation of a satisfying assignment (i.e., witness) for a sender-specified circuit $\phi_{s,r}(\,\cdot\,; z)$ (i.e., relation) that may depend on some public input z. Further, there is a "time" bound, formalized as a round number τ, within which P_r has to act in order to claim the coins. An important property that we wish to stress is that the satisfying witness is made *public* by $\mathcal{F}_{\mathrm{CR}}^{\star}$. Any cryptocurrency that supports time-dependent scripts can be used to realize $\mathcal{F}_{\mathrm{CR}}^{*}$. Earlier Bitcoin implementations of $\mathcal{F}_{\mathrm{CR}}^{*}$ were given in [6, 5, 19], and we provide a more secure implementation in Appendix A. In a Bitcoin realization of $\mathcal{F}_{\mathrm{CR}}^{*}$, sending a message with $\mathsf{coins}(x)$ corresponds to broadcasting a transaction to the Bitcoin network, and waiting according to some time parameter until there is enough confidence that the transaction will not be reversed. We denote an $\mathcal{F}_{\mathrm{CR}}^{\star}$ transaction where sender P_s asks receiver P_r for a witness for a predicate ϕ in exchange for $\mathsf{coins}(q)$ with deadline τ by:

$$P_s \xrightarrow[q,\tau]{\phi} P_r$$

Next, we define an important metric of protocols that involve a sequence of $\mathcal{F}_{\mathrm{CR}}^{*}$ deposits called the "script complexity." This metric captures the load on the Bitcoin network for verifying the $\mathcal{F}_{\mathrm{CR}}^{*}$ transactions.

DEFINITION 3 (SCRIPT COMPLEXITY [14]). *Let Π be a protocol among $P_1, \ldots, P_n$ in the $\mathcal{F}_{\mathrm{CR}}^{*}$-hybrid model. For circuit ϕ, let $|\phi|$ denote its circuit complexity. For a given execution of Π starting from a particular initialization Ω of parties' inputs and random tapes and distribution of* coins*, let $V_{\Pi,\Omega}$ denote the sum of all $|\phi|$ such that some honest party claimed an $\mathcal{F}_{\mathrm{CR}}^{*}$ transaction by producing a witness for ϕ during an execution of Π. Then the* script complexity *of Π, denoted V_Π, equals $\max_\Omega(V_{\Pi,\Omega})$.* ♢

Secure computation with penalties—multiple executions. We now present the functionality $\mathcal{F}_{\mathrm{MSFE}}^{*}$ 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 Figure 3 for a formal description. The main difference between the prior definitions in [6, 15] is that $\mathcal{F}_{\mathrm{MSFE}}^{*}$ directly realizes multiple invocations of secure computation with penalties. For simplicitly $\mathcal{F}_{\mathrm{MSFE}}^{*}$ deals only with the non-reactive case.

In the first phase referred to as the *deposit phase*, the functionality $\mathcal{F}_{\mathrm{MSFE}}^{*}$ accepts safety deposits $\mathsf{coins}(d)$ from each honest party

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, a time-limit τ, set $H = [n] \setminus C$.

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

- Wait to get message $(\mathsf{setup}, sid, ssid, i, \mathsf{coins}(d))$ from P_i for all $i \in H$. Then wait to get message $(\mathsf{setup}, sid, ssid, \mathsf{coins}(hq))$ from $\mathcal{S}$ where $h = |H|$.

EXECUTION PHASE: Set $\mathsf{flg} = 0$. For $k = 1, \ldots$, sequentially do:

- Wait to receive a message $(\mathsf{input}, sid, ssid\|k, i, x_i^{(k)}, f_k)$ from P_i for all $i \in H$. Send $(\mathsf{function}, sid, ssid\|k, f_k)$ to all parties.
- Wait until time τ to receive a message $(\mathsf{input}, sid, ssid\|k, \{x_s^{(k)}\}_{s \in C}, f_k)$ from $\mathcal{S}$. If no such message was received within time τ, then go to the claim phase.
- Compute $(z_1^{(k)}, \ldots, z_n^{(k)}) \leftarrow f_k(x_1^{(k)}, \ldots, x_n^{(k)})$.
- Send message $(\mathsf{output}, sid, ssid\|k, \{z_s^{(k)}\}_{s \in C})$ to $\mathcal{S}$.
- If $\mathcal{S}$ returns $(\mathsf{continue}, sid, ssid)$, then send $(\mathsf{output}, sid, ssid\|k, z_i^{(k)})$ to each P_i.
- Else if $\mathcal{S}$ returns $(\mathsf{abort}, sid, ssid)$, update $\mathsf{flg} = 1$, and go to the claim phase.

CLAIM PHASE: At time τ, do:

- If $\mathsf{flg} = 0$ or $\perp$, send $(\mathsf{return}, sid, ssid, \mathsf{coins}(d))$ to all P_r for $r \in H$. 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 $(\mathsf{extra}, sid, ssid, \{q_i\}_{i \in H}, \sum_{i \in H} \mathsf{coins}(q_i))$.

Figure 3: Special ideal functionality $\mathcal{F}^*_{\mathrm{MSFE}}$ for multiple sequential SFE with penalties.

and penalty deposit $\mathsf{coins}(hq)$ from the adversary. Note that the penalty deposit suffices to compensate each honest party in 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 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. Note that penalties are distributed only at time τ. 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 at time τ.

Note that we require $\mathcal{S}$ to deposit $\mathsf{coins}(hq)$ up front. This is different from the definition of secure computation with penalties in [6], where $\mathcal{S}$ may not submit $\mathsf{coins}(hq)$ and yet the computation might proceed. We believe that our definition is more natural. We are able to support this definition because in our protocol (unlike the case in [6]), the computation happens only after all the deposits are made. Next, we discuss the reactive case.

Reactive case. The definition for the secure computation with penalties in the reactive setting $\mathcal{F}^*_{\mathrm{MMPC}}$ is identical to $\mathcal{F}^*_{\mathrm{MSFE}}$ except that the function f_k is composed of sub-functions for the different stages, i.e., $f_k = (f_{k,1}, \ldots, f_{k,\rho})$, where ρ denotes the number of stages. Now, $\mathcal{S}$ can abort between different stages of f_k or within a single stage, say $f_{k,\rho'}$. In either case, the honest parties will be compensated via the penalty deposit $\mathsf{coins}(hq)$ submitted by $\mathcal{S}$ in the deposit phase. For lack of space, the formal definition is presented in the full version.

3. TWO PARTY NON-REACTIVE CASE

We describe the protocol for the 2-party non-reactive case in Figure 4. For clarity, we annotate each of the steps in (1) the master deposits as Tx_j, (2) the k-th local setup phase as $\mathsf{sp}_j^{(k)}$, (3) the k-th local exchange phase as $\mathsf{ex}_j^{(k)}$, (4) the master claims as clm_j. Sometimes we treat these annotations as Boolean variables which are set to 1 if the corresponding event occurred or else they are set to 0. As an example, we say "$\mathsf{sp}_1^{(k)} = 1$" iff $\mathcal{F}_{\widehat{f}_k}^{\mathrm{ord}}$ delivered output to P_1. We now explain the design of the protocol and describe each of the phases in more detail. In the presentation here we ignore some details on the time-limits.

In the *master setup phase*, parties interact with an unfair ideal functionality that runs the key generation algorithm for a digital signature scheme, and outputs the master verification key mvk to both parties, and secret shares the master signing key msk. In addition, the master function will authenticate the shares of the master signing key. Looking ahead we will need this authentication because each subsequent local execution will need to produce signatures verifiable by the master verification key. To do so, these subsequent local executions will reconstruct the master signing key from the authenticated secret shares held by both parties. Following this, parties enter the *master deposit phase* where they make $\mathcal{F}^*_{\mathrm{CR}}$ deposits as follows. In the following, $\tau_2 > \tau_1$.

$$P_1 \xrightarrow[q,\tau_2]{\phi_2} P_2 \qquad (\mathsf{Tx}_2)$$

$$P_2 \xrightarrow[q,\tau_1]{\phi_1} P_1 \qquad (\mathsf{Tx}_1)$$

Here, the predicates ϕ_1, ϕ_2 have the master verification key mvk hardcoded in them. The predicates essentially check if their input is a valid signature against the master verification key mvk. The messages that are signed under msk will be secret shares of the output of a function evaluation (more on this in the next paragraph), and we will append the player index and an execution number denoted id, and then sign the message consisting of player id, nonce, and secret share under the master signing key msk. As we will see below, the predicate ϕ_1 takes as input one message and a corresponding signature, while the predicate ϕ_2 takes as input two messages and corresponding signatures. In addition to checking the validity of the signatures, the predicates also verify an additional condition on the nonces contained in the underlying signed messages. Below, we explicitly specify the predicates ϕ_1 and ϕ_2:

$$\phi_1(\mathsf{id}_1, t_1, \sigma_1; mvk) = \mathsf{SigVerify}(mvk, (1, \mathsf{id}_1, t_1), \sigma_1)$$

$$\phi_2(\mathsf{id}_1, t_1, \sigma_1, \mathsf{id}_2, t_2, \sigma_2; mvk) = \begin{pmatrix} (\mathsf{id}_1 = \mathsf{id}_2) \\ \bigwedge \mathsf{SigVerify}(mvk, (1, \mathsf{id}_1, t_1), \sigma_1) \\ \bigwedge \mathsf{SigVerify}(mvk, (2, \mathsf{id}_2, t_2), \sigma_2) \end{pmatrix}$$

