Shielded Computations in Smart Contracts Overcoming Forks*

Vincenzo Botta[2], Daniele Friolo[†2], Daniele Venturi[1], and Ivan Visconti[2]

[1]*Department of Computer Science, Sapienza University of Rome, Italy*
[2]*DIEM, University of Salerno, Italy*

Abstract

In this work, we consider executions of smart contracts for implementing secure multi-party computation (MPC) protocols on forking blockchains (e.g., Ethereum), and we study security and delay issues due to forks. In this setting, the classical double-spending problem tells us that messages of the MPC protocol should be confirmed on-chain before playing the next ones, thus slowing down the entire execution.

Our contributions are twofold:

- For the concrete case of fairly tossing multiple coins with penalties, we notice that the lottery protocol of Andrychowicz et al. (S&P '14) becomes insecure if players do not wait for the confirmations of several transactions. In addition, we present a smart contract that instead retains security even when all honest players immediately answer to transactions appearing on-chain. We analyze the performance using Ethereum as testbed.
- We design a compiler that takes any "digital and universally composable" MPC protocol (with or without honest majority), and transforms it into another one (for the same task and same setup) which maintains security even if all messages are played on-chain without delays. The special requirements on the starting protocol mean that messages consist only of bits (e.g., no hardware token is sent) and security holds also in the presence of other protocols. We further show that our compiler satisfies fairness with penalties as long as honest players only wait for confirmations once.

By reducing the number of confirmations, our protocols can be significantly faster than natural constructions.

Keywords: MPC, blockchains, finality, forks, smart contracts.

---

*This paper is the full version of an FC' 21 publication [BFVV21].

†Part of the work done during his PhD at Department of Computer Science, Sapienza University of Rome, Italy.

Contents

1 Introduction

The rise of blockchains[1] is progressively changing the way transactions are executed over the Internet. Indeed, the traditional client-server paradigm turns out to be insufficient when many parties want to perform a distributed computation, especially in cases where features like public verifiability and automatic punishment are desired. Blockchains through the execution of smart contracts naturally allow many players to perform a joint computation, even when they are not simultaneously online; moreover, they allow to publicly check the actions of all players[2] and enforce a proper behavior through financial punishments.

1.1 Forks, Finality and Double Spending

Typical blockchains experience some delays before a transaction can be considered confirmed. Indeed, a large part of the most used blockchains consists of a list of blocks that can temporary fork. In such cases, fork-resolution mechanisms decide which branch is eventually part of the list of blocks and which one is discarded, at the price of cutting off some transactions that for some time have appeared on the blockchain. These finality limitations generate delays and uncertainty, and a significant effort has been made recently to design blockchains with better finality [DMM$^+$20, BG17, PS17, PS18, GHM$^+$17, CPS18].

It is well known that the existence of transactions that appear and then disappear from a blockchain is the source of the (in)famous double-spending attack. The solution to the double spending problem is pretty harsh: the receiver of a payment will have to wait long time (i.e., until the transaction is confirmed and becomes irreversible) before taking future actions. Obviously, this can be problematic when an entire process consists of many sequential transactions and the confirmation time is long.

The double spending problem does not seem to extend to the case where another on-chain transaction is connected to the payment transaction. Indeed, in this case, if as a consequence of a fork the payment transaction disappears, then the connected transaction disappears too. This chaining of transactions related to the same process can be easily implemented through smart contracts.

1.2 Insecurity of Smart Contracts with Hasty Players

Since transactions are not immediately confirmed in a forking blockchain, the full execution of a smart contract with multiple sequential transactions might take too long. It would thus be natural to speed up the execution of smart contracts by playing messages immediately. Indeed, as mentioned above, by appropriately chaining the transactions of a smart contract, attacks exploiting the cancellation of a transaction like in the double-spending attack are not effective,[3] and therefore playing immediately without waiting confirmations could be a valid option.

However, we notice that forks can help an adversary to mount more subtle attacks. For example, let us consider a smart contract executed by two players, Alice and Bob, willing to establish jointly a random string: 1) Alice starts the protocol by sending to the smart contract a commitment to a random string r_1; 2) Bob sends a random string r_2 to the smart contract; 3) Alice then opens the commitment, and if the opening is valid the common string is defined to be $r = r_1 \oplus r_2$. For concreteness, say that Alice is honest and Bob is corrupted, and assume that a fork happens after Alice already sent the commitment. If Bob runs the protocol

[1]We use the terms "blockchain" and "distributed ledger" interchangeably.

[2]We will often use the two terms "party" and "player" as synonyms.

[3]Since we are considering protocols running entirely on-chain, double spending attacks can not be exploited to avoid the payment of some off-chain service.

honestly on the first branch, he gets to see Alice's opening, and thus he can completely bias the output on the other branch by just sending $r'_2 = r' \oplus r_1$ to the smart contract, for any value r' of his choice. The above scenario can be a serious threat for integrity of data and even confidentiality in other protocols. This motivating example clearly shows that, unless one has proven some kind of resilience to forks, it is certainly preferable to always wait that transactions are confirmed, at the price of having very slow executions of a smart contract. Such slowness could be unacceptable in some applications.

1.3 Why MPC on Blockchains?

Blockchains offer public verifiability of distributed computations, in the sense that in case of dispute everyone can verify what happened and when. Moreover, smart contracts can automatically punish whoever violates some a-priori established rules. Clearly, the above advantages are useful also when players are running a privacy-preserving computation, in the form of a multi-party computation (MPC) protocol.

A popular example of MPC that can benefit from a blockchain is e-voting, since public verifiability is an important property of remote elections and several systems rely on a bulletin board that can be instantiated with a blockchain. Another well known example is the one illustrated by Andrychowicz et al. [ADMM14, ADMM16] who, despite the very limited expressive power of Bitcoin transactions, have shown how to use blockchains to obtain fairness through penalties to MPC protocols with dishonest majority, somehow circumventing the impossibility result of Cleve [Cle86] (that holds without assuming setup).

Note that executing an MPC protocol on-chain allows players not to be online all at the same time. Moreover, differently from protocols running on a TCP/IP WAN where players must know each other's IP address beforehand[4], with the aid of a ledger any player can join a protocol execution by just reading[5] a transaction containing the required information (e.g., the functionality, the minimum number of parties, or any other identifying information).

The above features, and the dilemma about playing immediately risking security or waiting for confirmation making the entire process very slow, motivate our work aiming at obtaining smart contracts for fast/fair/secure/publicly-verifiable MPC protocols on forking blockchains.

We remark that at least some of the aforementioned advantages provided by our constructions do not come already from the use of payment channels. Consider for instance payment channels allowing to run a computation in large part off-chain. The use of similar channels for MPC would require players to be simultaneously online with point-to-point connections, therefore suffering of the issues discussed above.

1.4 Our Contributions

Fair lottery with penalties and fully hasty players. In Andrychowicz et al. [ADMM14, ADMM16] and in Kumaresan et al. [BK14, KB14] it was shown how to obtain fairness (i.e., the adversary should be discouraged from avoiding that honest players learn the output after he gets it) through penalties. The idea is that a player should deposit some coins of the underlying cryptocurrency and the smart contract should return the coins back only in case the player completes correctly the execution of the protocol defined by the smart contract.

In light of the negative result by Cleve [Cle86] on achieving fairness without honest majority, we will also consider fairness with penalties. Recall that we are planning to do so still admitting

[4] We remark that executing a protocol on a payment channel does not offer any advantage in terms of anonymity with respect to an off-chain execution.

[5] Blockchain identifiers are usually public pseudonyms not necessarily correlated with the real user identities. This feature offers some privacy compared to IP addresses.

that the blockchain could fork and trying to obtain fast executions avoiding as much as possible to wait for confirmations of transactions.

We analyze a variant of the attack described earlier that can be applied to a smart contract based on Andrychowicz *et al.* [ADMM14, ADMM16] protocol for securely realizing multi-party lotteries[6] The main difference with the toy example from above is that in their work each player commits to a random value r_i between 1 and n (where n is the total number of participants to the lottery), and then, after all the commitments have been opened, the winner of the lottery is defined to be the player $w = r_1 + \ldots + r_n \pmod{n} + 1$. An appealing feature of this protocol is that it achieves fairness with penalties: if a malicious player aborts the protocol (e.g., it does not open the commitment before a certain time bound), then a previously deposited amount of coins is automatically transferred to the honest players (i.e., to those that correctly opened the commitment on time).

We note that in the protocol of Andrychowicz et al. it is vital that players are non-hasty and therefore post new transactions only after the previous ones are already confirmed on the blockchain. Indeed, in the presence of hasty players, a malicious party can commit to a value r_i such that $\sum_i r_i \pmod{n} + 1 = i$, assuming that all players already opened the commitments on a minor branch of a fork. As our main contribution, in Sec. 4, we circumvent the limitations of [ADMM14, ADMM16], and present a smart contract that implements the lottery functionality[7] remaining secure even in the presence of hasty players. Fairness with penalties can be added without affecting the efficiency of the protocol. In fact, the smart contract we design is more general, in that it allows the players to establish a common, uniformly random, string (which in turn allows to run a lottery). When referring to our protocol depending on the context we will sometimes say lottery protocol and sometimes parallel coin-tossing protocol.

The main idea in our construction consists of combining unique signatures [Lys02] and random oracles (similarly to constructions of verifiable random functions) as follows: first of all, players compute unique signatures on input the concatenation of the ordered sequence of their public keys. Notice that as long as at least one player is honest, we have a long string that no PPT player could predict when selecting his public key. Then this long string is given in input to a random oracle, returning a uniformly distributed string as an output. The simulator will program the random oracle to force in the simulation the same random string obtained in the ideal-world execution.

There is still an attack that can be mounted. Assume that in the presence of a fork the entire protocol is executed in a branch. The adversary could take advantage of the output in one branch to decide to play the same first round or a different first round in the other branch biasing successfully the distribution of the output. To circumvent this problem, we make executions in different branches completely independent by also passing a branch id as input to the unique signature evaluation procedure. As branch id we take the hash of the block containing the last deposit. Therefore, when a protocol is entirely run in a branch, we have that the two branch ids are different and thus there is no point in adaptively choosing the same or a different message in another branch. Indeed, in any case, the outputs in different branches will be completely independent. In order to deal with multiple executions of the real-world protocol in different branches, we will also have a simulator that will play multiple times in the ideal world. Since the output of the protocol is a random string, it can be then used in many applications, not only to run a multi-party lottery. Note that our protocol is around 50% more

[6]Protocols of [ADMM14, ADMM16] is based on Bitcoin, but this makes no difference for our attack.

[7]We specify that our smart contract implements a parallel coin-tossing protocol. In some cases, we say that our smart contract implements a lottery protocol since we are interested in comparing our protocol with the lottery protocol of Andrychowicz et al. We remark that the output of a coin-tossing protocol can be used to compute a lottery winner.

efficient than the lottery of Andrychowicz et al. Let's say that t is the number of blocks needed for transaction confirmation, then our lottery protocol can be run by using only $t + 1$ blocks, whereas Andrychowicz et al. requires $2 \cdot t$ blocks to be completed.

Notice that this result makes no use of finality of transactions on a blockchain except from the one needed for calculating the output. The protocol can be run in the presence of fully hasty players, and is therefore very efficient.

We stress that we consider the adversary as a player that tries to exploit the existence of forks in order to bias the output of the smart contract. We are not modelling the adversary of the smart contract as a player that has control over forks, deciding which branch will eventually be discarded and which one will become permanently part of the blockchain. Obviously, a powerful adversary that has control over the forks can always play the protocol with a different input on each branch to then select the one that produced the output that she likes the most. This is unavoidable when there is little use of finality of transactions. Nevertheless, notice that in many cases this is not a problem. Indeed think of the need of establishing a random string to then use it as first round of a statistically hiding commitment scheme or as common reference string for a non-interactive zero-knowledge proof. In such scenarios the adversary can freely select a random string from any polynomially large set of randomly sampled strings without compromising any security. In other cases like playing bingo, the fact that the adversary can decide the string out of several candidates can be an issue.

Defining on-chain MPC with hasty players. We model the execution of a smart contract through transactions sent by different players is a computation involving multiple parties, and therefore when considering "security" of such computations we naturally refer to secure MPC. We formalize how to execute an MPC protocol in the presence of a blockchain. Our definition builds on the model of blockchain protocols, introduced in [PSS17, GG17]. A blockchain protocol allows players to keep a consistent record of transactions satisfying: (i) *consistency* (i.e., the view of the blockchain obtained by different players is identical up to pruning k blocks from the chain); and (ii) *liveness* (i.e., if all honest parties attempt to broadcast a message, then after w rounds, an honest party will see that message at depth k in the ledger).