Next, we describe the *local setup phase*. In the k-th local setup phase, the parties submit their authenticated shares of the master signing key, and further also submit the inputs to an unfair ideal functionality $\mathcal{F}_{\widehat{f}_k}^{\mathsf{ord}}$ computing the function f_k that is to be computed in this phase. As described before, the k-th setup phase first reconstructs the master signing key from the authenticated shares submitted by the parties. Then it computes the function f_k on the inputs submitted by the parties to obtain the output $z^{(k)}$. (For simplicity, we assume that all parties obtain the same output.) Following this, the output $z^{(k)}$ is secret shared using an additive secret sharing scheme to produce shares $s_1^{(k)}, s_2^{(k)}$. Each of these shares is then authenticated *twice*: once using the reconstructed master sign-

MASTER SETUP PHASE: P_1 and P_2 interact with an ideal functionality $\mathcal{F}_{\widehat{f}}$ that computes $(mvk, msk) \leftarrow \mathsf{SigKeyGen}(1^\lambda)$ and computes secret shares msk_1, msk_2 of msk and delivers $msk_1, mvk, \mathsf{MAC}(msk_2)$ to P_1 and $msk_2, mvk, \mathsf{MAC}(msk_1)$ to P_2 where MAC is (an information-theoretic) message authentication code.

MASTER DEPOSIT PHASE: Parties make the following $\mathcal{F}^*_{\mathrm{CR}}$ deposits:

$$P_1 \xrightarrow[q,\tau_2]{\phi_2} P_2 \qquad (\mathsf{Tx}_2)$$

$$P_2 \xrightarrow[q,\tau_1]{\phi_1} P_1 \qquad (\mathsf{Tx}_1)$$

where:

$$\phi_1(\mathsf{id}_1, t_1, \sigma_1; mvk) = \mathsf{SigVerify}(mvk, (1, \mathsf{id}_1, t_1), \sigma_1)$$
$$\phi_2(\mathsf{id}_1, t_1, \sigma_1, \mathsf{id}_2, t_2, \sigma_2; mvk) = \left(\begin{array}{l} (\mathsf{id}_1 = \mathsf{id}_2) \\ \quad \bigwedge \mathsf{SigVerify}(mvk, (1, \mathsf{id}_1, t_1), \sigma_1) \\ \quad\quad \bigwedge \mathsf{SigVerify}(mvk, (2, \mathsf{id}_2, t_2), \sigma_2) \end{array} \right)$$

EXECUTION PHASE: In the k-th local setup phase: Parties agree on the function to be executed f_k via broadcast. If there is disagreement, then parties enter the master claim phase. Else, P_1 and P_2 interact with an ideal functionality $\mathcal{F}^{\mathrm{ord}}_{\widehat{f}_k}$ to which they input: (1) the function f_k and inputs to f_k, and (2) mvk, secret shares of msk, and the corresponding MACs. $\mathcal{F}^{\mathrm{ord}}_{\widehat{f}_k}$ computes the output $z^{(k)}$ obtained by evaluating f_k on the inputs provided by the parties, then it secret shares $z^{(k)} = s_1^{(k)} \oplus s_2^{(k)}$. It then computes $(vk^{(k)}_{\mathsf{loc}}, sk^{(k)}_{\mathsf{loc}}) \leftarrow \mathsf{SigKeyGen}(1^\lambda)$ and computes $\sigma_i^{(k)} = \mathsf{SigSign}(msk, (i, k, s_i^{(k)}))$ and $\psi_i^{(k)} = \mathsf{SigSign}(sk^{(k)}_{\mathsf{loc}}, (i, k, s_i^{(k)}))$. $\mathcal{F}^{\mathrm{ord}}_{\widehat{f}_k}$ sends the outputs in the following order (i.e., $\chi = (2, 1)$ for $\mathcal{F}^{\mathrm{ord}}_{\widehat{f}_k}$):

1. $(s_2^{(k)}, \sigma_2^{(k)}, \psi_2^{(k)}; vk^{(k)}_{\mathsf{loc}})$ to P_2, $(\mathsf{sp}_2^{(k)})$
2. $(s_1^{(k)}, \sigma_1^{(k)}, \psi_1^{(k)}; vk^{(k)}_{\mathsf{loc}})$ to P_1. $(\mathsf{sp}_1^{(k)})$

In the k-th local exchange phase:

1. P_1 sends $(s_1, \psi_1) = (s_1^{(k)}, \psi_1^{(k)})$ to P_2. $(\mathsf{ex}_1^{(k)})$
2. If $\mathsf{SigVerify}(vk^{(k)}_{\mathsf{loc}}, (1, k, s_1), \psi_1) = 1$: P_2 sends $(s_2^{(k)}, \psi_2^{(k)})$ to P_1. $(\mathsf{ex}_2^{(k)})$

CLAIM PHASE: Parties enter this phase when either all local executions are completed or in the event of aborts after/during the master deposit phase.

1. At time τ_1: let k denote the last completed local execution. If $\mathsf{sp}_1^{(k+1)} = 1$, then P_1 claims Tx_1 using witness $(k+1, s_1^{(k+1)}, \sigma_1^{(k+1)})$, else claim Tx_1 using witness $(k, s_1^{(k)}, \sigma_1^{(k)})$ if $k > 0$. (clm_1)
2. At time τ_2, if party P_1 claimed Tx_1 using witness $(\mathsf{id}_1, t_1, \sigma_1)$, then party P_2 claims Tx_2 at time τ_2 using witness $(\mathsf{id}_1, t_1, \sigma_1, \mathsf{id}_2 = \mathsf{id}_1, t_2 = s_2^{(\mathsf{id}_1)}, \sigma_2 = \sigma_2^{(\mathsf{id}_1)})$. If there exists k such that $\mathsf{sp}_2^{(k)} = 1$ but $\mathsf{ex}_2^{(k)} = 0$, then both parties output $t_1 \oplus t_2$ as the output of the k-th execution. (clm_2)

Figure 4: 2-party realization of $\mathcal{F}^*_{\mathrm{MSFE}}$

ing key msk, and once using a local signing key $sk^{(k)}_{\mathsf{loc}}$ generated inside the unfair ideal functionality. We stress that the local signing key $sk^{(k)}_{\mathsf{loc}}$ is never revealed to any party; recall that the global signing key msk is never revealed to any party either. Finally, the *local outputs* of the unfair ideal functionality in the k-th local setup phase are distributed in the following *order* to the two parties:

1. Party P_2 obtains its secret share of the output $s_2^{(k)}$ along with a signature $\sigma_2^{(k)}$ on $T_2^{(k)} = (2, k, s_2^{(k)})$ signed under msk and a signature $\psi_2^{(k)}$ on $T_2^{(k)}$ signed under $sk^{(k)}_{\mathsf{loc}}$ and the corresponding local verification key $vk^{(k)}_{\mathsf{loc}}$. $(\mathsf{sp}_2^{(k)})$
2. Party P_1 obtains its secret share of the output $s_1^{(k)}$ along with a signature $\sigma_1^{(k)}$ on $T_1^{(k)} = (1, k, s_1^{(k)})$ signed under msk and a signature $\psi_1^{(k)}$ on $T_1^{(k)}$ signed under $sk^{(k)}_{\mathsf{loc}}$, and the corresponding local verification key $vk^{(k)}_{\mathsf{loc}}$. $(\mathsf{sp}_1^{(k)})$

We will shortly discuss why the order of outputs as above is needed (i.e., why P_1 obtains the output of the local setup phase after P_2). Observe that to obtain the output of the local phase, parties simply have to exchange the shares $s_1^{(k)}$ and $s_2^{(k)}$, and the output of the local phase equals $s_1^{(k)} \oplus s_2^{(k)}$. The *local exchange phase* happens in the following *order*:

1. Party P_1 first sends $T_1^{(k)}$ and $\psi_1^{(k)}$ to P_2. $(\mathsf{ex}_1^{(k)})$
2. If a valid message was received, then P_2 sends $T_2^{(k)}$ and $\psi_2^{(k)}$ to P_1. $(\mathsf{ex}_2^{(k)})$

After this, the local phase completes, and the parties have obtained the outputs. Note that since signatures under $sk^{(k)}_{\mathsf{loc}}$ are unforgeable except with negligible probability (because each party only has an additive share of $sk^{(k)}_{\mathsf{loc}}$), it follows except with negligible probability that a valid $(T_i^{(k)}, \psi_i^{(k)})$ pair sent by party P_i has to be the one generated by the local setup phase, and hence results in parties generating the correct output. Following this, the parties can then proceed to the next local phase and so on. Suppose ℓ denote the total number of successfully completed local executions. Once all the ℓ local executions are completed, the parties proceed to *master claim phase* where the following happens in order:

1. At time τ_1: let k denote the last completed local execution. If $\mathsf{sp}_1^{(k+1)} = 1$, then P_1 claims Tx_1 using witness $(k+1, s_1^{(k+1)}, \sigma_1^{(k+1)})$, else claim Tx_1 using witness $(k, s_1^{(k)}, \sigma_1^{(k)})$ if $k > 0$. (clm_1)
2. At time τ_2, if party P_1 claimed Tx_1 using witness $(\mathsf{id}_1, t_1, \sigma_1)$, then party P_2 claims Tx_2 at time τ_2 using witness $(\mathsf{id}_1, t_1, \sigma_1, \mathsf{id}_2 = \mathsf{id}_1, t_2 = s_2^{(\mathsf{id}_1)}, \sigma_2 = \sigma_2^{(\mathsf{id}_1)})$. If there exists k such that $\mathsf{sp}_2^{(k)} = 1$ but $\mathsf{ex}_2^{(k)} = 0$, then both parties output $t_1 \oplus t_2$ as the output of the k-th execution. (clm_2)

The master claim phase is designed in a way that allows the honest party to force the completion of the most recent local execution that is incomplete. For instance, P_1 can force the completion of the $(k+1)$-th execution by claiming Tx_1 using witness $(k+1, s_1^{(k+1)}, \sigma_1^{(k+1)})$. This then forces P_2 to reveal the secret share $s_2^{(k+1)}$ without which it cannot claim Tx_2. This is because the only signature under msk on messages of the form $(2, k+1, *)$ that P_2 possesses is $T_2^{(k+1)} = (2, k+1, s_2^{(k+1)})$. Thus, we have that either P_2 claims Tx_2 or pays a penalty $\mathsf{coins}(q)$ to honest P_1.

On the other hand, if P_1 was dishonest or if all local executions were completed, then parties effectively replay some old execution. That is, P_1 will claim Tx_1 using witnesses obtained from the k-th local execution for which $\mathsf{ex}_2^{(k)} = 1$. Following this P_2 can claim Tx_2 using witness revealed by P_1 and witness obtained from the k-th local setup phase. We prove:

THEOREM 1. *Assume one-way functions exist. There exists a 2-party protocol that* SCC-realizes $\mathcal{F}^*_{\mathrm{MSFE}}$ *in the* $(\mathcal{F}_{\mathrm{OT}}, \mathcal{F}^*_{\mathrm{CR}})$*-hybrid model such that the number of calls to* $\mathcal{F}^*_{\mathrm{CR}}$*, its script complexity, and deposit amounts are independent of the number of executions.*

Proof sketch. Let P_j denote the party corrupted by the adversary $\mathcal{A}$. We describe the simulator $\mathcal{S}$ for the protocol of Figure 4. $\mathcal{S}$ begins by acting as the unfair ideal functionality in the master setup phase, and runs the key generation algorithm of a digital signature scheme to produce (mvk, msk). It then chooses a random msk_j to give to $\mathcal{A}$. If $\mathcal{A}$ aborts the master setup phase, then $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and terminates the simulation. Else, in the master deposit phase, $\mathcal{S}$ acts as $\mathcal{F}^*_{\mathrm{CR}}$. If $j = 2$, it waits to receive a deposit from $\mathcal{A}$. If the deposit was not received or the deposit is not of the specified format, then $\mathcal{S}$ aborts outputting whatever $\mathcal{A}$ outputs. Else, $\mathcal{S}$ obtains $\mathsf{coins}(q)$ from $\mathcal{A}$ which it forwards to $\mathcal{F}^*_{\mathrm{MSFE}}$ as the penalty deposit. On the other hand, if $j = 1$, then $\mathcal{S}$ acting as $\mathcal{F}^*_{\mathrm{CR}}$ informs $\mathcal{A}$ that (honest) P_2 made the deposit as instructed. Then it waits for $\mathcal{A}$ to make the deposit. Again if the deposit is not of the correct form or was not made, then $\mathcal{S}$ terminates the simulation outputting whatever $\mathcal{A}$ outputs. In this case, the simulation is indistinguishable from the real execution since honest P_2 would have got $\mathsf{coins}(q)$ refunded from Tx_2 with all but negligible probability (except in the case $\mathcal{A}$ manages to forge signatures under msk). Else, it obtains $\mathsf{coins}(q)$ from $\mathcal{A}$ which it forwards to $\mathcal{F}^*_{\mathrm{MSFE}}$ as the penalty deposit. This concludes the simulation of the master setup and deposit phases.

In the k-th local setup phase, $\mathcal{S}$ learns of the function to be evaluated f_k from $\mathcal{F}^*_{\mathrm{MSFE}}$ and acts as the unfair ideal functionality $\mathcal{F}^{\mathsf{ord}}_{\tilde{f}_k}$, and obtains the input for this execution from $\mathcal{A}$. Note that if $\mathcal{A}$ sends incorrect shares of msk, then $\mathcal{S}$ terminates the simulation, and the simulation will be indistinguishable from the real execution since the MAC checks won't pass in the real execution except with negligible probability. Then $\mathcal{S}$ runs the key generation algorithm of a digital signature scheme to generate $(vk^{(k)}_{\mathsf{loc}}, sk^{(k)}_{\mathsf{loc}})$, and computes the signature $\psi^{(k)}_j$ under $sk^{(k)}_{\mathsf{loc}}$ on message $T^{(k)}_j = (j, k, s)$ for random value s. It then sends $T^{(k)}_j, \psi^{(k)}_j, vk^{(k)}_{\mathsf{loc}}$ to $\mathcal{A}$. If $\mathcal{A}$ aborts in this step, then $\mathcal{S}$ rejects any further local executions and goes directly to the simulation of the master claim phase (described below). This still results in a valid simulation since $\mathcal{A}$ should not be able to forge a signature under $sk^{(k)}_{\mathsf{loc}}$ due to the unforgeability property of the digital signature scheme. Otherwise, $\mathcal{S}$ begins the simulation of the local exchange phase. If $j = 1$, then it waits to receive (T, ψ) from $\mathcal{A}$. If $(T, \psi) \neq (T^{(k)}_j, \psi^{(k)}_j)$, $\mathcal{S}$ terminates the simulation (since this is a forgery that should happen only with negligible probability). Otherwise, $\mathcal{S}$ contacts $\mathcal{F}^*_{\mathrm{MSFE}}$ with the extracted input (obtained while acting as $\mathcal{F}^{\mathsf{ord}}_{\tilde{f}_k}$) to obtain the output of the local execution $z^{(k)}$. $\mathcal{S}$ acting as P_2 sends the value $T^{(k)}_2 = (2, k, z^{(k)} \oplus s)$ and the signature $\psi^{(k)}_2$ on $T^{(k)}_2$ to $\mathcal{A}$. The case when $j = 2$ is also handled similarly. $\mathcal{S}$ first submits the extracted input to $\mathcal{F}^*_{\mathrm{MSFE}}$ to get the output of the k-th local execution $z^{(k)}$. Then $\mathcal{S}$ acting as honest P_1 sends the value $T^{(k)}_1 = (1, k, s \oplus z^{(k)})$ along with a signature $\psi^{(k)}_1$ on $T^{(k)}_1$ to $\mathcal{A}$. It is easy to see that the simulation is indistinguishable from the real execution.

Finally, we describe the simulation of the master claim phase. $\mathcal{S}$ enters this phase either because there were: (1) aborts in the local setup phase, (2) aborts in the local exchange phase, or (3) all executions were successfully completed. We analyze separately the case when P_1 is corrupt and the case when P_2 is corrupt. Suppose $j = 1$. $\mathcal{S}$ waits until time τ_1 to see if P_1 claims Tx_1. Suppose P_1 does not claim Tx_1, then $\mathcal{S}$ waits to get its penalty deposit back from $\mathcal{F}^*_{\mathrm{MSFE}}$ and sends it to P_1 as refund obtained from Tx_2. The simulation is indistinguishable from the real execution because P_2 always obtains the output first in the local execution; thus if P_1 had received the output of a local execution phase, then so did P_2. Therefore, $\mathcal{S}$ will be able to get its penalty deposit $\mathsf{coins}(q)$ back from $\mathcal{F}^*_{\mathrm{MSFE}}$. Now on the other hand, suppose P_1 did claim Tx_1 using some witness $(\mathsf{id}_1, t_1, \sigma_1)$, then $\mathcal{S}$ checks if $\sigma_1^{(\mathsf{id}_1)} = \sigma_1$ (i.e., if σ_1 was handed to $\mathcal{A}$ during the simulation). The check will pass with all but negligible probability since this corresponds to $\mathcal{A}$ forging a signature under msk. In the rest of the analysis we will assume that $\sigma_1 = \sigma_1^{(\mathsf{id}_1)}$. Now, $\mathcal{S}$ acting as $\mathcal{F}^*_{\mathrm{CR}}$ will need to produce $\mathsf{coins}(q)$ to $\mathcal{A}$ as the claim reward for claiming Tx_1. To do so, $\mathcal{S}$ will need to obtain its penalty amount from $\mathcal{F}^*_{\mathrm{MSFE}}$. As before, this step is possible since P_1 cannot learn the output of a local execution before the honest party (recall P_2 always obtains outputs first in the local exchange phase), and thus $\mathcal{S}$ will be able to get its penalty deposit back from $\mathcal{F}^*_{\mathrm{MSFE}}$ which it can send to P_1 as the money obtained by claiming Tx_1. Now all that $\mathcal{S}$ needs to do is to produce witnesses for claiming Tx_2 in order to justify that $\mathsf{coins}(q)$ from Tx_2 are not going to be refunded back to P_1. This is easy since the witness $(\mathsf{id}_1, t_1, \sigma_1, \mathsf{id}_2 = \mathsf{id}_1, t_2 = z^{(\mathsf{id}_1)} \oplus t_1, \sigma_2 = \sigma_2^{(\mathsf{id}_2)})$ satisfies ϕ_2. In other words, the secret shares and corresponding signatures from the id_1-th execution will allow honest P_2 to claim Tx_2. This concludes the simulation in the case when P_1 is corrupt. It is easy to see that the simulation is indistinguishable from the real execution.