Hence, running an MPC protocol π with the aid of a blockchain protocol simply means that the players exchange messages using the blockchain. Intuitively, a player is called *non-hasty* if she always waits that the previous messages are confirmed on the blockchain before sending the next one. On the other hand, a *hasty player* sends her next message by just looking at her current view of the blockchain (without pruning blocks). Apart from these changes, security is defined similarly as in the standard real-ideal world paradigm. Intuitively, in a protocol running with hasty players, block confirmation is not needed. However, if parties wants to keep a natural blockchain feature like public verifiability, the last message exchanged in the protocol must be necessary confirmed. Throughout the paper, when we talk about no confirmation we implicitly assume the last message is confirmed for public verifiability guarantees.

The definition of security in the presence of hasty players has importance in forking blockchains, in which miners can discard non-confirmed blocks, achieving consensus on other blocks. Our definition applies to forking sidechains too.

General-purpose MPC with hasty players and fairness with penalties. Having motivated the problem of running MPC protocols on forking blockchains, we show a general compiler to obtain smart contracts that implements ideal multi-party functionalities retaining security in the presence of forks and allowing players to be hasty.[8]

[8]In this work all our positive results consist of on-chain protocols for secure computation that are stand-alone secure, with security preserved under sequential composition. The reason why we do not try to obtain universal

In order to preserve security in the presence of forks, our compiler makes sure that, whenever an execution of the MPC protocol is repeated in multiple branches, each honest player protects herself from attacks exploiting forks by refusing to play again a message of the same execution of the protocol in case the blockchain shows a different prefix in the transcript of the execution. Specifically, if on one branch $\mathcal{B}_2$ there is a player that changes the message already played in a different branch $\mathcal{B}_1$, then each honest player that played already in $\mathcal{B}_1$ and is asked to play again on input a different prefix in $\mathcal{B}_2$ will abort the execution in $\mathcal{B}_2$. Clearly, this strategy forces a unique execution regardless of forks, and thus security holds even in the presence of fully hasty players. For more details about our compiler and its extension adding fairness with penalties see Sec. 5.

1.5 Related Work

Following [ADMM14, ADMM16], other works focus on achieving fairness with penalties for different applications of interest, including lotteries [BK14], decentralized poker [KMB15, BKM17], and general-purpose computation [BK14, KMS$^+$16, KB16, KVV16, BDD20]. In the more recent work of [BDD20] the authors proposed a fair with penalties MPC protocol with increased efficiency of the off-chain phase. In particular, the line of works by Kumaresan et al. relies on an elegant paradigm working in two phases: 1) during the first phase, players run an MPC protocol to obtain the output in hidden form (e.g., a secret sharing of the output); since the output is hidden, such a protocol can be executed off chain, as malicious aborts do not violate fairness; 2) during the second phase, the output is reconstructed in a fair manner on chain. Unfortunately, the security of this paradigm in the presence of hasty players is difficult to assess, as protocols relying on intermediate ideal functionalities (such as the "claim-or-refund" and "multi-lock" functionality [BK14, KB14], or a smart contract functionality [BDD20]), although implementable using Bitcoin or Ethereum, may be insecure when executed with hasty players. Moreover, known results about designing protocols in a hybrid model allowing to make calls to a functionality are applicable only to the classical setting where multiple executions of the same instance of the protocol due to forks are not possible. Also note that performing a large part of the computation off chain hinders one of the main advantages of blockchain-aided MPC (i.e., public verifiability of the entire process). Our results, in contrast, consider MPC protocols executed completely on-chain through smart contracts.

A different line of works, shows how to perform MPC in the presence of an abstract transaction ledger [KZZ16, GG17, BMTZ17, SSV19, CGJ19], of which Bitcoin and Ethereum are possible implementations. However, such an idealized ledger does not account for the possibility of forks, thus (implicitly) meaning that the players using it are modeled as non-hasty.

Our main contribution is a protocol to jointly generate a random beacon. It is known that there exist protocols suited for blockchains generating random values. A well known implementation is RANDAO [RT]. The smart contract introduced in RANDAO is similar to a smart contract implementation of the Andrychowicz et al. lottery protocol [ADMM16]. As we show in Sec. 4.2 even this smart contract is subject to attacks in case some party does not wait for the confirmation of the first phase of the protocol. On the contrary, our lottery protocol described in Sec. 4.3 is secure even if parties do not wait for block confirmations.

composability is that existing notions of universal composability with a ledger [CGJ19] rely on non-forking ledger functionalities and therefore on non-hasty players.

2 Preliminaries

2.1 Notation

Given an integer n, we let $[n] = \{1, \dots, n\}$. If x is a string, we denote its length by $|x|$; if $\mathcal{X}$ is a set, $|\mathcal{X}|$ is the number of elements in $\mathcal{X}$. When x is chosen randomly in $\mathcal{X}$, we write $x \leftarrow\$ \mathcal{X}$. When A is an algorithm, we write $y \leftarrow\$ \mathsf{A}(x)$ to denote a run of A on input x and output y; if A is randomized, then y is a random variable and $\mathsf{A}(x;\omega)$ denotes a run of A on input x and random coins $\omega \in \{0,1\}^*$.

Throughout the paper, we denote the security parameter by $\lambda \in \mathbb{N}$. A function $\nu(\lambda)$ is negligible in λ (or just negligible) if it decreases faster than the inverse of every polynomial in λ, i.e. $\nu(\lambda) \in O(1/p(\lambda))$ for every positive polynomial $p(\cdot)$. A machine is said to be probabilistic polynomial time (PPT) if it is randomized, and its number of steps is polynomial in the security parameter.

For a random variable $\mathbf{X}$, we write $\mathbb{P}[\mathbf{X} = x]$ for the probability that $\mathbf{X}$ takes a particular value x in its domain. A distribution ensemble $\mathbf{X} = \{\mathbf{X}(\lambda)\}_{\lambda\in\mathbb{N}}$ is an infinite sequence of random variables indexed security parameter $\lambda \in \mathbb{N}$. Two distribution ensembles $\mathbf{X} = \{\mathbf{X}(\lambda)\}_{\lambda\in\mathbb{N}}$ and $\mathbf{Y} = \{\mathbf{Y}(\lambda)\}_{\lambda\in\mathbb{N}}$ are said to be *computationally indistinguishable*, denoted $\mathbf{X} \approx_c \mathbf{Y}$ if for every non-uniform PPT algorithm D there exists a negligible function $\nu(\cdot)$ such that:

$$|\mathbb{P}[\mathsf{D}(\mathbf{X}(\lambda)) = 1] - \mathbb{P}[\mathsf{D}(\mathbf{Y}(\lambda)) = 1]| \leq \nu(\lambda).$$

When the above equation holds for all (even unbounded) distinguishers D, we say that $\mathbf{X}$ and $\mathbf{Y}$ are statistically close, denoted $\mathbf{X} \approx_s \mathbf{Y}$.

2.2 Standard Primitives

Public-key encryption. A public-key encryption (PKE) scheme is a tuple of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ specified as follows. (i) The randomized algorithm Gen takes as input the security parameter, and outputs a pair of keys (pk, sk); (ii) The randomized algorithm Enc takes as input a public key pk and a message $m \in \mathcal{M}$, and outputs a ciphertext c; (iii) The deterministic algorithm Dec takes as input a secret key sk and a ciphertext c, and outputs a value in $\mathcal{M} \cup \{\bot\}$ (where $\bot$ denotes decryption error). Correctness says that for every key $\lambda \in \mathbb{N}$, every (pk, sk) in the support of $\mathsf{Gen}(1^\lambda)$, and every message $m \in \mathcal{M}$, it holds that $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m$ with probability one over the randomness of Enc.

Definition 1 (Semantic security)**.** We say that $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ satisfies semantic security if for all PPT attackers $\mathsf{A} := (\mathsf{A}_0, \mathsf{A}_1)$ there exists a negligible function $\nu(\cdot)$ such that:

Signature schemes. A signature scheme is a tuple of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ specified as follows. (i) The randomized algorithm Gen takes as input the security parameter and outputs a secret key sk together with a public verification key pk; (ii) The deterministic algorithm Sign takes as input the secret key sk and a message $x \in \{0,1\}^*$ and outputs a signature y; (iii) The randomized algorithm Verify takes as an input the verification key pk, a message/signature pair (x, y) and outputs a decision bit.

Correctness says that for all $\lambda \in \mathbb{N}$, for all $(pk, sk) \in \mathsf{Gen}(1^\lambda)$, and for all $x \in \{0,1\}^*$ it holds that $\mathsf{Verify}(pk, x, \mathsf{Sign}(sk, x)) = 1$ (with probability one over the coin tosses of Verify).

We will need so called *unique* signature schemes, which satisfy two properties known as uniqueness and unforgeability as defined below.

Definition 2 (Uniqueness)**.** For every pk, x, y_0, y_1 with $y_0 \neq y_1$ there exists a negligible function $\nu(\cdot)$ such that the following holds for either $i = 0$ or $i = 1$:

$$\mathbb{P}\left[\mathsf{Verify}(pk, x, y_i) = 1\right] \leq \nu(\lambda).$$

In words, for every string pk and every x, there exists at most one value y that is a accepting signature of x.

Definition 3 (Unforgeability)**.** For all PPT *valid* attackers A there exists a negligible function $\nu(\cdot)$ such that:

$$\mathbb{P}\left[\mathsf{Sign}(sk, x) = y : \begin{array}{c}(pk, sk) \leftarrow_{\$} \mathsf{Gen}(1^\lambda) \\ (x, y) \leftarrow_{\$} \mathsf{A}^{\mathsf{Sign}(sk,\cdot)}(pk)\end{array}\right] \leq \nu(\lambda),$$

where A is called *valid* if it never queries m to its oracle.

Unique signatures are sometimes also known under the name of verifiable unpredictable functions, and exist based on a variety of assumptions [BR96, MRV99, Lys02, DY05].

Commitment schemes. A non-interactive commitment Commit is a PPT algorithm taking as input a message $m \in \{0,1\}^\ell$, and outputting a commitment $\gamma = \mathsf{Commit}(m; \delta)$, where $\delta \in \{0,1\}^*$ is the randomness used to generate the commitment. The pair (m, δ) is called the opening.

Intuitively, a secure commitment satisfies two properties called binding and hiding. The first property says that it is hard to open a commitment in two different ways. The second property says that a commitment hides the underlying message.

Definition 4 (Binding)**.** We say that a non-interactive commitment Commit is *perfectly binding* if pairs (m_0, δ_0), (m_1, δ_1) such that $m_0 \neq m_1$ and $\mathsf{Commit}(m_0; \delta_0) = \mathsf{Commit}(m_1; \delta_1)$ do not exist.

Definition 5 (Hiding)**.** We say that a non-interactive commitment Commit is *computationally hiding* if for all non-uniform PPT adversaries A the following quantity is negligible

$$\left|\mathbb{P}\left[\mathsf{A}^{\mathsf{LR}(0,\cdot,\cdot)}(1^\lambda) = 1\right] - \mathbb{P}\left[\mathsf{A}^{\mathsf{LR}(1,\cdot,\cdot)}(1^\lambda) = 1\right]\right|,$$

where the oracle $\mathsf{LR}(b, \cdot, \cdot)$ with hard-wired $b \in \{0,1\}$ takes as input pairs of messages $m_0, m_1 \in \{0,1\}^\ell$, and outputs $\mathsf{Commit}(m_b)$.

Secret sharing schemes. An n-party secret sharing scheme $(\mathsf{Share}, \mathsf{Recon})$ is a pair of poly-time algorithms specified as follows. (i) The randomized algorithm Share takes as input a message $m \in \mathcal{M}$ and outputs n shares $\sigma = (\sigma_1, \ldots, \sigma_n) \in \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$; (ii) The deterministic algorithm Recon takes as input a subset of the shares, say $\sigma_{\mathcal{I}}$ with $\mathcal{I} \subseteq [n]$, and outputs a value in $\mathcal{M} \cup \{\perp\}$.

Definition 6 (Threshold secret sharing)**.** Let $n \in \mathbb{N}$. For any $t \leq n$, we say that $(\mathsf{Share}, \mathsf{Recon})$ is an (t, n)-secret sharing scheme if it satisfies the following properties.

- **Correctness**: For any message $m \in \mathcal{M}$, and for any $\mathcal{I} \subseteq [n]$ such that $|\mathcal{I}| \geq t$, we have that $\mathsf{Recon}(\mathsf{Share}(m)_{\mathcal{I}}) = m$ with probability one over the randomness of Share.
- **Privacy**: For any pair of messages $m_0, m_1 \in \mathcal{M}$, and for any $\mathcal{U} \subset [n]$ such that $|\mathcal{U}| < t$, we have that

$$\{\mathsf{Share}(1^\lambda, m_0)_{\mathcal{U}}\}_{\lambda \in \mathbb{N}} \approx_c \{\mathsf{Share}(1^\lambda, m_1)_{\mathcal{U}}\}_{\lambda \in \mathbb{N}}.$$

2.3 Multi-Party Computation

We recall standard notion of UC-security for multi-party computation (MPC). Let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be a function, and consider n players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ executing a protocol π for computing f. Our default network model consists of the players interacting in synchronous rounds via private and authenticated point-to-point channels.