Next, we consider the case when $j = 2$. Now $\mathcal{S}$ will need to act first (as honest P_1) in the master claim phase. Let k denote the number of completed local executions, i.e., $\mathsf{ex}_2^{(k)} = 1$. If $k = 0$, then $\mathcal{S}$ does not have to act in the master claim phase. It will simply get back its penalty deposit from $\mathcal{F}^*_{\mathrm{MSFE}}$ and return it to P_2 as refund of Tx_1. The simulation is indistinguishable from real since except with negligible probability P_2 will not be able to produce signatures under msk to claim Tx_2. In the rest of the simulation, we assume $k > 0$. At time τ_1, $\mathcal{S}$ will have to claim Tx_1. To do so, $\mathcal{S}$ first checks if there was an incomplete local execution, i.e., if $\mathsf{sp}_1^{(k+1)} = 1$. If there was, this means that the output of the $(k+1)$-th execution was not obtained by both parties (in fact, it is possible that only P_2 obtained the output and not P_1). $\mathcal{S}$ will claim Tx_1 using the witness $(k+1, s \oplus z^{(k+1)}, \sigma_1^{(k+1)})$ where s was the secret share given to P_2 as part of the output of the $(k+1)$-th local setup phase. Now, $\mathcal{S}$ waits to see if P_2 claims Tx_2. Suppose P_2 does not claim Tx_2, then this means that in the real execution honest P_1 would not obtain the output, but only the penalty. Thus, to make the simulation indistinguishable from real, $\mathcal{S}$ will send an abort message to $\mathcal{F}^*_{\mathrm{MSFE}}$, and terminate the simulation (in particular, it will not get its penalty deposit back from $\mathcal{F}^*_{\mathrm{MSFE}}$). On the other hand, if P_2 did claim Tx_2, then except with negligible probability it has to do using the witness $(k+1, s \oplus z^{(k+1)}, \sigma_1^{(k+1)}, k+1, s, \sigma_2^{(k+1)})$. This is because the only signature under msk on messages of the form $(k+1, *, *)$ that $\mathcal{A}$ possesses is on the message $(k+1, s, \sigma_2^{(k+1)})$ obtained during the interaction with $\mathcal{S}$. Thus, in this case, $\mathcal{S}$ asks $\mathcal{F}^*_{\mathrm{MSFE}}$ to

Note on time-limits: $\tau_1 < \tau_2 < \cdots < \tau_n$.
For each $i \in [n-1]$: $\tau^{\mathsf{lock}}_{i+1,i} = \cdots \tau^{\mathsf{lock}}_{n,i} < \tau_i = \tau^{\mathsf{unlock}}_{i+1,i} = \cdots \tau^{\mathsf{unlock}}_{n,i}$.

ROOF DEPOSITS. For each $j \in [n-1]$: $(\mathsf{Tx}_{n,j})$

$$P_j \xrightarrow[q,\tau_n]{\phi_n} P_n$$

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

- Rung unlock: For $j = n$ down to $i+1$: $(\mathsf{Tx}^{\mathsf{unlock}}_{j,i})$

$$P_j \xrightarrow[q,\tau^{\mathsf{unlock}}_{j,i}]{\phi^{\mathsf{unlock}}_{j,i}} P_i$$

- Rung climb: (Tx_i)

$$P_{i+1} \xrightarrow[i\cdot q,\tau_i]{\phi_i} P_i$$

- Rung lock: For each $j = n$ down to $i+1$: $(\mathsf{Tx}^{\mathsf{lock}}_{j,i})$

$$P_i \xrightarrow[q,\tau^{\mathsf{lock}}_{j,i}]{\phi^{\mathsf{lock}}_{j,i}} P_j$$

Figure 5: Locked ladder mechanism from [15].

deliver the output to P_1 for execution $k+1$, and obtains back the penalty deposit from $\mathcal{F}^*_{\mathrm{MSFE}}$ which it forwards to P_2 as the reward obtained for claiming Tx_2. Finally, we consider the case when $\mathsf{sp}^{(k+1)}_1 = 0$, i.e., the $(k+1)$-th execution did not deliver outputs to either party. In this case, $\mathcal{S}$ gets its penalty deposit $\mathsf{coins}(q)$ back from $\mathcal{F}^*_{\mathrm{MSFE}}$. Then $\mathcal{S}$ claims Tx_1 using witnesses from the k-th execution, i.e., $(k, s \oplus z^{(k)}, \sigma^{(k)}_1)$ where s was the random secret share sent to P_2. Now if P_2 claims Tx_2, except with negligible probability it has to do using witness $(k, s \oplus z^{(k)}, \sigma^{(k)}_1, k, s, \sigma^{(k)}_2)$. Suppose P_2 claimed Tx_2, then $\mathcal{S}$ produces the necessary $\mathsf{coins}(q)$ from the returned penalty deposit. On the other hand, if P_2 did not claim Tx_2, then $\mathcal{S}$ sends the returned $\mathsf{coins}(q)$ back to $\mathcal{F}^*_{\mathrm{MSFE}}$ to be delivered to the honest party as extra reward. It is easy to see that the simulation is indistinguishable from real both in the standard sense as well as with respect to the distribution of coins. This concludes the proof of the theorem. □

4. MULTIPARTY CASE

We describe the protocol for the multiparty non-reactive case.

MASTER SETUP PHASE AND MASTER DEPOSIT PHASE. In the *master setup phase*, parties interact with an unfair ideal functionality that realizes the master setup function which runs the key generation algorithm for a digital signature scheme, and outputs the master verification key mvk to all n parties, and secret shares the master signing key msk. In addition, the master function will authenticate the shares of the master signing key. That is, in spirit, the master setup phase is identical to the one in the 2-party case, except now it caters to n parties. Next, parties enter the *master deposit phase* where they make $\mathcal{F}^*_{\mathrm{CR}}$ deposits as in Figure 5 (i.e., identical to the locked ladder mechanism in [15]). Note that relation between time-limits is specified in Figure 5.

Here, the predicates $\{\phi_i\}$, $\{\phi^{\mathsf{unlock}}_{j,i}\}$, $\{\phi^{\mathsf{lock}}_{j,i}\}$ all have the master verification key mvk hardcoded in them. The predicates essentially check if their input is a valid signature against the master verification key mvk. The messages that are signed under msk will be secret shares of the output of a function evaluation (more on this in the next paragraph), and we will append the player index and a nonce denoted id which essentially denotes an execution number, and then sign the message consisting of player id, nonce, and secret share under the master signing key msk. In addition to checking the validity of the signatures, the predicates also verify an additional structural relation on the nonces contained in the underlying signed messages. Below, we explicitly specify the predicates $\{\phi^{\mathsf{lock}}_{j,i}\}, \{\phi_i\}, \{\phi^{\mathsf{unlock}}_{j,i}\}$:

$$\phi^{\mathsf{lock}}_{j,i}(\mathrm{TT}, \mathsf{id}, \sigma; mvk) = \mathsf{tv}^{(\mathsf{id})}_{i-1}(\mathrm{TT}) \bigwedge \mathsf{SigVerify}(mvk, (j, i, \mathsf{id}), \sigma)$$

$$\phi_i(\mathrm{TT}, \mathsf{id}; mvk) = \mathsf{tv}^{(\mathsf{id})}_i(\mathrm{TT})$$

$$\phi^{\mathsf{unlock}}_{j,i}(\mathrm{TT}, \mathsf{id}, \sigma; mvk) = \mathsf{tv}^{(\mathsf{id})}_i(\mathrm{TT}) \bigwedge \mathsf{SigVerify}(mvk, (j, i, \mathsf{id}), \sigma)$$

In the above, we refer to the witness σ as the "lock witness." The description of the predicates use the *transcript validator* tv which we define below:

Let $\mathrm{TT} = (T^{(\mathsf{id}_1)}_1, \sigma^{(\mathsf{id}_1)}_1) \| \cdots \| (T^{(\mathsf{id}_i)}_i, \sigma^{(\mathsf{id}_i)}_i)$. Then $\mathsf{tv}^{(\mathsf{id})}_i(\mathrm{TT}) = 1$ iff

- $\mathsf{id}_1 = \cdots = \mathsf{id}_i \geq \mathsf{id}$.
- for all $j \leq i$: $T^{(\mathsf{id}_j)}_j$ is a message of the form $(j, \mathsf{id}_j, *)$ and $\sigma^{(\mathsf{id}_j)}_j$ is a valid signature on $T^{(\mathsf{id}_j)}_j$ under msk.

That is, $\mathsf{tv}^{(\mathsf{id})}_i$ ensures that the witnesses reveal the partial transcript containing the first i message-signature pairs, i.e., (T, σ) pairs. Furthermore the relation between the transcript witness and the lock witness is that the id's contained in them are such that $\mathsf{id}_1 = \cdots = \mathsf{id}_i \geq \mathsf{id}$.

LOCAL SETUP PHASE. Next, we describe the *local setup phase*. In the k-th local setup phase, the parties submit their authenticated shares of the master signing key, and further also submit the inputs to an unfair ideal functionality $\mathcal{F}^{\mathsf{ord}}_{\widehat{f}_k}$ computing the function f_k. The k-th local setup phase first reconstructs the master signing key from the authenticated shares submitted by the parties. Then it computes the function f_k on the inputs submitted by the parties to obtain the output $z^{(k)}$. Following this, the output $z^{(k)}$ is secret shared using an additive secret sharing scheme to produce shares $s^{(k)}_1, \ldots, s^{(k)}_n$. Each of these shares is then authenticated *twice*: once using the reconstructed master signing key msk, and once using a fresh local signing key $sk^{(k)}_{\mathsf{loc}}$ generated inside $\mathcal{F}_{\widehat{f}_k}$. In addition signatures under msk are generated on messages (j, i, k) for all $i, j \in [n] \times [n]$ such that $j \geq i$. We stress that the local signing key $sk^{(k)}_{\mathsf{loc}}$ is never revealed to any party; recall that the global signing key msk is never revealed to any party either. Finally, the *local outputs* of the unfair ideal functionality in the k-th local setup phase are distributed in the following *order* to the parties:

For $i = n$ down to 1:

(a) [**Main witness signed under both local and global keys**] Party P_i obtains its secret share of the output $s^{(k)}_i$ along with a signature $\sigma^{(k)}_i$ on the message $T^{(k)}_i = (i, k, s^{(k)}_i)$ signed under msk and a signature $\psi^{(k)}_i$ on $T^{(k)}_i$ signed under $sk^{(k)}_{\mathsf{loc}}$ and the corresponding local verification key $vk^{(k)}_{\mathsf{loc}}$. $(\mathsf{sp}^{(k)}_i)$

(b) [**Lock witness signed under global key**] For $j = n$ to $i+1$:

P_j obtains a signature $\sigma_{j,i}^{(k)}$ on "(j,i,k)" under msk. ($\mathsf{sp}_{j,i}^{(k)}$)

Observe that the witnesses delivered by $\mathcal{F}_{\hat{f}_k}^{\mathsf{ord}}$ are in the reverse order of the witnesses that are required to claim the master deposits. Later in the local exchange phase, these witnesses will be exchanged among the parties in the reverse order in which they were distributed in the setup phase.

LOCAL EXCHANGE PHASE. To obtain the output of the local phase, parties simply have to exchange the shares $\{s_i^{(k)}\}$, and the output of the local phase equals $\bigoplus_i s_i^{(k)} = z^{(k)}$. The *local exchange phase* happens in the following *order*:

For $i = 1$ to n:

(a) Party P_i broadcasts $T_i^{(k)}$ and $\psi_i^{(k)}$ to all parties. ($\mathsf{ex}_i^{(k)}$)

(b) [**Abort**] Let $\mu_i^{(k)} = (T_i^{(k)}, \psi_i^{(k)})$ denote the message sent by P_i. If either $T_i^{(k)}$ is not of the form $(i,k,*)$ or if $\mathsf{SigVerify}(vk_{\mathsf{loc}}^{(k)}, T_i^{(k)}, \psi_i^{(k)}) \neq 1$, then all parties terminate the k-th local phase, do not participate in any further local executions, and enter the master claim phase.

After this, the local phase completes, and the parties have obtained the outputs. Note that since signatures under $sk_{\mathsf{loc}}^{(k)}$ are unforgeable except with negligible probability (because each party only has an additive share of $sk_{\mathsf{loc}}^{(k)}$), it follows except with negligible probability that a valid $(T_i^{(k)}, \psi_i^{(k)})$ pair sent by party P_i has to be the one generated by the local setup phase, and hence results in parties generating the correct output. Following this, the parties can then proceed to the next local phase and so on. Alternatively, if some party did not send a valid message, then the honest parties would simply terminate the local executions and enter the master claim phase. An important note is that it may be the case that at this point the adversary already knows the output, therefore in these cases, we have to ensure that the honest party is compensated. This will be handled in the master claim phase.

MASTER CLAIM PHASE. From the discussion above, it is clear that parties may enter the master claim phase if there was an abort that happened during one of the earlier phases. We will handle all these cases in our description of the *master claim phase*. Let k denote the most recent completed execution, i.e., $\mathsf{ex}_n^{(k)} = 1$. It is possible that the $(k+1)$-th execution was never started (either there was an abort or the parties unanimously agreed to terminate all local executions), or there was an abort in the middle which means $\mathsf{sp}_i^{(k+1)} = 1$ or even $\mathsf{ex}_i^{(k+1)} = 1$ for some i. Note however that $\mathsf{ex}_n^{(k+1)} = 0$ must hold (otherwise $(k+1)$-th execution was also completed). We describe the master claim phase for each P_i.

1. For $j = 1$ to $i-1$: At time $\tau_{i,j}^{\mathsf{lock}}$ if $j = 1$ or $\mathsf{clm}_{j-1} = 1$ or $\mathsf{clm}_{i',j-1}^{\mathsf{unlock}} = 1$ for some i': ($\mathsf{clm}_{i,j}^{\mathsf{lock}}$)

(a) If $j = 1$ and $i \neq 1$, then claim $\mathsf{Tx}_{i,1}^{\mathsf{lock}}$ using witness $(\mathrm{TT}_0 = \mathsf{NULL}, k', \sigma')$, where $(k', \sigma') = (k+1, \sigma_{i,1}^{(k+1)})$ if $\mathsf{sp}_1^{(k+1)} = 1$, else $(k', \sigma') = (k, \sigma_{i,1}^{(k)})$.

(b) Else, let $\mathbf{TT}$ be the set of transcripts that were revealed during the claim of $\mathsf{Tx}_{j-1}, \{\mathsf{Tx}_{i',j-1}^{\mathsf{unlock}}\}$. Let $\mathbf{ID}$ denote the set of id's that the transcripts in $\mathbf{TT}$ are consistent with, i.e., for $\mathrm{TT} \in \mathbf{TT}$, there is an $\mathsf{id} \in \mathbf{ID}$, such that $\mathsf{tv}_{j-1}^{(\mathsf{id})}(\mathrm{TT}) = 1$. Let id' denote the maximum value in $\mathbf{ID}$ and let TT' denote the corresponding transcript. Claim $\mathsf{Tx}_{i,j}^{\mathsf{lock}}$ using witness $\mathrm{TT}', \mathsf{id}', \sigma_{i,j}^{(\mathsf{id}')}$.

2. At time τ_i if (1) $i = 1$ or (2) $\mathsf{clm}_{i-1} = 1$ or (3) $\mathsf{clm}_{j,i-1}^{\mathsf{unlock}} = 1$ for some j or (4) $\mathsf{clm}_{j,i}^{\mathsf{lock}} = 1$ for some j: (clm_i)

(a) If $i = 1$, then claim Tx_1 using $T_1^{(k')}, \sigma_1^{(k')}$ where k' is the maximum value such that $\mathsf{sp}_1^{(k')} = 1$.

(b) Else, let $\mathbf{TT}$ be the set of transcripts that were revealed during the claim of $\mathsf{Tx}_{i-1}, \{\mathsf{Tx}_{j,i-1}^{\mathsf{unlock}}\}, \{\mathsf{Tx}_{j,i}^{\mathsf{lock}}\}$. (Note that all these transcripts contain the first $i-1$ secret shares.) Let $\mathbf{ID}$ denote the set of id's that the transcripts in $\mathbf{TT}$ are consistent with, i.e., for $\mathrm{TT} \in \mathbf{TT}$, there is an $\mathsf{id} \in \mathbf{ID}$, such that $\mathsf{tv}_{i-1}^{(\mathsf{id})}(\mathrm{TT}) = 1$. Let id' denote the maximum value in $\mathbf{ID}$ and let TT' denote the corresponding transcript. Claim Tx_i using witness $\mathrm{TT}_i = (\mathrm{TT}' \| (T_i^{(\mathsf{id}')} \| \sigma_i^{(\mathsf{id}')}), \mathsf{id}')$. Save the value TT_i to use in the next step.

3. For $j = i+1$ to n: At time $\tau_{j,i}^{\mathsf{unlock}}$ if $\mathsf{clm}_{j,i}^{\mathsf{lock}} = 1$: ($\mathsf{clm}_{j,i}^{\mathsf{unlock}}$)

(a) Claim $\mathsf{Tx}_{j,i}^{\mathsf{unlock}}$ using TT_i (from the previous step), $k', \sigma_{j,i}$ where $(*, k', \sigma_{j,i})$ was the witness used to claim $\mathsf{Tx}_{j,i}^{\mathsf{lock}}$.

This concludes the description of the master claim phase. We present a series of propositions which will be useful to prove that our protocol realizes $\mathcal{F}_{\mathrm{MSFE}}^{*}$. Detailed proofs of the propositions are available in the full version. In the following, we assume that k denotes the most recent completed execution, i.e., $\mathsf{ex}_n^{(k)} = 1$.