Intuitively, the security of π is formalized by comparing its execution in the real world (where an attacker may corrupt a subset of the players) with the ideal execution in which a trusted party computes the function f on behalf of the players.

The real model. In the real world, the protocol π is run in the presence of an adversary A coordinated by a non-uniform environment $\mathsf{Z} = \{\mathsf{Z}_\lambda\}_{\lambda \in \mathbb{N}}$. At the outset, Z chooses the inputs $(1^\lambda, x_i)$ for each player P_i, and gives $\mathcal{I}$, $\{x_i\}_{i \in \mathcal{I}}$ and z to A, where $\mathcal{I} \subseteq [n]$ represents the set of corrupted players and z is some auxiliary input. For simplicity, we only consider static corruptions (i.e., the environment decides who is corrupt at the beginning of the protocol). The parties then start running π, with the honest players P_i behaving as prescribed in the protocol (using input x_i), and with malicious parties behaving arbitrarily (directed by A). The attacker may delay sending the messages of the corrupted parties in any given round until after the honest parties send their messages in that round; thus, for every r, the round-r messages of the corrupted parties may depend on the round-r messages of the honest parties.

Z can interact with A throughout the course of the protocol execution. At the end of the execution Z additionally receives the outputs of the honest parties and must output a bit. We denote by $\mathbf{REAL}_{\pi,\mathsf{A},\mathsf{Z}}(\lambda)$ the random variable corresponding to Z's guess.

The ideal model. In the ideal world, a trusted third party evaluates the function f on behalf of a set of dummy players $(\mathsf{P}_i)_{i \in [n]}$. As in the real setting, Z chooses the inputs $(1^\lambda, x_i)$ for each honest player P_i, and gives $\mathcal{I}$, $\{x_i\}_{i \in \mathcal{I}}$ and z to the ideal adversary S, corrupting the dummy parties $(\mathsf{P}_i)_{i \in \mathcal{I}}$. Hence, honest parties send their input $x'_i = x_i$ to the trusted party, whereas the parties controlled by S might send an arbitrary input x'_i. The trusted party computes $(y_1, \ldots, y_n) = f(x'_1, \ldots, x'_n)$, and sends y_i to P_i. Finally, S gives to Z an arbitrary function of its view, and Z additionally receives the outputs of the honest parties and must output a bit. We denote by $\mathbf{IDEAL}_{f,\mathsf{S},\mathsf{Z}}(\lambda)$ the random variable corresponding to Z's guess.

The above specification of the ideal model automatically implies *fairness* (i.e., corrupted parties get the output if and only if honest parties do as well). Unfortunately, as shown by Cleve [Cle86], such a strong guarantee is impossible to achieve for some functionalities without assuming honest majority. For this reason, we also consider a weaker flavor of the ideal model yielding a middle-ground notion known as security with aborts, which is possible to achieve even in the presence of honest minority. Let $\mathcal{H} := [n] \setminus \mathcal{I}$. The only difference with the above specification is that the trusted party at first forwards only the outputs $\{y_i\}_{i \in \mathcal{I}}$ to the ideal adversary S. Hence, S might send either a message $(\texttt{continue}, \mathcal{H}')$ or $\texttt{abort}$ to the trusted party. In the former case, all the honest parties in $\mathcal{H}'$ are given their output y_i whereas the honest parties in $\mathcal{H} \setminus \mathcal{H}'$ receive an abort symbol $\perp$. In the latter case, all honest parties receive $\perp$. We denote by $\mathbf{IDEAL}_{f_\perp,\mathsf{S},\mathsf{Z}}(\lambda)$ the random variable corresponding to Z's final guess.

The definition. We are now ready to define security.

Definition 7 (UC-Secure MPC)**.** Let π be an n-party protocol for computing a function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$. We say that π t-securely UC-realizes f in the presence of malicious adversaries for every PPT adversary A there exists a PPT simulator S such that for every non-uniform PPT environment Z corrupting at most t parties the following holds:

$$\left\{\mathbf{REAL}_{\pi,\mathsf{A},\mathsf{Z}}(\lambda)\right\}_{\lambda\in\mathbb{N}} \approx_c \left\{\mathbf{IDEAL}_{f,\mathsf{S},\mathsf{Z}}(\lambda)\right\}_{\lambda\in\mathbb{N}}.$$

When replacing $\mathbf{IDEAL}_{f,\mathsf{S},\mathsf{Z}}(\lambda)$ with $\mathbf{IDEAL}_{f_\perp,\mathsf{S},\mathsf{Z}}(\lambda)$ we say that π t-securely computes f *with aborts* in the presence of malicious adversaries.

2.4 A Blockchain Model

Below, we describe verbatim the blockchain model of [GG17] (which in turn builds on [PSS17, GKL15]). A blockchain protocol Γ consists of the following algorithms:

- $\mathsf{UpdateState}(1^\lambda)$: It is a stateful algorithm that take as input a security parameter $\lambda \in \mathbb{N}$, and maintains a local state $st \in \{0,1\}^*$ which essentially consists of the entire blockchain (i.e., the sequence of minted blocks).
- $\mathsf{GetRecords}(1^\lambda, st)$: It takes as input the security parameter and a state $st \in \{0,1\}^*$. It outputs the longest *ordered* sequence of valid blocks (or simply blockchain) $\mathcal{B} = (\beta_1, \beta_2, \ldots)$ contained in the state variable, where each block β in the chain itself contains an *unordered* sequence of records/messages $(m_1, m_2, \ldots)$.
- $\mathsf{Broadcast}(1^\lambda, m)$: It takes as input the security parameter and a message $m \in \{0,1\}^*$, and broadcasts the message over the network to all nodes executing the blockchain protocol. It does not give any output.

The blockchain protocol is also parameterized by a validity predicate V that captures the semantics of any particular blockchain application. The validity predicate takes as input a sequence of blocks $\mathcal{B}$ and outputs a bit, where the value 1 certifies the validity of the blockchain $\mathcal{B}$. Since V is immaterial for our purposes, in what follows we simply omit it.

Blockchain execution. Each participant in the protocol runs $\mathsf{UpdateState}$ to keep track of the latest blockchain state. This corresponds to listening on the broadcast network for messages from other nodes. $\mathsf{GetRecords}$ is used to extract an ordered sequence of blocks encoded in the blockchain state variable, which is considered as the common public ledger among all the nodes. Finally, $\mathsf{Broadcast}$ is used by a party when it wants to post a new message on the blockchain; such messages are accepted by the blockchain protocol only if they satisfy the validity predicate given the current state.

The execution of a blockchain protocol $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ is directed by an environment $\mathsf{Z}(1^\lambda)$ which activates the parties as either honest or corrupt, and is also responsible for providing inputs/records to all parties in each round. All the corrupted parties are controlled by the adversary A, which is also responsible for delivery of all network messages. Honest parties start by executing $\mathsf{UpdateState}$ on input 1^λ, with an empty local state $st = \varepsilon$. Then, the protocol execution proceeds in rounds that model times steps, as detailed below.

- In round $r \in \mathbb{N}$, each honest player P_i potentially receives messages from Z, and incoming network messages (delivered by A). It may then perform any computation, broadcast a message to all other players (which will be delivered by the adversary as explained below), and update its local state st_i. It could also attempt to add a new block to its chain i.e., run the mining procedure.
- The attacker A is responsible to deliver all messages sent by parties (honest or corrupted) to all other parties. The adversary cannot modify the content of messages broadcast by honest players, but it may delay or reorder the delivery of a message as long as it eventually delivers all messages within a certain time limit.
- At any point, Z can communicate with A or access $\mathsf{GetRecords}(1^\lambda, st_i)$ where st_i is the local state of player P_i.

With the notation $\mathcal{B} \preceq \mathcal{B}'$, we denote that the blockchain $\mathcal{B}$ is a prefix of $\mathcal{B}'$. We also let $\mathcal{B}^{\lceil k}$ be the chain resulting from pruning the last k blocks in $\mathcal{B}$. Let $\mathbf{EXEC}_{\Gamma,\mathsf{A},\mathsf{Z}}(\lambda)$ be the random variable denoting the joint view of all parties in the execution of protocol Γ with adversary A, and environment Z. Note that this view fully determines the execution.

Blockchain properties. We now define two natural guarantees that are respected by an ideal ledger. The first property, called *consistency*, intuitively states that the view of the blockchain obtained by different players is identical up to pruning a certain number of blocks from the top of the chain. Let $\mathsf{Consistent}^k(\cdot)$ be the predicate that returns 1 iff for all rounds $r \leq \tilde{r}$, and all parties $\mathsf{P}_i, \mathsf{P}_j$ (potentially the same) such that P_i is honest at round r with blockchain $\mathcal{B}$ and player P_j is honest at round $\tilde{r}$ with blockchain $\tilde{\mathcal{B}}$, we have that $\mathcal{B}^{\lceil k} \preceq \tilde{\mathcal{B}}$.

Definition 8 (Chain consistency)**.** A blockchain protocol Γ satisfies $k(\cdot)$-consistency with adversary A and environment Z, if there exists a negligible function $\nu(\cdot)$ such that for every $\bar{k} > k(\lambda)$, the following holds:

$$\mathbb{P}\left[\mathsf{Consistent}^{\bar{k}}(view) = 1 : \ view \leftarrow\!\!\$\ \mathbf{EXEC}_{\Gamma,\mathsf{A},\mathsf{Z}}(\lambda)\right] \geq 1 - \nu(\lambda).$$

We note that previous work considered an even stronger property, called *persistence*, stipulating that if some honest player reports a message m at depth k in its local ledger, then m will be always reported in the same position and equal or more depth by all honest parties. We omit a formal definition, as this property is not required for our purposes.

The second property, called *liveness*, intuitively says that if all honest parties attempt to broadcast a message m, then after w rounds, an honest party will see m at depth k in the ledger. Let $\mathsf{Live}^k(\cdot, w)$ be the predicate that returns 1 iff for any w consecutive rounds $r, \ldots, r+w$ there exists some round $r' \in [r, r+w]$ and index $i \in [n]$ such that: (1) P_i is honest and received a message m at round r, and (2) for every player P_j that is honest at $r+w$ with blockchain $\mathcal{B}$, it holds that $m \in \mathcal{B}^{\lceil k}$.

Definition 9 (Liveness)**.** A blockchain protocol Γ satisfies $(w(\cdot), k(\cdot))$-liveness with adversary A and environment Z, if there exists a negligible function $\nu(\cdot)$ such that for every $\bar{w} \geq w(\lambda)$ the following holds:

$$\mathbb{P}\left[\mathsf{Live}^{k}(view, \bar{w}) = 1 : \ view \leftarrow\!\!\$\ \mathbf{EXEC}_{\Gamma,\mathsf{A},\mathsf{Z}}(\lambda)\right] \geq 1 - \nu(\lambda).$$

3 Running MPC on Forking Blockchains

In this section, we formalize different ways to run an MPC protocol with the aid of a blockchain. In Sec. 3.1 we specify what it means to run an MPC protocol on the blockchain both in the presence of hasty and non-hasty players. The security definition appears in Sec. 3.2.

3.1 Blockchain-Aided MPC

Next, we define what it means to run an n-party protocol π for securely computing some function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ over a blockchain protocol Γ.

Intuitively, running π on Γ simply means that the players write the protocol's messages on the blockchain instead of using point-to-point connections. However, since the blockchain may fork, the protocol's participants have to choose how to manage possibly unconfirmed blocks that are part of the current chain. Looking ahead, this choice will have impact both on the efficiency and on the security of the protocol execution. In particular, we distinguish between *hasty* and *non-hasty* players as formalized below.

Non-hasty execution. Roughly speaking, a player is said to be *non-hasty* if it always decides its next message by looking at the transcript of the protocol that is obtained by pruning the last k blocks of the blockchain, where k is the parameter for the consistency property of the underlying blockchain.