PROPOSITION 2. *Honest parties never lose money during the claim phase. That is, for every honest P_i:*

1. *If Tx_{i-1} was claimed, then P_i will be able to claim Tx_i.*
2. *If $\mathsf{Tx}_{j,i}^{\mathsf{lock}}$ was claimed by P_j for $j > i$, then P_i will be able to claim $\mathsf{Tx}_{j,i}^{\mathsf{unlock}}$.*
3. *If $\mathsf{Tx}_{i,j}^{\mathsf{unlock}}$ was claimed by P_j for $j > i$,then it must hold that $\mathsf{Tx}_{i,j}^{\mathsf{lock}}$ was claimed by P_i.*

Proof sketch. The main argument is that the local setup phase releases witnesses in a way such that if Tx_{i-1} can be claimed then so can Tx_i. This is because Step $\mathsf{sp}_{i-1}^{(k)}$ occurs after Step $\mathsf{sp}_i^{(k)}$. Next, note that a claim of $\mathsf{Tx}_{j,i}^{\mathsf{lock}}$ will require witnesses for claiming Tx_{i-1} and the lock witness $\sigma_{j,i}$. Now to claim $\mathsf{Tx}_{j,i}^{\mathsf{unlock}}$, P_i needs witnesses for claiming Tx_i and the lock witness $\sigma_{j,i}$. Therefore, if $\mathsf{Tx}_{j,i}^{\mathsf{lock}}$ is claimed then P_i can claim $\mathsf{Tx}_{j,i}^{\mathsf{unlock}}$ using witnesses for claiming Tx_i (this follows from the argument for the previous case) and the lock witness $\sigma_{j,i}$ revealed during the claim of $\mathsf{Tx}_{j,i}^{\mathsf{lock}}$. This completes the proof.

PROPOSITION 3. *There exists a unique $k' \leq k+1$ such that for each $i \in H$ (i.e., P_i is honest), the only signatures under msk on messages of the form $(i, k'', *)$ that are revealed to adversary are for $k' = k''$.*

Proof sketch. The main argument is that the lexicographically first honest party, say P_i will reveal only one signature under msk on messages of the form $(i, *, *)$. That is there will be a unique k' for which P_i will release only one signature under msk on a message of the form $(i, k', *)$. Denote this message-signature pair as (T_i, σ_i). Actually, $(T_i, \sigma_i) = (T_i^{(k)}, \sigma_i^{(k)})$ where the latter is received in Step $\mathsf{sp}_i^{(k)}$. Clearly, (T_i, σ_i) is released when P_i claims Tx_i. Let TT_i be the witness used to claim Tx_i. The master claim is designed in a way such that TT_i along with a lock witness is used to claim $\mathsf{Tx}_{j,i}^{\mathsf{unlock}}$ (see Steps (2b) and (3a)). Now given that

honest P_i releases signatures under msk only on T_i, it follows that any valid partial transcript $\mathtt{TT}_{i'}$ released by honest party $P_{i'}$ to claim $\mathsf{Tx}_{i'}$ or $\mathsf{Tx}^{\mathsf{unlock}}_{j,i'}$ or $\mathsf{Tx}^{\mathsf{lock}}_{i',j}$ for $j > i$ will necessarily have $(T_i, \sigma_i) \in \mathtt{TT}_{i'}$. Thus, for $\mathtt{TT}_{i'}$ to be a valid partial transcript, it must hold that $(T_{i'}, \sigma_{i'}) \in \mathtt{TT}_{i'}$ must be such that $T_{i'} = (i', k', *)$. This completes the proof.

PROPOSITION 4. *Suppose the local setup phase of the $(k+1)$-th execution is successfully completed. Then, either the $(k+1)$-th execution was completed in the master claim phase, or all honest parties obtained a penalty. More precisely, for every $j \in [n]$, the following holds right after the j-th unlock phase: either $\mathtt{TT}_j$ containing the transcript of the most recent execution up to the j-th secret share was revealed by P_j, or all honest parties have obtained a penalty already.*

Proof sketch. Suppose the $(k+1)$-th local setup phase was completed. This means that all honest parties obtained the local witnesses for the $(k+1)$-th execution. Thus, each honest party P_i will begin claiming $\mathsf{Tx}^{\mathsf{lock}}_{i,1}$ using the lock witness $\sigma^{(k+1)}_{i,1}$ on message $(i, 1, k+1)$. To claim $\mathsf{Tx}^{\mathsf{unlock}}_{i,1}$, party P_1 must produce $\mathtt{TT}_1$ which contains main witness $(1, k+1, *)$, i.e., corresponding to the $(k+1)$-th execution. It no such main witness is released, then it follows that honest parties claim penalty $\mathsf{coins}(q)$ from P_1. (In this case, by Proposition 2 we have that honest parties don't lose money elsewhere, so they end up with penalty $\mathsf{coins}(q)$.) The remainder of the proof follows by an induction argument on each P_j (with the base proved above for $j = 1$) for the statement exactly as in the proposition. We note that for the general case, honest parties whose index is less than j would have claimed penalty $\mathsf{coins}(q)$ from the $\mathsf{coins}((j-1)q)$ deposited in Tx_{j-1}, and the honest parties whose index is above j would have claimed penalty $\mathsf{coins}(q)$ from the lock deposits $\mathsf{Tx}^{\mathsf{lock}}_{i,j}$. Note that the proposition implies that if the adversary gets the output of the $(k+1)$-th execution, then either honest parties also obtain the output or they obtain a penalty.

PROPOSITION 5. *Suppose there was an abort in the $(k+1)$-th local setup phase. Then, the adversary obtains the output of the $(k+1)$-th execution only if (1) the honest parties also obtained the output of $(k+1)$-th execution, or (2) all honest parties obtained a penalty.*

Proof sketch. Suppose $\mathsf{sp}^{(k+1)}_1 = 0$. Then we argue that no party gets the output of the $(k+1)$-th execution. This is because no party obtains the first secret share of the output. Then by Proposition 2 we have that honest parties don't lose money, and this suffices for security. On the other hand if $\mathsf{sp}^{(k+1)}_1 = 1$, then it is possible that the adversary obtains the output of the $(k+1)$-th execution. Note that since an abort happened in the $(k+1)$-th local setup phase, it follows that the honest parties would not have broadcasted any messages in the $(k+1)$-th local exchange phase. Thus, for the adversary to get output, it needs honest parties to reveal the main witnesses corresponding to the $(k+1)$-th execution during the master claim phase. By Proposition 3, it follows that the adversary must ensure that the first honest party, say P_i reveals the main witness corresponding to the $(k+1)$-th execution (even though the adversary might have obtained this witness from the $(k+1)$-th local exchange phase). Then, by an argument similar to the proof of Proposition 4 and starting the base case of the induction from index $i+1$ we have that either the $(k+1)$-th local execution was completed or all honest parties obtained a penalty. On the other hand, if the first honest party does not produce a main witness corresponding to the $(k+1)$-th execution, then the adversary will not obtain the output of the $(k+1)$-th execution. In this case, invoking Proposition 2 is sufficient to ensure security. We defer the proof of the following theorem to the full version.

THEOREM 6. *Assume one-way functions exist. There exists a protocol that* SCC-realizes $\mathcal{F}^*_{\mathrm{MSFE}}$ *in the $(\mathcal{F}_{\mathrm{OT}}, \mathcal{F}^*_{\mathrm{CR}})$-hybrid model s.t. the number of calls to $\mathcal{F}^*_{\mathrm{CR}}$and its script complexity are independent of the number of executions.*

4.1 The Reactive Case

Due to space limitations, we only provide a short sketch of our protocol. More details are avaiable in the full version. We will follow the idea used in [15] to realize secure computation of reactive functionalities with penalties. At a high level, the idea is to let the parties run an MPC protocol π' for the underlying reactive functionality, and have the predicates in the $\mathcal{F}^*_{\mathrm{CR}}$ transactions check the validity of the partial protocol transcript. That is, let an n-party m-message protocol π' be defined by pairs of algorithms $\{\mathsf{nmf}'_j, \mathsf{tv}'_j\}_{j\in[n]}$. Here tv'_j is the transcript validator function that takes a transcript of the protocol up to the j-th message, and outputs 1 iff it is a valid transcript of π'. The algorithm nmf'_j is the next message function that takes a valid partial transcript $\mathtt{TT}'_{j-1}$ (i.e., $\mathsf{tv}'_{j-1}(\mathtt{TT}'_{j-1}) = 1$), party $P_{j \bmod n}$'s input $x_{j \bmod n}$ and its private randomness, say $\omega_{j \bmod n}$, and produces the j-th message μ'_j of the protocol signed under $P_{j \bmod n}$'s public key. We define the j-th partial transcript $\mathtt{TT}'_j = \mathtt{TT}'_{j-1} \| \mu'_j$.

Protocol transformation. As in [15], we will transform a n-party m-message protocol $\pi' = \{\mathsf{nmf}'_j, \mathsf{tv}'_j\}_{j\in[m]}$ into an equivalent n-message m-message protocol $\widetilde{\pi} = \{\widetilde{\mathsf{nmf}}_j, \widetilde{\mathsf{tv}}_j\}_{j\in[m]}$ where the protocol messages contain layers of signatures. That is, each party signs each message it sends, so messages contain layers of signatures. Also, parties do not accept messages which do not have correct signatures.

Our construction. Surprisingly, our construction for the reactive case is very similar to the protocol for the non-reactive case. In fact, the master setup phase and the master deposit phase is identical. That is, the sequence of deposits is the same as the locked ladder mechanism presented in Figure 5. As it turns out, the predicate descriptions are also identical as in the non-reactive case, of course with the important difference that now tv will also need to include the *transcript validator* of the MPC protocol realizing the reactive functionality. Thus, our predicates are:

$$\phi^{\mathsf{lock}}_{j,i}(\mathtt{TT}, \mathsf{id}, \sigma; mvk) = \mathsf{tv}^{(\mathsf{id})}_{i-1}(\mathtt{TT}) \bigwedge \mathsf{SigVerify}(mvk, (j, i, \mathsf{id}), \sigma)$$

$$\phi_i(\mathtt{TT}, \mathsf{id}; mvk) = \mathsf{tv}^{(\mathsf{id})}_i(\mathtt{TT})$$

$$\phi^{\mathsf{unlock}}_{j,i}(\mathtt{TT}, \mathsf{id}, \sigma; mvk) = \mathsf{tv}^{(\mathsf{id})}_i(\mathtt{TT}) \bigwedge \mathsf{SigVerify}(mvk, (j, i, \mathsf{id}), \sigma)$$

The *transcript validator* tv defined below, now depends on $\widetilde{\pi} = \{\widetilde{\mathsf{nmf}}_j, \widetilde{\mathsf{tv}}_j\}_{j\in[m]}$:

> Let $\mathtt{TT} = (\mu'_1, \mathsf{id}_1, \psi_1, T_1, \sigma_1) \| \cdots \| (\mu'_i, \mathsf{id}_i, \psi_i, T_i, \sigma_i)$. Then $\mathsf{tv}^{(\mathsf{id})}_i(\mathtt{TT}) = 1$ iff
>
> - $\mathsf{id}_1 = \cdots = \mathsf{id}_i \geq \mathsf{id}$.
> - for all $j \leq i$: T_j is a message of the form $(j, \mathsf{id}_j, *)$ and σ_j is a valid signature on T_j under msk
> - $\widetilde{\mathsf{tv}}^{(\mathsf{id})}_i(\mu'_1, \mathsf{id}_1, \psi_1) \| \cdots \| (\mu'_i, \mathsf{id}_i, \psi_i) = 1$.

We use $\mathtt{TT}$ to denote concatenation of five-tuples $(\mu_j, \mathsf{id}_j, \psi_j, T_j, \sigma_j)$ and $\widetilde{\mathtt{TT}}$ to denote concatenation of three-tuples $(\mu_j, \mathsf{id}_j, \psi_j)$. $\mathsf{tv}, \widetilde{\mathsf{tv}}$ are the respective transcript validators

of $\mathrm{TT}, \widetilde{\mathrm{TT}}$. We give details on the rest of the protocol. At a high level, all of the phases are very similar to the non-reactive case except for the use of additional witnesses $(\mu_i', \mathsf{id}, \psi_i)$ which essentially correspond to the actual MPC execution of the reactive functionality. Specifically, the main new argument to prove security will be the unforgeability of the signature ψ_i for honest P_i on the message (μ_i', id) and that in a witness TT_j used by corrupt P_j, the id's in TT_j must be consistent with $T_i^{(\mathsf{id})}$ for honest P_i (the unique id under which honest parties release signatures under msk) which in turn forces $\mu_i', \mathsf{id}, \psi_i$ used as part of TT_j to be exactly as the ones released by P_i. As in the non-reactive case, we will be able to prove that the id corresponds to an incomplete local execution if there is one.

LOCAL SETUP PHASE. In the k-th *local setup phase* parties submit the authenticated secret shares of the master signing key as input to an unfair ideal functionality $\mathcal{F}_{\widehat{f}_k}^{\mathsf{ord}}$ that delivers outputs in the following order:

For $i = n$ down to 1:

(a) [**Main witness signed under global keys**] Party P_i obtains a random value $s_i^{(k)}$ along with a signature $\sigma_i^{(k)}$ on the message $T_i^{(k)} = (i, k, s_i^{(k)})$ signed under msk. $(\mathsf{sp}_i^{(k)})$

(b) [**Lock witness signed under global key**] For $j = n$ down to $i + 1$:

P_j obtains a signature $\sigma_{j,i}^{(k)}$ on message (j, i, k) under msk. $(\mathsf{sp}_{j,i}^{(k)})$

Observe that this phase is identical to the non-reactive case except now the values $s_i^{(k)}$ are completely random.

LOCAL EXCHANGE PHASE. Parties start exchanging messages corresponding to the local reactive MPC evaluating reactive function f_k. We denote this reactive MPC protocol as $\widetilde{\pi}^{(k)}$ defined by a pair of algorithms $\{\widetilde{\mathsf{nmf}}_j^{(k)}, \widetilde{\mathsf{tv}}_j^{(k)}\}_{j\in[m]}$. Note that we will be using the protocol obtained as a result of transformation procedure described above. Party P_1 starts by running $\widetilde{\mathsf{nmf}}_1^{(k)}$ to generate the first message $\mu_1^{(k)}$ of the protocol $\pi^{(k)}$ realizing the reactive function f_k. Then party P_2 upon receiving $\mu_1^{(k)}$ from P_1, invokes $\widetilde{\mathsf{nmf}}_2^{(k)}$ to generate $\mu_2^{(k)}$ and broadcasts this to all parties. The protocol proceeds like this till the very end when P_n sends the message $\mu_n^{(k)}$. More precisely, let $\mu_0^{(k)} = \widetilde{\mathrm{TT}}_0^{(k)} = \mathsf{NULL}$, and let protocol $\widetilde{\pi}^{(k)}$ be defined as $\{\widetilde{\mathsf{nmf}}_j^{(k)}, \widetilde{\mathsf{tv}}_j^{(k)}\}_{j\in[n]}$. For $j \in [n]$:

Upon receiving $\mu_{j-1}^{(k)}$ from party $P_{j-1 \bmod n}$, party $P_{j \bmod n}$ with input $x_{j \bmod n}^{(k)}$ and randomness $\omega_{j \bmod n}^{(k)}$ checks if $\widetilde{\mathsf{tv}}_{j-1}^{(k)}(\mu_{j-1}^{(k)}) = 1$ and if so, sets $\widetilde{\mathrm{TT}}_{j-1}^{(k)} = \widetilde{\mathrm{TT}}_{j-2}^{(k)} \| \mu_{j-1}^{(k)}$, sends $\mu_j^{(k)} = \widetilde{\mathsf{nmf}}_j^{(k)}(\widetilde{\mathrm{TT}}_{j-1}^{(k)}; (x_{j \bmod n}^{(k)}, \omega_{j \bmod n}^{(k)}))$ to all parties, and sets $\widetilde{\mathrm{TT}}_j^{(k)} = \widetilde{\mathrm{TT}}_{j-1}^{(k)} \| \mu_j^{(k)}$. $(\mathsf{ex}_j^{(k)})$

MASTER CLAIM PHASE. Denote by k the most recent completed execution, i.e., $\mathsf{ex}_n^{(k)} = 1$. It is possible that the $(k+1)$-th execution was never started (either there was an abort or the parties unanimously agreed to terminate all local executions), or there was an abort in the middle which means $\mathsf{sp}_i^{(k+1)} = 1$ or even $\mathsf{ex}_i^{(k+1)} = 1$ for some i. Note that party P_i needs to act only at time instances $\{\tau_{i,j}^{\mathsf{lock}}\}_{i>j}, \tau_i, \{\tau_{j,i}^{\mathsf{unlock}}\}_{j>i}$ We describe the master claim phase for P_i:

1. For $j = 1$ to $i - 1$: At time $\tau_{i,j}^{\mathsf{lock}}$ if $j = 1$ or $\mathsf{clm}_{j-1} = 1$ or $\mathsf{clm}_{i',j-1}^{\mathsf{unlock}} = 1$ for some i': $(\mathsf{clm}_{i,j}^{\mathsf{lock}})$

(a) If $j = 1$ and $i \neq 1$, then claim $\mathsf{Tx}_{i,1}^{\mathsf{lock}}$ using witness $(\mathrm{TT}_0 = \mathsf{NULL}, k', \sigma')$, where $(k', \sigma') = (k+1, \sigma_{i,1}^{(k+1)})$ if $\mathsf{sp}_1^{(k+1)} = 1$, else $(k', \sigma') = (k, \sigma_{i,1}^{(k)})$.

(b) Else, let $\mathbf{TT}$ be the set of transcripts that were revealed during the claim of $\mathsf{Tx}_{j-1}, \{\mathsf{Tx}_{i',j-1}^{\mathsf{unlock}}\}$. Let $\mathbf{ID}$ denote the set of id's that the transcripts in $\mathbf{TT}$ are consistent with, i.e., for $\mathrm{TT} \in \mathbf{TT}$, there is an $\mathsf{id} \in \mathbf{ID}$, such that $\mathsf{tv}_{j-1}^{(\mathsf{id})}(\mathrm{TT}) = 1$. Let id' denote the maximum value in $\mathbf{ID}$ and let TT' denote the corresponding transcript. Claim $\mathsf{Tx}_{i,j}^{\mathsf{lock}}$ using witness $\mathrm{TT}', \mathsf{id}', \sigma_{i,j}^{(\mathsf{id}')}$.