Definition 10 (Non-hasty player)**.** Let $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ be a blockchain protocol with k-consistency. A player P_i is said to be *non-hasty* if it behaves as follows:

- Initialize $\tau_i^{(0)} := \varepsilon$, $st_i := \varepsilon$ and $r_i := 0$.
- Run the following loop:
  - Update the state st_i by running $\mathsf{UpdateState}(1^\lambda)$, and retrieve $\mathcal{B}_i \leftarrow\$ \mathsf{GetRecords}(st_i)$ until the partial transcript $\tau^{(r_i)}$ is contained in $\mathcal{B}_i^{\lceil k}$.
  - If the protocol is over (i.e., the transcript $\tau^{(r_i)}$ is sufficient for determining the output), output the value y_i as a function of $\tau_i^{(r_i)}$ and terminate.
  - Else, compute the next protocol message $m_i^{(r_i+1)}$, invoke $\mathsf{Broadcast}(m_i^{(r_i+1)})$, and set $r_i := r_i + 1$.

Hasty execution. On the other hand, a player is *hasty* if it decides and broadcasts its next message by looking at the latest version of the blockchain (i.e., without pruning blocks). Since the consistency property does not hold for the last k blocks, hasty players may retrieve different protocol's transcripts as the protocol proceeds. In particular, it may happen that at a given time step party P_i reads from the blockchain a partial transcript $\tau^{(\tilde{r})}$, whereas at a later time step the same player reads $\tau^{(\tilde{r}')}$ for some $\tilde{r}' < \tilde{r}$. This is due to the fact that some of the messages contained in $\tau^{(\tilde{r})}$ may end up in unconfirmed blocks, and thus be discarded.

Definition 11 (Hasty execution)**.** Let $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ be a blockchain protocol with k-consistency. A player P_i is said to be *hasty* if it behaves as follows:

- Initialize $\tau_i^{(0)} := \varepsilon$ and $st_i := \varepsilon$.
- Run the following loop:
  - Update the state st_i by running $\mathsf{UpdateState}(1^\lambda)$, and let $\mathcal{B}_i \leftarrow\$ \mathsf{GetRecords}(st_i)$.
  - Let $\tilde{r} \geq 0$ be the maximum value such that the partial transcript $\tau^{(\tilde{r})} \in \mathcal{B}_i$.
  - If the protocol is over (i.e., the transcript $\tau^{(\tilde{r})}$ is sufficient for determining the output), output the value y_i as a function of $\tau^{(\tilde{r})}$ and terminate.
  - Else, compute the next protocol message $m_i^{(\tilde{r}+1)}$ and invoke $\mathsf{Broadcast}(m_i^{(\tilde{r}+1)})$.

More generally, we call φ-hasty a player that is non-hasty until a partial transcript $\tau^{(\varphi)}$ is at least k blocks deep in the blockchain, and afterwards it starts being hasty. We sometimes call φ the *finality parameter*. Note that a 0-hasty player is identical to a hasty player, whereas an ∞-hasty player is identical to a non-hasty player. We call (χ, φ)-hasty a player that is hasty for the first χ rounds, and then behaves like a φ-hasty player.

3.2 Security in the Presence of Hasty Players

We can now define security of MPC protocols running on the blockchain. As in the standard setting, the definition compares a protocol execution in the real world with one in the ideal setting where a trusted party is made available. The main difference with the standard definition is that the attacker A is given black-box access to the algorithms in Γ, which it can use arbitrarily. The simulator is not allowed to control the blockchain (i.e. it must simulate the view of the adversary while invoking the algorithms in Γ on behalf of the honest players).

The real model: This is the execution of π on Γ, where the honest players are φ-hasty. As usual, the adversary A is coordinated by a non-uniform distinguisher D. At the outset, D

chooses the inputs $(1^\lambda, x_i)$ for each player P_i, and gives $\mathcal{I}$, $\{x_i\}_{i\in\mathcal{I}}$ and z to A, where $\mathcal{I} \subseteq [n]$ represents the set of corrupted players and z is some auxiliary input. The parties then start running π on Γ, with the honest players P_i being φ-hasty and behaving as prescribed in π (using input x_i), and with malicious parties behaving arbitrarily (directed by A). At some point, A gives to D an arbitrary function of its view; note that the latter includes the view generated via $\mathbf{EXEC}_{\Gamma,\mathsf{A},\mathsf{D}}(\lambda)$ in the blockchain protocol. Finally, D receives the outputs of the honest parties and must output a bit. We denote by $\mathbf{REAL}^{\Gamma,\varphi}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ the random variable corresponding to D's guess.

The ideal model: This is identical to the ideal model for standard MPC (Sec. 2.3), with the only difference that the simulator S is also responsible for simulating the attacker's view corresponding to the interaction of the honest players with the blockchain. The latter is achieved using the algorithms of the underlying blockchain protocol Γ. We denote by $\mathbf{IDEAL}^{\Gamma}_{f,\mathsf{S},\mathsf{D}}(\lambda)$ and $\mathbf{IDEAL}^{\Gamma}_{f_\perp,\mathsf{S},\mathsf{D}}(\lambda)$ the random variable corresponding to D's guess in the ideal world, where the latter is for the case of security with aborts.

Definition 12 (Secure MPC in the presence of hasty players)**.** Let π be an n-party protocol run over a blockchain protocol Γ. We say that π t-securely computes f in the presence of φ-hasty players and malicious adversaries if for every PPT adversary A there exists a PPT simulator S such that for every non-uniform PPT distinguisher D corrupting at most t parties the following holds:

$$\left\{\mathbf{REAL}^{\Gamma,\varphi}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)\right\}_{\lambda\in\mathbb{N}} \approx_c \left\{\mathbf{IDEAL}^{\Gamma}_{f,\mathsf{S},\mathsf{D}}(\lambda)\right\}_{\lambda\in\mathbb{N}}.$$

When replacing $\mathbf{IDEAL}^{\Gamma}_{f,\mathsf{S},\mathsf{D}}(\lambda)$ with $\mathbf{IDEAL}^{\Gamma}_{f_\perp,\mathsf{S},\mathsf{D}}(\lambda)$ we say that π t-securely computes f *with aborts* in the presence of φ-hasty players and malicious adversaries.

Remark 1 (On $\varphi = \infty$)**.** *One may think that every protocol π that t-securely computes f (with or without aborts) in the presence of malicious adversaries, must t-securely compute f (with or without aborts) in the presence of ∞-hasty (i.e., non-hasty) players and malicious adversaries.*

Remark 2 (On $\varphi = 0$)**.** *Note that when the players are fully hasty (i.e., $\varphi = 0$), the adversary's view in the real world may include multiple executions of the original protocol π (upon the same inputs chosen by the distinguisher). This view may not be possible to simulate in the ideal world, where the simulator can invoke the ideal functionality f only once.*

For this reason, whenever $\varphi = 0$, we implicitly assume that the simulator is allowed to query the ideal functionality f multiple times. Note that this yields a meaningful security guarantee only for certain functionalities f, similarly to the setting of resettably secure computation [GS09].

Remark 3 (On the power of the adversary)**.** *We stress that we assume that the adversary of the MPC protocol has no impact on the execution of the consensus protocol of the underlying blockchain. Note that if we would instead assume that the adversary of the MPC protocol also creates new branches and/or contributes in deciding which branch of a fork is eventually confirmed on the blockchain then he can have an unfair advantage. Indeed the adversary can start more branches when he does not like the output computed in a branch, and/or can decide which output among the various outputs appearing in different branches should be confirmed on the blockchain. Obviously the above unfair advantages are unavoidable and our protocol is still secure by introducing the unavoidable real-world attack into the ideal world, similarly to the classical fairness issue resolved through aborts in the ideal world.*

Remark 4 (On public verifiability)**.** *We notice that any on-chain MPC protocol with hasty players admits the case where a honest player complete her execution computing an output that* *does not necessarily correspond to the transcript that others later on will see on the blockchain. In other words, the local output computed by players could not match the publicly verifiable execution that remains visible on the blockchain. The reason why public verifiability could fail is that an execution of the protocol could be entirely contained in a branch of a fork that will not become permanent in the blockchain. The above issue is intrinsic in all protocols played on-chain in the presence of forks and hasty players. An obvious solution for a honest player consists of waiting that the last message of the protocol is confirmed on the blockchain and only after that the computation ends returning the computed output.*

Random oracle model. Our result in Sec. 4.3 are secure in the random oracle model (ROM). The definition remains the same, except that each player in the real world has now access to a truly random hash function Hash chosen at the beginning of the experiment. The simulator of the ideal world can program the random oracle.

4 Parallel Coin Tossing

A coin-tossing protocol allows a set of players to agree on a uniformly random string, and has many important applications (e.g., it allows to easily implement a decentralized lottery). Our protocol leverages standard techniques to achieve fairness with penalties, but does not require finality (thus allowing players to be fully hasty). We start summarizing the protocol of [ADMM16] below and we show that their protocol becomes completely insecure in the presence of hasty players. This naturally leads to our new protocol, which we describe and analyze in Sec. 4.3 and Sec. 4.4.

4.1 The Protocol of Andrychowicz et al.

Recall that in the Bitcoin ledger, each account is associated to a pair of keys (pk, sk), where pk is the verification key of a signature scheme—representing the address of an account—while sk is the corresponding secret key used to sign (the body of) the transactions. Each block on the ledger contains a list of transactions, and new blocks are issued by an entity called *miner*. The blockchain is maintained via a consensus mechanism based on proof of work; users willing to add a transaction to the ledger forward it to the miners, which will try to include it in the next minted block.

In the description below, we say that a transaction is *valid* if it is computed correctly (i.e., the signature is valid, the coins have not been spent already, and so on), and that it is *confirmed* if it appears in the common-prefix of all the miners (i.e., it is at least k-blocks deep in the ledger). Each transaction Tx includes:

- A set of input transactions $\mathsf{Tx}_1, \mathsf{Tx}_2, \cdots$ from which the coins needed for the actual transaction Tx are taken;
- A set of input scripts containing the input for the output scripts of $\mathsf{Tx}_1, \mathsf{Tx}_2, \cdots$;
- An output script defining in which condition Tx can be claimed;
- The number of coins taken from the redeemed transactions;
- A time lock t specifying when Tx becomes valid (i.e., a time-locked transaction won't be accepted by the miners before time t has passed).

The construction by [ADMM14, ADMM16] relies on a primitive called *time-locked commitment*. Let n denote the number of parties. Each party P_j creates $n-1$ $\mathsf{Commit}^j_{i\neq j}$ transactions containing a commitment to its lottery value. In particular, the output script of such a transaction ensures that it can be claimed either by P_j via an Open^j_i transaction exhibiting a valid opening for the commitment, or by another transaction that is signed by both P_j and P_i. Before posting these transactions on the ledger, P_j creates a time-locked transaction $\mathsf{PayDeposit}_i^j$ redeeming Commit_i^j, sends it off-chain to each $\mathsf{P}_{i\neq j}$, and finally posts all the Commit_i^j transactions on the ledger. In case P_j does not open the commitment before time τ, then each recipient of a $\mathsf{PayDeposit}_i^j$ transaction can sign it and post it on the ledger; since time τ has passed, the miners will now accept the transaction as a valid transaction redeeming Commit_i^j. More in details:

Deposit phase: Each player P_j computes a commitment $y_j = \mathsf{Hash}(x_j||\delta_j)$, where δ_j is some randomness, sends off-chain the $\mathsf{PayDeposit}_i^j$ transactions (with time-lock τ) to each $\mathsf{P}_{i\neq j}$, and posts the Commit_i^j transactions.

Betting phase: P_j bets one coin in the form of a transaction $\mathsf{PutMoney}_j$ (redeeming a previous transaction held by P_j, and with P_j's signature as output script). All the players agree and sign off-chain a $\mathsf{Compute}$ transaction taking as input all the $(\mathsf{PutMoney}_j)_{j\in[n]}$ transactions, and then the last player that receives the $\mathsf{Compute}$ transaction posts it on the ledger. In order to claim this transaction, a player $\mathsf{P}_{w'}$ must exhibit the openings of the commitments of all participants: The script checks that the openings are valid, computes the index of the winner w (as a function of the values $x_1, \ldots, x_n$), and checks that $w' = w$ (i.e., the only participant that can claim the $\mathsf{Compute}$ transaction is the winner of the lottery).

Compensation phase: After time τ, in case some player P_j did not send all of its $\{\mathsf{Open}_i^j\}_{i\in[n],i\neq j}$ transactions, all the other players $\mathsf{P}_{i\neq j}$ can post the $\mathsf{PayDeposit}_i^j$ transaction, thus obtaining a compensation.

4.2 A Simple Attack in the Presence of Hasty Players

The main idea behind our attack is that, in the presence of hasty players, the protocol's messages can end-up answering messages appeared on (still) unconfirmed blocks. By looking at different branches of a fork, an attacker can try to change an old (in the sense that even an answer to it has already been published on-chain) unconfirmed transaction by re-posting it, with the hope that it will end-up on a different branch and become part of the common prefix. This essentially corresponds to a reset attack on the protocol.