2. At time τ_i if (1) $i = 1$ or (2) $\mathsf{clm}_{i-1} = 1$ or (3) $\mathsf{clm}_{j,i-1}^{\mathsf{unlock}} = 1$ for some j or (4) $\mathsf{clm}_{j,i}^{\mathsf{lock}} = 1$ for some j: (clm_i)

(a) If $i = 1$, then claim Tx_1 using $(\mu_1^{(k')}, k', \psi_1^{(k')}, T_1^{(k')}, \sigma_1^{(k')})$ where k' is the maximum value such that $\mathsf{sp}_1^{(k')} = 1$, and $\mu_1^{(k')}$, $\psi_1^{(k')}$ is obtained by applying $\widetilde{\mathsf{nmf}}_1^{(k')}$ on inputs $x_1^{(k')}$ and randomness $\omega_1^{(k')}$.

(b) Else, let $\mathbf{TT}$ be the set of transcripts that were revealed during the claim of $\mathsf{Tx}_{i-1}, \{\mathsf{Tx}_{j,i-1}^{\mathsf{unlock}}\}, \{\mathsf{Tx}_{j,i}^{\mathsf{lock}}\}$. (Note that all these transcripts contain the first $i-1$ secret shares.) Let $\mathbf{ID}$ denote the set of id's that the transcripts in $\mathbf{TT}$ are consistent with, i.e., for $\mathrm{TT} \in \mathbf{TT}$, there is an $\mathsf{id} \in \mathbf{ID}$, such that $\mathsf{tv}_{i-1}^{(\mathsf{id})}(\mathrm{TT}') = 1$. Let id' denote the maximum value in $\mathbf{ID}$ and let TT' denote the corresponding transcript.

i. If $\mathsf{ex}_i^{(\mathsf{id}')} = 1$, then set $\mathrm{TT}_{i-1} = (\mu_1, \mathsf{id}', \psi_1, T_1, \sigma_1) \| \cdots \| (\mu_{i-1}, \mathsf{id}', \psi_{i-1}, T_{i-1}, \sigma_{i-1})$ where the messages $\mu_1, \ldots, \mu_{i-1}$ and the corresponding signatures $\psi_1, \ldots, \psi_{i-1}$ are obtained from the id'-th local exchange phase (i.e., from a transcript before) and the values $T_1, \ldots, T_{i-1}$ and the corresponding signatures $\sigma_1, \ldots, \sigma_{i-1}$ are obtained from TT'. Let $(\mu_i^{(\mathsf{id}')}, \mathsf{id}', \psi_i^{(\mathsf{id}')})$ be the message that P_i sent during the id'-th local exchange phase. Set $\mathrm{TT}_i = \mathrm{TT}_{i-1} \| (\mu_i^{(\mathsf{id}')}, \mathsf{id}', \psi_i^{(\mathsf{id}')}, T_i^{(\mathsf{id}')}, \sigma_i^{(\mathsf{id}')})$.

ii. Else: let $(\mu_i^{(\mathsf{id}')}, \mathsf{id}', \psi_i^{(\mathsf{id}')})$ be the message obtained by applying $\widetilde{\mathsf{nmf}}_i^{(\mathsf{id}')}$ on transcript $\widetilde{\mathrm{TT}}'$ that is implicit in TT', using input $x_i^{(\mathsf{id}')}$ and randomness $\omega_i^{(\mathsf{id}')}$ (i.e., exactly the inputs/random tape it would have used in the id'-th local execution). Set $\mathrm{TT}_i = \mathrm{TT}' \| (\mu_i^{(\mathsf{id}')}, \mathsf{id}', \psi_i^{(\mathsf{id}')}, T_i^{(\mathsf{id}')}, \sigma_i^{(\mathsf{id}')})$.

Claim Tx_i using witness TT_i and save the value TT_i to use in the next step.

3. For $j = i + 1$ to n: At time $\tau_{j,i}^{\mathsf{unlock}}$ if $\mathsf{clm}_{j,i}^{\mathsf{lock}} = 1$: $(\mathsf{clm}_{j,i}^{\mathsf{unlock}})$

(a) Claim $\mathsf{Tx}_{j,i}^{\mathsf{unlock}}$ using TT_i (from the previous step), $k', \sigma_{j,i}$ where $(*, k', \sigma_{j,i})$ was the witness used to claim $\mathsf{Tx}_{j,i}^{\mathsf{lock}}$.

This concludes the description of the master claim phase and of the protocol. Please see the full version for the formal description and the security proof.

5. CONCLUSIONS

We made a distinction between "on-chain" complexity (verification complexity imposed on miners) and "off-chain" complexity (that is borne by the protocol participants). In this paper we showed how to amortize the "on-chain"cost of secure computation

with penalties. Several important questions remain. Could we reduce the "on-chain" complexity of a single execution? Alternatively, can we derive the amortization result for the reactive case using only $O(nr)$ initial deposits? Also, can we improve the practicality of our schemes by possibly removing the need to do the signature generation/verification part inside the MPC?

Acknowledgements

Research of the first author is supported in part by NSF Grants CNS-1350619 and CNS-1414119, in part by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Research Office under contracts W911NF-15-C-0226, and an MIT Translational Fellowship. Research of the second author is supported by funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258 and NSF grant 1561209.

6. REFERENCES

[1] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via the bitcoin deposits. In *Bitcoin Workshop, FC*, 2014.
[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *IEEE Security and Privacy*, 2014.
[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. How to deal with malleability of bitcoin transactions, 2013. Available from http://arxiv.org/pdf/1312.3230.pdf.
[4] A. Back and I. Bentov. Note on fair coin toss via bitcoin. http://arxiv.org/abs/1402.3698, 2013.
[5] S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better - how to make bitcoin a better currency. In *Financial Cryptography*, 2012.
[6] I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *Crypto*, 2014.
[7] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, 1986.
[8] C. Decker and R. Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. http://www.tik.ee.ethz.ch/file/716b955c130e6c703fac336ea17b1670/duplex-micropayment-channels.pdf.
[9] O. Goldreich. Foundations of cryptography vol.2. 2004.
[10] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. 1987.
[11] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In *Bitcoin Workshop, Financial Cryptography*, 2016.
[12] A. Kiayias, H-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Eurocrypt*, pages 705–734, 2016.
[13] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P*, 2016.
[14] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *CCS*, 2014.
[15] R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *CCS*, 2015.
[16] R. Kumaresan, V. Vaikuntanathan, and P. Vasudevan. Improvements to secure computation with penalties. In *CCS*, 2016.
[17] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. In *CT-RSA*, 2010.
[18] Andrew Y. Lindell. Legally-enforceable fairness in secure two-party computation. In *CT-RSA*, 2008.
[19] G. Maxwell. 2011. https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.
[20] Rafael Pass and Abhi Shelat. Micropayments for decentralized currencies. In *22nd CCS*, 2015.
[21] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf.
[22] Peter Todd. OP_CLTV, 2014. https://github.com/petertodd/bips/blob/checklocktimeverify/bip-checklocktimeverify.mediawiki.
[23] Andrew Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

APPENDIX

A. IMPLEMENTATION OF $\mathcal{F}^*_{\mathrm{CR}}$

Earlier works described how to realize $\mathcal{F}^*_{\mathrm{CR}}$ in Bitcoin by means of refund transactions (cf. [6, Appendix F]). This proposed implementation required an ideal Bitcoin system, since an actual deployment in current Bitcoin network would be exposed to a malleability [3] attack. However, the `OP_CHECKLOCKTIMEVERIFY` [22] (abbrev. CLTV) softfork of November 2015 enables a simpler $\mathcal{F}^*_{\mathrm{CR}}$ implementation that does not rely a refund transaction, thus avoiding transaction malleability altogether. The November 2015 softfork added the new CLTV opcode to the Bitcoin scripting language, which enables execution of instructions conditioned upon the requirement that the transaction resides in block whose index is large enough (or that its timestamp is late enough).

Given `OP_CHECKLOCKTIMEVERIFY`, we can easily implement $\mathcal{F}^*_{\mathrm{CR}}$ (with $\phi_{s,r}(\cdot) = 1 \Leftrightarrow \mathsf{Hash}(\cdot) = h_0$) by using a Bitcoin script whose high-level description is as follows:

CLTV Pseudocode: pk_S, pk_R, h_0, τ are hardcoded.

```
if (block# > τ) then
  P_s can sign with sk_S to spend the coins(q)
else
  P_r can spend the coins(q) by
     signing with sk_R
     AND
     supplying w such that Hash(w) = h_0
```

An actual implementation as a Bitcoin script can be given as follows:

CLTV Bitcoin script:

```
<timeout> CHECKLOCKTIMEVERIFY IF HASH256
<h_0> EQUALVERIFY <P_r> CHECKSIGVERIFY ELSE
<P_s> CHECKSIGVERIFY ENDIF
```

The above script assumes that the redemption script will put on the top of stack w, sig_{P_r} with $\mathtt{SHA256d}(w) = h_0$ in the first case, or sig_{P_s} in the second case.

Irrespective of CLTV, let us note that our realization of $\mathcal{F}^*_{\mathrm{MSFE}}$ (cf. Figure 4) relies on a sender-specified circuit $\phi_{s,r}(\cdot)$ that verifies a signature. Thus, deployment of our protocols in Bitcoin requires a script instruction that verifies an arbitrary signature. In principle, such an instruction is not any more complex than the standard `OP_CHECKSIGVERIFY` instruction, but the current Bitcoin scripting language lacks this instruction. Since multiple other proposals would also benefit from an arbitrary signature verification instruction (see, e.g., [20, 11]), such an opcode might be added to Bitcoin in the future.