The construction described above relies on the (implicit) assumption that the players are non-hasty. In particular, each player P_j should wait to post its $\mathsf{PutMoney}_i^j$ transaction only after all the Commit_i^j transactions are confirmed on the ledger, in such a way that all players are aligned on the same branch (and so the miners have the $\{\mathsf{Commit}_i^j\}_{i\in[n],j\neq i}$ transactions in their common prefix). In the case of hasty players, when a fork occurs, an attacker can take advantage of the openings of the other parties played in a faster branch in order to bias the result of the lottery on a slower branch. If eventually the slower branch remains permanently in the blockchain, then clearly the attack is successful.

For concreteness, let us focus on Blum's coin tossing, in which the winner is defined to be $w = x_1 + \ldots + x_n \mod n+1$. Consider the following scenario:

- The (hasty) players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ run a full instance of the protocol; note that this requires at least 3 blocks.
- The attacker P_n hopes to see a fork containing all the $\{\mathsf{Commit}_j^i\}$ transactions of the other $n-1$ players.
- Since the attacker P_n now knows the openings $x_1, \ldots, x_{n-1}$, it can post a new set of $\{\mathsf{Commit}'^i_n\}_{i\in[n],i\neq n}$ transactions containing a commitment to a value x'_n such that $x_1 + \ldots + x_{n-1} + x'_n \mod n+1 = n$.

In case the new set of transactions ends up on a different branch which is finally included in the common prefix, P_n wins the lottery. In the next section, we propose a new protocol that does not suffer from this problem.

4.3 Our PCT Protocol

We now present a parallel coin-tossing (PCT) protocol on blockchain that is secure in the presence of hasty players. The main challenge that we face is that the protocol must prevent an adversary from choosing adaptively her contribution to the coin tossing in a branch of a fork, after possibly seeing the contributions of the other players in different branches.

We tackle this problem by requiring that each honest party computes his contribution by evaluating a unique signature upon input the public keys of all players. Notice that if the adversary A sees some signatures in a branch, and changes her public key in another branch, then A cannot predict the signatures of the honest players on this other branch by the unforgeability property of the signature scheme, and thus A will not manage to bias the final output. Hence, we hash the concatenation of all the signatures in order to determine the final output. Assuming that the hash function is modelled as a random oracle, we would like to argue that the output of the protocol looks uniform.

However, the following subtlety arises. Assume without loss of generality that only P_n is corrupt and that the protocol proceeds until the end on a given branch of the blockchain. Denote by pk_n the public key chosen by the attacker. Further, assume that A notices another branch where all honest players have already sent their public keys. Now, the adversary can either: (i) publish a different public key pk'_n, or (ii) publish the same public key pk_n as in the other branch. In case A "likes" the outcome of the protocol on the first branch, she will choose option (ii) and thus can bias the protocol output.

To avoid the above attack, we identify each branch with a string bid that is uniquely associated to it, and include bid as part of the message to sign. Intuitively, this solves the previous problem as, even if all the public keys stay the same on two different branches, the value bid will change thus ensuring that the protocol output will also be different (and uniformly random). We proceed with a more detailed description of our protocol (see also Fig. 1).[9]

- One of the players chooses a random value sid that represents the identifier of the current protocol execution, and publishes sid on the blockchain.
- Each player P_i willing to participate generates the public and private keys for the unique signature $(pk_i, sk_i) \leftarrow_{\$} \mathsf{Gen}(1^\lambda)$, and publishes pk_i on the blockchain.
- Each player P_i lets $y_i = \mathsf{Sign}(sk_i, pk_1||\cdots||pk_n||sid||bid)$, where bid is the hash of the blockchain[10] up to the block that contains the last public key, and publishes y_i on the blockchain.
- Each player P_i checks that $\mathsf{Verify}(pk_j, x, y_j) = 1$ for all $j \neq i$, where $x = pk_1||\cdots||pk_n||sid||bid$, and outputs $\mathsf{Hash}(y_1||\cdots||y_n)$.

We stress that thanks to the value bid, the protocol execution becomes branch dependent. In particular, the chances of success of a corrupted P_j to bias the output are not affected by the potential use of different public keys in branches of a fork corresponding to a protocol run with a given sid.

4.4 Security Analysis

Let f^{pct} be the n-party functionality that picks a uniformly random string ω and sends it to all the n parties. The theorem below establishes the security of our coin-tossing protocol in the (programmable) random oracle (RO) model. We note that the security of the original protocol

[9]Note that our protocol can be run on generic blockchains. In App. A, we provide an implementation using Ethereum smart contracts, but the protocol can also be implemented in Bitcoin using the opcode OP_RETURN in case players do not need to get fairness with penalties.

[10]For efficiency the hash can be more simply applied to the block containing pk_n. Nevertheless, for the sake of simplicity of the protocol description and of the security analysis we will stick with hashing the entire blockchain.

Parallel Coin Tossing Protocol π^*_{pct}

Let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme with message space $\mathcal{M} = \{0,1\}^*$, and $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function.

- P_1 picks $sid \leftarrow\$ \{0,1\}^\lambda$, and runs $\mathsf{Broadcast}(sid)$.
- For each $i \in [n]$, P_i generates $(pk_i, sk_i) \leftarrow\$ \mathsf{Gen}(1^\lambda)$ and runs $\mathsf{Broadcast}(pk_i)$.
- For each $i \in [n]$, P_i executes $\mathcal{B}_i \leftarrow\$ \mathsf{GetRecords}(1^\lambda, \mathsf{UpdateState}(1^\lambda))$ until all public keys $pk_1, \ldots, pk_n$ are contained in $\mathcal{B}_i$, and then defines $bid := \mathsf{Hash}(\mathcal{B}_i)$.
- For each $i \in [n]$, P_i computes $y_i = \mathsf{Sign}(sk_i, x)$, where $x = pk_1 || \ldots || pk_n || sid || bid$ and runs $\mathsf{Broadcast}(y_i)$.
- For each $i, j \in [n]$, with $i \neq j$, P_i checks that $\mathsf{Verify}(pk_i, x, y_i) = 1$, and, if so, it outputs $\mathsf{Hash}(y_1 || \cdots || y_n)$ and else it aborts.

Figure 1: Our new protocol for parallel coin tossing.

by Andrychowicz et al. [ADMM14, ADMM16] also relies on the RO heuristic, as do all currently known analysis of blockchain protocols [GKL15, PSS17].

Theorem 1. *If* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is a unique signature scheme, the protocol of Fig. 1 securely implements the functionality* f^{pct} *in the presence of hasty players and malicious adversaries with aborts, in the programmable RO model.*

We need to show that for every PPT adversary A, there exists a PPT simulator S such that no non-uniform PPT distinguisher D can tell apart the experiments $\mathbf{REAL}^{\Gamma,0}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ and $\mathbf{IDEAL}^{\Gamma}_{f^{\mathsf{pct}}_{\perp},\mathsf{S},\mathsf{D}}(\lambda)$. In particular, the simulator needs to simulate the interaction of the honest players with the blockchain protocol Γ as it happens in the real experiment.

Recall that the ideal coin-tossing functionality does not take any input, and returns a random string ω to all parties. Intuitively, the simulator S will just emulate a real execution of the protocol by reading the attacker's messages from the blockchain. Furthermore, after querying the ideal functionality, S will program the random oracle upon input ω. At the same time, S needs to simulate the answer to A's random oracle queries q, which is done as follows:

- If A already queried the random oracle upon q, obtaining answer $\tilde{\omega}$, then S returns the same value $\tilde{\omega}$;
- Else, if $pk_1, \ldots, pk_n$ are not all available on the blockchain, S answers the query q with a fresh random value $\tilde{\omega}$;
- Otherwise, S parses q as $\tilde{y}_1 || \ldots || \tilde{y}_n$, and checks that $\mathsf{Verify}(pk_i, x, \tilde{y}_i) = 1$ for all $i \in [n]$, where $x = pk_1 || \ldots || pk_n || sid || bid$ and bid is derived by hashing the blockchain up to the block containing the last public key. If the check passes, S programs $\mathsf{Hash}(q)$ to the value ω obtained by $f^{\mathsf{pct}}_{\perp}$, else it answers with a fresh random value $\tilde{\omega}$.

The above strategy does not consider the fact that the real protocol may be run multiple times on different branches. However, since each execution has associated a different $bid' \neq bid$, the simulator can simply query again the functionality in order to obtain a new random value ω', and program the random oracle on ω' in the execution corresponding to bid'. Roughly speaking, the attacker A can potentially take advantage from the following three actions: (i) A can refuse to publish her signed message; (ii) A can try to change the public keys of corrupted parties over different branches; (iii) A can try to choose the signature that produces the best possible output. Action (i) is equivalent to aborting the protocol, and can be easily handled by the simulator, since we achieve security with aborts. Action (ii) is tackled by the use effect of the of bid: regardless of A using the same or another public key, the outcome of the protocol in different branches are always independent. Finally, as for action (iii), note that this attack is prevented by the uniqueness property of the signature scheme that ensures that for every (possibly malicious) public key pk and every input x, there exists at most one valid signature y that is not rejected by the verification algorithm.

Proof of Theorem 1. We begin by describing the simulator. Upon input the set of corrupted parties $\mathcal{I}$, and auxiliary input z, the simulator S proceeds as follows:[11]

1. Initialize an empty array $\mathcal{L}$. Sample a random value $sid \leftarrow\!\!\$ \{0,1\}^\lambda$, and run $\mathsf{Broadcast}(sid)$.
2. Generate $(pk_i, sk_i) \leftarrow\!\!\$ \mathsf{Gen}(1^\lambda)$ and run $\mathsf{Broadcast}(pk_i)$ for each $i \notin \mathcal{I}$.
3. Keep running $\mathsf{GetRecords}(\mathsf{UpdateState}(1^\lambda))$ until all public keys $pk_1, \ldots, pk_n$ are published on the blockchain; when that happens define bid to be the hash of the blockchain up to the block containing the last public key.
4. If $\mathcal{L}$ does not already contain the value bid, query the ideal functionality $f_\perp^{\mathsf{pct}}$, obtaining a random value ω, and store (bid, ω) in $\mathcal{L}$. Then complete the simulation for branch bid as follows:
   (a) For each $i \notin \mathcal{I}$, let $y_i = \mathsf{Sign}(sk_i, x)$ where $x = pk_1||\cdots||pk_n||sid||bid$. Then run $\mathsf{Broadcast}(y_i)$.
   (b) Keep running $\mathsf{GetRecords}(\mathsf{UpdateState}(1^\lambda))$ until all values $y_1, \ldots, y_n$ are published on the blockchain; when that happens check that $\mathsf{Verify}(pk_i, x, y_i) = 1$ for all $i \in [n]$. If any of these checks fails, send `abort` to $f_\perp^{\mathsf{pct}}$, simulate A aborting, and terminate. Else, in case $\mathsf{Hash}(y_1||\cdots||y_n)$ was already set to $\omega' \neq \omega$, abort the simulation and terminate. In this last case, we say that the simulator fails to program the random oracle.
5. Upon input a random oracle query q from A, answer as follows:
   (a) If there exists a pair $(bid, \omega) \in \mathcal{L}$ for which it is possible to parse $q := y_1||\cdots||y_n$ so that $\mathsf{Verify}(pk_i, x, y_i) = 1$ for all $i \in [n]$—where $x =$ $pk_1||\cdots||pk_n||sid||bid$ for the values $sid, (pk_1, \ldots, pk_n)$ that appear in the simulation of branch bid—program $\mathsf{Hash}(q) := \omega$ and answer query q with ω. If ω was already given as output for a different query then we say that the simulator fails creating a collision when programming the random oracle.
   (b) Else, return a random value (maintaining consistency among repeated queries).

Notice that S simulates perfectly the messages written on the blockchain by the honest players in a real execution of π^*_{pct}, including their interaction with the blockchain protocol Γ. Moreover, in each branch bid, the simulator perfectly emulates an abort of the protocol due to the fact that A sends invalid signatures. Hence, the only difference between the real and ideal experiment is that in the latter, for each branch bid, the simulator forces the protocol output to be a fresh random value received from the ideal coin-tossing functionality. Consider the following events:

Event $\mathbf{BAD}_1$: The event becomes true in case the simulator fails to program the random oracle in step 4b of the simulation.

Event $\mathbf{BAD}_2$: The event becomes true in case, the simulator fails creating a collision as described in step 5a. This is possible when for a given branch bid, there exists an index $i \in \mathcal{I}$ such that the attacker produces two outputs y_i and y_i' such that $y_i \neq y_i'$ and $\mathsf{Verify}(pk_i, x, y_i) = \mathsf{Verify}(pk_i, x, y_i') = 1$, and queries the random oracle first using y_i' therefore obtaining ω, and then querying y_i therefore obtain again ω.

Event $\mathbf{BAD}_3$: The event becomes true in case there exist two branches with different public keys, but for which the value bid is the same.

[11] For simplicity, we assume that the player P_1 initiating the protocol is honest; if not, it is easy to adapt the simulation by having S using the value sid written by A on the blockchain.

Let $\mathbf{BAD} = \mathbf{BAD}_1 \cup \mathbf{BAD}_2 \cup \mathbf{BAD}_3$. We claim that conditioning on $\overline{\mathbf{BAD}}$, the experiments $\mathbf{REAL}^{\Gamma,0}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ and $\mathbf{IDEAL}^{\Gamma}_{f^{\mathsf{pct}}_{\perp},\mathsf{S},\mathsf{D}}(\lambda)$ are identical. This is because conditioning on $\mathbf{BAD}_1$ not happening, the attacker never queries the random oracle on $pk_1||\cdots||pk_n||sid||bid$ before the protocol execution on branch bid terminates, and thus the final output ω on that branch is uniformly distributed from the point of view of A (as it happens to be in the ideal world). Furthermore, conditioning on $\mathbf{BAD}_2$ and $\mathbf{BAD}_3$ not happening, the simulator correctly assigns a single random value ω to each branch identified by bid.

Next, we show that each of the above events happens at most with negligible probability. By a standard argument, this concludes the proof as the computational distance between $\mathbf{REAL}^{\Gamma,0}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ and $\mathbf{IDEAL}^{\Gamma}_{f^{\mathsf{pct}}_{\perp},\mathsf{S},\mathsf{D}}(\lambda)$ is at most equal to the probability of event $\mathbf{BAD}$.

Lemma 1. *For all PPT* A*, there exists a negligible function* $\nu_1(\cdot)$ *such that* $\mathbb{P}[\mathbf{BAD}_1] \leq \nu_1(\lambda)$.

Proof. Notice that event $\mathbf{BAD}_1$ happens if and only if there exists a protocol execution corresponding to a branch bid for which the attacker A queries the random oracle upon input $y_1||\cdots||y_n$ before these values appear on the blockchain. Intuitively, this requires that A forges a signature for one of the public keys corresponding to one of the honest players, and thus $\mathbb{P}[\mathbf{BAD}_1]$ must be negligible. The reduction is straightforward: Given a PPT attacker provoking event $\mathbf{BAD}_1$ with non-negligible probability, we can construct a PPT attacker A' as follows. Initially, A' tries to guess the index $i \notin \mathcal{I}$ and the branch index $j \in \mathtt{poly}(\lambda)$ corresponding to the protocol execution in which A will provoke event $\mathbf{BAD}_1$. Hence, A' simulates the execution of protocol π^*_{pct} with A as done in the real experiment, except that on the j-th branch it sets pk_i to be the public key pk^* received from the challenger.

Finally, A' waits that A makes a random oracle query $y_1||\cdots||y_n$ such that $\mathsf{Verify}(pk_i, x, y_i) = 1$, where $x = pk_1||\cdots||pk_n||sid||bid$; if the latter does not happen, A' aborts, else it forwards (x, y_i) to the challenger. The proof follows by observing that, with non-negligible probability, A' does not abort, and thus it breaks unforgeability with non-negligible probability. □

Lemma 2. *For all PPT* A*, there exists a negligible function* $\nu_2(\cdot)$ *such that* $\mathbb{P}[\mathbf{BAD}_2] \leq \nu_2(\lambda)$.

Proof. Notice that event $\mathbf{BAD}_2$ directly contradicts uniqueness of the signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$. Hence, $\mathbb{P}[\mathbf{BAD}_2]$ must be negligible. The reduction is straightforward: Given a PPT attacker provoking event $\mathbf{BAD}_2$ with non-negligible probability, we can construct a PPT attacker A' as follows. Initially, A' tries to guess the index $i \in \mathcal{I}$ and the branch index $j \in \mathtt{poly}(\lambda)$ corresponding to the protocol execution in which A will provoke event $\mathbf{BAD}_2$. Hence, A' simulates the execution of protocol π^*_{pct} with A as done in the real experiment.

Finally, A' waits that A publishes on the j-th branch the two values y_i, y_i' which make the event $\mathbf{BAD}_2$ become true; if the latter does not happen, A' aborts, else it forwards (pk_i, y_i, y_i') to the challenger where the public key pk_i is the public key corresponding to the i-th player on the j-th branch. The proof follows by observing that, with non-negligible probability, A' does not abort, and thus it breaks uniqueness with non-negligible probability. □

Lemma 3. *For all PPT* A*, there exists a negligible function* $\nu_3(\cdot)$ *such that* $\mathbb{P}[\mathbf{BAD}_3] \leq \nu_3(\lambda)$.

Proof. Notice that event $\mathbf{BAD}_3$ directly contradicts collision resistance of the hash function Hash. Hence, $\mathbb{P}[\mathbf{BAD}_3]$ must be negligible. The reduction is straightforward: Given a PPT attacker provoking event $\mathbf{BAD}_3$ with non-negligible probability, we can construct a PPT attacker A' that simply emulates a protocol execution with A as in the real experiment. The values bid, as well as the answers to A's random oracle queries, are obtained by querying the target random oracle. Hence, whenever A' finds a fork with two different branches $\mathcal{B}$ and $\mathcal{B}'$

such that $\mathsf{Hash}(\mathcal{B}) = \mathsf{Hash}(\mathcal{B}')$, it outputs $(\mathcal{B}, \mathcal{B}')$ and stops. Since A provokes event $\mathbf{BAD}_3$ with non-negligible probability, A' wins with the same probability. This concludes the proof. $\square$

Putting the above lemmas together, by a union bound, there exists a negligible function $\nu(\cdot)$ such that $\mathbb{P}[\mathbf{BAD}] \leq \nu(\lambda)$, as desired. $\square$

4.5 Fairness with Penalties

In their work, Andrychowicz et al. [ADMM14, ADMM16] proposed a different notion of fairness for MPC protocols that run on blockchain: fairness with penalties. This notion states that if an adversary in an MPC protocol decides to abort the execution of the protocol it will be financially penalized. To obtain the penalization in the lottery protocol, Andrychowicz et al. added the deposit step in the protocol.

We now discuss how to augment the protocol π^*_{pct} in order to achieve fairness with penalties. First of all, each party should publish also a deposit along with her public key on the blockchain. The deposit can be redeemed by showing a valid signature on the value $x = pk_1||\cdots||pk_n||sid||bid$.

Assume that P_n is corrupted. The adversary can wait that the honest parties publish their value $y_1, \ldots, y_{n-1}$ on a given branch, and thus locally compute the output $\mathsf{Hash}(y_1||\cdots||y_n)$, where y_n is P_n's signatures on x corresponding to public key pk_n. Now, if P_n does not like the output it can either: (i) publish y_n in any case , or (ii) decide not to publish y_n. In case (i), P_n plays honestly, takes back his deposit and every player obtains the output. In case (ii), P_n aborts the protocol, but loses her deposit.

Note that the penalty mechanism for our protocol is too sophisticated for the scripting language used in Bitcoin. Instead in Ethereum we can design a smart contract to define the PCT protocol, having fairness with penalties and without penalizing the efficiency.

In App. A we give details about how the smart contract works.

We call $\tilde{\pi}^*_{\mathsf{pct}}$ the fair (with penalties) version of protocol π^*_{pct}. The informal description of the smart contract used in $\tilde{\pi}^*_{\mathsf{pct}}$ is given in Fig. 2 and the protocol is described below:

(i) *Setup phase:* At the beginning, one of the players creates the smart contract. When the contract is posted on the blockchain, the constructor automatically generates a unique session identifier sid.

(ii) *Deposit phase:* For each $i \in [n]$, P_i can decide to participate to the PCT protocol by triggering the function `deposit` to send a safety deposit and his public key pk_i of an unique signature scheme. After `time1` blocks have passed, if $(pk_1, \ldots, pk_n)$ are collected by the smart contract, it computes bid as $\mathsf{Hash}(\mathcal{B})$, where $\mathcal{B}$ is the blockchain that contains $(pk_1, \ldots, pk_n)$. The deposit phase ends and parties can start to redeem their deposit.

(iii) *Claim phase:* For each $i \in [n]$, P_i can claim his deposit back by triggering the function `claim` of the smart contract and sending a value y_i such that $\mathsf{Verify}(pk_i, x, y_i) = 1$, where $x = pk_1||\cdots||pk_n||sid||bid$, and pk_i is the public key of P_i. After `time2` blocks have passed, the claim phase ends and the smart contract computes and publishes the output as $\mathsf{Hash}(y_1||\cdots||y_n)$.

Theorem 2. *If* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is a unique signature scheme,* $\tilde{\pi}^*_{\mathsf{pct}}$ *described in Fig. 2 securely realizes* f^{pct} *and satisfies fairness with penalties in the presence of hasty players and malicious adversaries, in the programmable RO model.*

Proof sketch. The privacy of the protocol is proven by following the same outline of Theorem. 1. We have to prove that $\tilde{\pi}^*_{\mathsf{pct}}$ is fair (with penalties).

The **Parallel Coin Tossing Smart Contract** runs with players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ and consists of two main functions `deposit` and `claim` and two fixed timestamps `time1`,`time2` and a session id sid.

Deposit Phase: In round t_1, when `deposit`(pk_i) together with d coins is triggered from a party P_i, store (i, pk_i). Then, if $(pk_1, \ldots, pk_n)$ are stored, compute and store $bid := \mathsf{Hash}(\mathcal{B})$ and proceed to the Claim Phase. Otherwise, for all i, if the message (i, pk_i) has been stored, send back d coins to P_i and terminate.

Claim Phase: In round t_2, when `claim`(i, y_i) is triggered from P_i, check if $\mathsf{Verify}(pk_i, x, y_i) = 1$, where $x = pk_1||\cdots||pk_n||sid||bid$. If the check is correct send d coins back to P_i.

Compute Phase: If, after `time2`, all the y_i are correctly claimed, compute and publish $\mathsf{Hash}(y_1, \ldots, y_n)$.

Figure 2: Smart contract for parallel coin tossing.

There are four possible scenarios that can happen and in each of them either all parties learn the output or the adversary A loses her deposit. An output out of $\tilde{\pi}^*_{\mathsf{pct}}$ is considered valid if it is confirmed on the blockchain. The four scenarios are described below.

Scenario 1: A does not exploit branches to play different public keys on different execution of $\tilde{\pi}^*_{\mathsf{pct}}$. Fairness follows from the smart contract execution (i.e., if A does not provide the signature of x, A will lose her deposit).

Scenario 2: There is the following time-line: there is a fork with two branches b_1 and b_2 and the Setup Phase is published before the fork, but the Deposit Phase is executed after the fork. A aborts (i.e., A does not provide the signature of x) the execution of $\tilde{\pi}^*_{\mathsf{pct}}$ in b_1 and exploits b_2 in the following way: in b_2 a corrupted player P_i double spends (for any kind of transaction) the coins deposited in b_1. In this case, either b_1 gets confirmed, thus, boiling down to Scenario 1, or b_2 get confirmed and all the transactions sent to the smart contract of $\tilde{\pi}^*_{\mathsf{pct}}$ in branch b_1 are not valid b_2 since the deposit of P_i is previously spent in another transaction. It guarantees fairness since in b_1 the adversary is punished, and in b_2 there's no execution, and so no valid output

Scenario 3: This scenario follows the same time-line of Scenario 2. A aborts (i.e., A does not provide the signature of x) the execution of $\tilde{\pi}^*_{\mathsf{pct}}$ in b_1 and exploits b_2 in the following way: a corrupted player P_i publishes a different public key pk_i in b_2 (wlog., we analyze the case with two executions but it can be extended to multiple executions). In this case, the execution of $\tilde{\pi}^*_{\mathsf{pct}}$ is restarted from the beginning of the Deposit Phase in b_2 and the output that A learned on b_1 is not valid anymore. If b_1 gets confirmed it boils down to Scenario 1, otherwise the honest players learn the valid output computed in b_2. It guarantees fairness because in b_1 the adversary is punished, while in b_2 all parties will learn the output.

Scenario 4: There is the following time-line: there is a fork with two branches b_1 and b_2 and the Deposit Phase is published before the fork, but the Claim Phase is executed after the fork. A corrupted party P_i sends $y_i = \mathsf{Sign}(sk_i, x)$ in the Claim Phase in b_1 and disliking the output out of the execution of $\tilde{\pi}^*_{\mathsf{pct}}$ in b_1, P_i exploits b_2 to send $y_i' \neq y_i$. We note that if y_i' is a valid signature for x under secret key sk_i we can create an adversary A' breaking the uniqueness of the signature scheme. It means that y_i' cannot be a valid signature. If b_1 gets confirmed P_i is not penalized and every player will learn out, otherwise if b_2 gets confirmed P_i is penalized and no party will learn the output.

□

A remark on DoS attacks. Note that in $\tilde{\pi}^*_{\mathsf{pct}}$ there is no need to fix the identity of the participants "before" the execution of the protocol. We can consider the case in which a party P_i participates to a protocol execution only after she triggers the `deposit` function giving as an input her public key. Our PCT protocol can be executed even if only two parties decide to participate and thus n does not need to be fixed beforehand. Moreover, `time1` is independent of n and of the number of blocks to wait for considering a transaction confirmed. Registered parties are not incentivized to abort the protocol (i.e., by not triggering the `claim` function) due to financial compensation (parties must send a collateral deposit together with the first message). This makes DoS attacks in which the attacker aborts the protocol multiple times (making honest parties waste time and money) financially inconvenient.

4.6 Experimental Evaluation

We also provide some experiments to show noticeable improvements in our PCT protocol with respect to the lottery protocol of Andrychowicz et al. in terms of the number of blocks needed for completion of the protocol. Since the confirmation time has a considerable impact on the overall communication with respect to the number of rounds, we measure the efficiency of on-chain protocols in terms of the number of blocks[12].

We evaluate the efficiency of $\tilde{\pi}^*_{\mathsf{pct}}$ compared with the protocol from [ADMM14, ADMM16]. To evaluate the efficiency in the best case we consider the following assumptions:

- Transactions in the last k blocks are considered not confirmed yet.
- All parties send the message at round i of the protocol as soon as they read all messages from round $i-1$ on $\mathcal{B}_i^{\lceil k}$, where k is 0 in case of hasty executions.
- Whenever a player broadcasts a transaction, it appears in the next block.
- All messages in a round of the MPC protocol fit in a single block.

In case of non-hasty executions if we have a ρ-round MPC protocol π running on the blockchain, the number of blocks needed to complete the execution with the previous assumption is $\rho \cdot k$.

Analysis. We now give a comparison between our coin-tossing protocol and the one of Andrychowicz et al. To allow for a fair comparison between our coin-tossing protocol and the one presented in [ADMM14, ADMM16], we implemented both protocols in Solidity using Ethereum smart contracts.[13] See Fig. 6 and Fig. 7 for the code of the smart contracts.

For both protocols, in the deposit phase, a timeout $\bar{t}$ must be provided by the contract creator, so that players have enough time to send their deposits together with the corresponding additional information required by the protocol. In the ideal conditions described above, the timeout can be of just one block. The same argument applies to the opening phase of [ADMM14, ADMM16]. A comparison is described below:

- *Lottery:* Due to the expressiveness of smart contracts, our implementation of [ADMM14, ADMM16] requires one step less than the original implementation using in bitcoin. Specifically, we can embed the betting phase in the commit/deposit phase, by just requiring that the players deposit 1 more coin. Since in their setting block confirmation is required at each step, the overall execution takes exactly $3 \cdot k$ blocks (including one round for posting the smart contract).
- *Our PCT:* As proven in Sec. 4.4, our PCT protocol can be executed in fully-hasty mode. The entire execution consists of $2 + k$ blocks (including 1 block for posting the contract

[12]Notice that whenever players are all online and ready to play, the execution should be fast and waiting for confirmations of all messages would be painful.

[13]Notice that the average time for a new block to appear is around 15 seconds[Eth].

and confirmation of the output). In the worst case, where all messages will appear to the state of the honest player after k blocks for each step, the overall execution takes $3 \cdot k$, as much as [ADMM16].

GAS consumption. As it can be seen in Fig. 3, PCT is more expensive in terms of GAS consumption than Lottery. It is well motivated by the fact that Lottery uses only hash function to compute the commitments and no other expensive computations. Our PCT protocol needs also unique signatures. Our GAS calculation for unique signature is based on the BLS signatures implementation provided for testing in [Sol], but improved implementations could potentially lower the GAS consumption. It can be seen anyway as an affordable cost to pay to achieve efficiency still maintaining the same security guarantees.

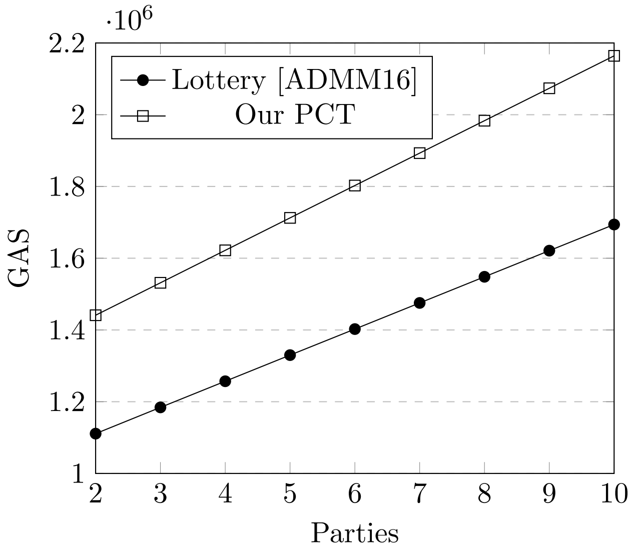

Figure 3: GAS consumption comparison between our smart contract implementation of PCT (Sec. 4.3) protocol and the Lottery of Andrychowicz et al. [ADMM16] (Sec. 4.1).

5 Our Generic Compiler

Our compiler starts from the observation that a stand-alone MPC protocol could be insecure when executed on a blockchain. To be concrete, a rewinding simulator of the MPC protocol can not be used to prove the security of the on-chain MPC protocol, since rewinding would have the unclear meaning of rewinding the blockchain. Moreover, we do not want to give control of the blockchain to the simulator (i.e., no control of the majority of the stake, of the computational power, and so on) since our result aims at being generic w.r.t. the type of blockchain used. Essentially, the simulator is going to incarnate just the honest players of the MPC protocol during the simulation. In order to perform a simulation in the presence of a concurrently played blockchain protocol, (i.e., rewinding is not possible and the blockchain is generic and therefore not controlled by the simulator), we therefore require the initial protocol π received in input by the compiler to be universally composable secure. This guarantees the existence of a straight-line simulator and allows us to avoid simulators that "control" the blockchain[14], therefore allowing the applicability of our results to generic blockchains. Additionally, we require

[14]Typically a simulator that controls the blockchain requires some specific assumptions on the blockchain like in [GG17] where only some restricted proof-of-stake blockchains were compatible with the simulation.

π to have only "digital" communication since players when running the protocol on-chain must produce messages that consists of bits only. Therefore an exchange of hardware (e.g., PUFs) in π cannot be accepted.

Notice that the original protocol might require private and authenticated channels. Since the entire traffic of our protocol will be redirected to the blockchain, we will use public-key encryption and digital signatures. The first message of each player in the compiled protocol will consist of a pair of public keys, one to receive encrypted messages and one to allow others to verify signatures of messages.

In this section, we propose and analyze a simple transformation that allows to run any MPC protocol safely on the blockchain, even when the players are hasty. In Sec. 5.1 we describe our compiler, while in Sec. 5.2 we analyze its security. Finally, in Sec. 5.3, we discuss how to extend our generic transformation in order to achieve fairness with penalties, as long as the players start being hasty after the confirmation of the first round.

5.1 Compiler Description

Intuitively our transformation proceeds as follows. Our starting point is any MPC protocol π UC-securely computing an n-party functionality $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ in the presence of malicious players (with aborts). Hence, the honest players fix the random tape for running π and simply execute protocol π by broadcasting their messages on the blockchain. Furthermore, each honest player P_i keeps track of the longest protocol transcript α_i generated so far and, in the presence of a fork, aborts the execution in case the view on a given branch is not consistent with α_i. This intuitively ensures that the underlying protocol π is run only once, even in the presence of forks.

Since the initial protocol π may require private channels between the players, we need to augment the above transformation in such a way that subsets of honest parties can exchange messages in a confidential and authenticated manner. Let $m_{i,j}^{(r)}$ be the message that P_i sends to P_j at the generic round r. The latter is achieved by having P_i encrypting $m_{i,j}^{(r)}$ using the public encryption key ek_j of P_j, and then signing the resulting ciphertext $c_{i,j}^{(r)}$ with its own private signing key sk_i, which is the standard way of building a secure channel. A formal description of the compiler is given in Fig. 4. Note that wlog. we assume that for each round of the underlying protocol π every P_i sends a single message to each $\mathsf{P}_{j\neq i}$ over a private and authenticated channel. Moreover, P_i picks a sufficiently long random tape ω_i that is then used to run π over Γ. Observe that ω_i includes both the randomness required to compute the messages in π and the random coins used to encrypt them. In particular, in the presence of forks, an honest P_i that does not abort broadcasts on the blockchain exactly the same ciphertexts on multiple branches.

5.2 Security Analysis

The theorem below establishes the security of our generic compiler.

Theorem 3. *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a semantically secure PKE scheme, and* $(\mathsf{Gen}', \mathsf{Sign}, \mathsf{Verify})$ *be a (deterministic) unforgeable signature scheme. Furthermore, let* π *be an* n*-party* ρ*-round protocol that* t*-securely UC-realizes a functionality* f *with aborts in the presence of malicious adversaries. Then, the protocol* π^* *of Fig. 4* t*-securely computes* f *with aborts in the presence of hasty players and malicious adversaries.*

UC security is needed due to the fact that the attacker in the real world may interact with the blockchain by posting messages and reading its state. As shown in [CGJ19], such blockchain-active adversaries render standard simulation techniques (e.g., black-box rewinding)

Generic Compiler π^*

Let π be an n-party ρ-round protocol, and $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ be a blockchain protocol. Further, let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a PKE scheme and $(\mathsf{Gen}', \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme, both with domain $\{0,1\}^*$. The protocol π^* proceeds as follows:

- For $i \in [n]$, each player P_i initializes $st_i := \varepsilon$, samples $(ek_i, dk_i) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda)$, $(vk_i, sk_i) \leftarrow\!\$\ \mathsf{Gen}'(1^\lambda)$, and $\omega_i \leftarrow\!\$\ \{0,1\}^*$, and invokes $\mathsf{Broadcast}(ek_i||vk_i||i)$.
- For $i \in [n]$, each player P_i keeps running $st_i \leftarrow\!\$\ \mathsf{UpdateState}(1^\lambda)$ and $\mathcal{B}_i \leftarrow\!\$\ \mathsf{GetRecords}(st_i)$ until all the messages $(ek_j, vk_j)_{j\in[n]} \in \mathcal{B}_i$.
- For $i \in [n]$, each player P_i sets $\tau^{(0)} := (ek_j, vk_j)_{j\in[n]}$ and $\alpha_i := \tau^{(0)}$, and then runs the following loop:
  1. Update the state st_i by running $\mathsf{UpdateState}(1^\lambda)$, and let $\mathcal{B}_i \leftarrow\!\$\ \mathsf{GetRecords}(st_i)$.
  2. Let $\tilde{r} \geq 0$ be the maximum value such that the partial transcript $\tau^{(\tilde{r})} \in \mathcal{B}_i$. Then:
     - If the ciphertexts in $\tau^{(\tilde{r})}$ are not consistent with those in α_i, output $\perp$ and terminate.
     - Else if $\tilde{r} = \rho$, output the value y_i as a function of $\tau^{(\rho)}$ and terminate.
     - Else, go to the next step and if α_i is a prefix of $\tau^{(\tilde{r})}$ let $\alpha_i := \tau^{(\tilde{r})}$.
  3. For each $j \in [n]$, with $j \neq i$, and for each $r \leq \tilde{r}$, decrypt the ciphertexts $c_{j,i}^{(r)}$ and use the corresponding values $m_{j,i}^{(r)}$ to compute the messages $m_{i,j}^{(\tilde{r}+1)}$ to be sent at round $\tilde{r}+1$ (using the corresponding portion of the random tape ω_i).
  4. Finally, let $c_{i,j}^{(\tilde{r}+1)} \leftarrow\!\$\ \mathsf{Enc}(ek_j, m_{i,j}^{(\tilde{r}+1)})$ (using again random coins coming from ω_i) and $\sigma_{i,j}^{(\tilde{r}+1)} = \mathsf{Sign}(sk_i, c_{i,j}^{(\tilde{r}+1)})$, and invoke $\mathsf{Broadcast}((c_{i,j}^{(\tilde{r}+1)}||\sigma_{i,j}^{(\tilde{r}+1)})_{j\in[n]\setminus\{i\}})$.

Figure 4: Generic compiler for obtaining blockchain-aided MPC with hasty players.

moot. Note also that Remark 2 does not hold for our protocol. If the adversary tries to furnish two different inputs in two different branches it can be spotted by some honest player, leading to an abort. Therefore only one possible input can be given to the functionality.

We need to show that for every PPT adversary A^*, there exists a PPT simulator S^* such that no non-uniform PPT distinguisher D^* can tell apart the experiments $\mathbf{REAL}^{\Gamma,0}_{\pi,\mathsf{A}^*,\mathsf{D}^*}(\lambda)$ and $\mathbf{IDEAL}^{\Gamma}_{f_\perp,\mathsf{S}^*,\mathsf{D}^*}(\lambda)$. In particular, the simulator S^* needs to simulate the interaction of the honest players with the blockchain protocol Γ as it happens in the real experiment. Intuitively, S^* relies on the simulator S guaranteed by the underlying protocol π as follows. At the beginning, S^* samples the public/secret keys for encryption/signatures for the honest players. Then, S^* runs A^* reading its messages from the emulated execution of the blockchain protocol Γ, and simulates its view as follows: (i) The round-r messages $m_{j,i}^{(r)}$ sent by the honest players P_j to the malicious players P_i are obtained from the simulator S; (ii) The round-r messages $m_{j,j'}^{(r)}$ that are exchanged by the honest players $\mathsf{P}_j, \mathsf{P}_{j'}$ are replaced with the all-zero string. Of course, S^* does additional bookkeeping in order to simulate a real execution of the protocol using the blockchain; in particular, S^* needs to check that the attacker plays consistently on different branches of a fork, and simulate an abort whenever the latter does not happen. Moreover, when S extracts the inputs for the malicious parties, the simulator S^* forwards the same inputs to the trusted party, obtains the outputs for the malicious parties, and sends it to S. Finally, S^* completes the simulation consistently with the choice of S of aborting or not.

Very roughly, the security of the PKE scheme and of the signature scheme imply that the view of the attacker is identical to that in a real execution of protocol π, so that security of π^* follows by that of π.

Proof. We begin by describing the simulator S^*. Let S be the PPT simulator guaranteed by the malicious security of π. Upon input the set of corrupted parties $\mathcal{I}$, inputs $(x_i)_{i\in\mathcal{I}}$, and auxiliary input z, the simulator S^* proceeds as follows:

1. Initialize S upon input $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$, with uniformly chosen random tape $\omega_{\mathsf{sim}} \leftarrow\!\$\ \{0,1\}^*$.

2. For each $j \notin \mathcal{I}$, sample $(ek_j, dk_j) \leftarrow\$ \mathsf{Gen}(1^\lambda)$, $(vk_j, sk_j) \leftarrow\$ \mathsf{Gen}'(1^\lambda)$, $\omega_j \leftarrow\$ \{0,1\}^*$, and invoke $\mathsf{Broadcast}(ek_j||vk_j||j)$.
3. For each $j \notin \mathcal{I}$, keep running $st_j \leftarrow\$ \mathsf{UpdateState}(1^\lambda)$ and $\mathcal{B}_j \leftarrow\$ \mathsf{GetRecords}(st_j)$ until all the messages $(ek_i, vk_i)_{i\in[n]} \in \mathcal{B}_j$. Set $\tau_j^{(0)} := (ek_i, vk_i)_{i\in[n]}$ and $\alpha_j := \tau^{(0)}$.
4. For each $j \notin \mathcal{I}$, emulate the behavior of party P_j as follows:
   (a) Update the state st_j by running $\mathsf{UpdateState}(1^\lambda)$, and let $\mathcal{B}_j \leftarrow\$ \mathsf{GetRecords}(st_j)$.
   (b) Let $\tilde{r} \geq 0$ be the maximum value such that the partial transcript $\tau^{(\tilde{r})} \in \mathcal{B}_j$. Then:
      - If the ciphertexts in $\tau^{(\tilde{r})}$ are not consistent with those in α_j, send `abort` to the trusted party, simulate A^* aborting in the real protocol, and terminate.
      - Else, go to the next step and if α_j is a prefix of $\tau^{(\tilde{r})}$ let $\alpha_j := \tau^{(\tilde{r})}$.
   (c) Extract from $\tau_j^{(\tilde{r})}$ the ciphertexts $(c_{i,j}^{(\tilde{r})})_{i\in\mathcal{I}}$ and the signatures $(\sigma_{i,j}^{(\tilde{r})})_{i\in\mathcal{I}}$ that A^* (on behalf of each corrupted player P_i) forwards to P_j. If there exists $i \in \mathcal{I}$ such that $\mathsf{Verify}(vk_i, \sigma_{i,j}^{(\tilde{r})}) = 0$, send `abort` to the trusted party, simulate A^* aborting in the real protocol, and terminate. Else, for each $r \leq \tilde{r}$, decrypt the ciphertexts $c_{i,j}^{(r)}$ using the decryption key dk_j, and pass the corresponding messages $((m_{i,j}^{(1)})_{i\in\mathcal{I},j\in\mathcal{H}}, \ldots, (m_{i,j}^{(\tilde{r})})_{i\in\mathcal{I},j\in\mathcal{H}})$ to S. Hence:
      - Upon receiving `abort` from S, send `abort` to the trusted party, simulate A^* aborting in the real protocol, and terminate.
      - Upon receiving $(x_i)_{i\in\mathcal{I}}$ from S, send $(x_i)_{i\in\mathcal{I}}$ to the trusted party, obtain the outputs $(y_i)_{i\in\mathcal{I}}$, and forward $(y_i)_{i\in\mathcal{I}}$ to S. In case S replies with $(\mathtt{continue}, \mathcal{H}')$, send $(\mathtt{continue}, \mathcal{H}')$ to the trusted party and terminate.
      - Upon receiving a set of messages $(m_{j,i}^{(\tilde{r}+1)})_{j\in\mathcal{H},i\in\mathcal{I}}$—corresponding to the simulated messages that each honest player P_j sends to the corrupted party P_i— for each $j \in \mathcal{H}$ and $i \in \mathcal{I}$ compute $c_{j,i}^{(\tilde{r}+1)} \leftarrow\$ \mathsf{Enc}(ek_i, , m_{j,i}^{(\tilde{r}+1)})$ (using coins from ω_j) and $\sigma_{j,i}^{(\tilde{r}+1)} = \mathsf{Sign}(sk_j, c_{j,i}^{(\tilde{r}+1)})$. Then, for each $j, j' \in \mathcal{H}$, let $c_{j,j'}^{(\tilde{r}+1)} \leftarrow\$ \mathsf{Enc}(ek_{j'}, 0^{|m_{j,j'}^{(\tilde{r}+1)}|})$ (using coins from ω_j) and $\sigma_{j,j'}^{(\tilde{r}+1)} \leftarrow\$ \mathsf{Sign}(sk_j, c_{j,j'}^{(\tilde{r}+1)})$, and finally invoke $\mathsf{Broadcast}((c_{j,i}^{(\tilde{r}+1)}||\sigma_{j,i}^{(\tilde{r}+1)})_{i\in[n]\setminus\{j\}})$.

To conclude the proof, we consider a sequence of hybrid experiments (ending with the real experiment) and argue that each pair of hybrids is computationally close thanks to the properties of the underlying cryptographic primitives.

Hybrid $\mathbf{H}_3(\lambda)$: This experiment is identical to $\mathbf{IDEAL}^{\Gamma}_{f_\perp,\mathsf{S}^*,\mathsf{D}^*}(\lambda)$.

Hybrid $\mathbf{H}_2(\lambda)$: Identical to $\mathbf{H}_3(\lambda)$ except that we replace the ciphertexts $(c_{j,j'}^{(r)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ that each honest party P_j sends to the other honest players $\mathsf{P}_{j'}$ with an encryption of the real messages $(m_{j,j'}^{(r)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ that the same parties would send in a real execution of π. Note that the other ciphertexts $(c_{j,i}^{(r)})_{j\in\mathcal{H},i\in\mathcal{I}}$ are still emulated using the simulator, and the output of the experiment is determined by the trusted party.
The inputs for the honest parties are chosen to be the values $(x_i)_{i\in\mathcal{H}}$ chosen by the distinguisher D^* at the beginning of the experiment, and the random tape of each player is chosen uniformly once and for all as in the real world.

Hybrid $\mathbf{H}_1(\lambda)$: Identical to $\mathbf{H}_2(\lambda)$ except that we artificially abort if A^* modifies one of the ciphertexts $(c_{j,i}^{(r)})_{j\in\mathcal{H},i\in[n]\setminus\{j\}}$ corresponding to the messages that each honest player sends in a given round. Note that these ciphertexts correspond to both the real messages $(m_{j,j'}^{(r)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ and the simulated messages $(m_{j,i}^{(r)})_{j\in\mathcal{H},i\in\mathcal{I}}$.

Hybrid $\mathbf{H}_0(\lambda)$: This experiment is identical to $\mathbf{REAL}^{\Gamma,0}_{\pi^*,\mathsf{A}^*,\mathsf{D}^*}(\lambda)$.

Lemma 4. $\{\mathbf{H}_3(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{\mathbf{H}_2(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. We reduce to semantic security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Let $h = |\mathcal{H}|$. For $k \in [0, h]$, consider the hybrid experiment $\mathbf{H}_{3,k}(\lambda)$ in which the distribution of the ciphertexts $(c_{j,j'}^{(r+1)})_{j \in \mathcal{H}, j' \in \mathcal{H} \setminus \{j\}}$ is modified as in $\mathbf{H}_2(\lambda)$ only for the first h honest parties. Clearly, $\{\mathbf{H}_{3,0}(\lambda)\}_{\lambda \in \mathbb{N}} \equiv \{\mathbf{H}_3(\lambda)\}_{\lambda \in \mathbb{N}}$ and $\{\mathbf{H}_{3,h}(\lambda)\}_{\lambda \in \mathbb{N}} \equiv \{\mathbf{H}_2(\lambda)\}_{\lambda \in \mathbb{N}}$.

Next, we prove that for every $k \in [0, h]$ it holds that $\{\mathbf{H}_{3,k}(\lambda)\}_{\lambda \in \mathbb{N}} \approx_c \{\mathbf{H}_{3,k+1}(\lambda)\}_{\lambda \in \mathbb{N}}$ which concludes the proof of the lemma. By contradiction, assume that there exists an index $k \in [0, h]$, and a pair of PPT algorithms $(\mathsf{D}^*, \mathsf{A}^*)$ such that D^* can distinguish the two experiments $\mathbf{H}_{3,k}(\lambda)$ and $\mathbf{H}_{3,k+1}(\lambda)$ with non-negligible probability. We construct a PPT attacker B breaking semantic security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ as follows:

- Receive the target public encryption key ek^* from the challenger.
- Run D^*, receiving the set of corrupted parties $\mathcal{I}$, the inputs $(x_i)_{i \in [n]}$, and the auxiliary input z, then pass $(\mathcal{I}, (x_i)_{i \in \mathcal{I}}, z)$ to A^*.
- Interact with A^* as described in the ideal experiment, except that:
  - The public encryption key for player P_{k+1} is set to be the target public key ek^*.
  - For each $j \leq k$, when it comes to simulating the ciphertexts $(c_{j,j'}^{(r)})_{j' \in \mathcal{H} \setminus \{j\}}$, use the real messages $(m_{j,j'}^{(r)})_{j' \in \mathcal{H} \setminus \{j\}}$, encrypt them using the public encryption key $ek_{j'}$ of $\mathsf{P}_{j'}$, and sign the ciphertexts with the secret key sk_j (which is known to the reduction).
  - When it comes to simulating the ciphertexts $(c_{k+1,j'}^{(r)})_{j' \in \mathcal{H} \setminus \{j\}}$, forward the pair of plaintexts $(m_{k+1,j'}^{(r)}, 0^{|m_{k+1,j'}^{(r)}|})_{j' \in \mathcal{H} \setminus \{k+1\}}$ to the left-or-right encryption oracle and sign the corresponding ciphertexts using the secret signing key sk_{k+1} of P_{k+1} (which is known to the reduction).
  - For each $j > k + 1$, when it comes to simulating the ciphertexts $(c_{j,j'}^{(r)})_{j' \in \mathcal{H} \setminus \{j\}}$, use the dummy messages $(0^{|m_{j,j'}^{(r)}|})_{j' \in \mathcal{H} \setminus \{j\}}$, encrypt them using the public encryption key $ek_{j'}$ of $\mathsf{P}_{j'}$, and sign the ciphertexts with the secret key sk_j (which is known to the reduction).
- Finally, run D^* upon the final output generated by A^*, and return whatever D^* outputs.

Note that the reduction can indeed simulate the interaction with the blockchain protocol Γ as in the ideal experiment, and moreover it can generate the real messages $(m_{j,j'}^{r})_{j \in \mathcal{H}, j' \in \mathcal{H} \setminus \{j\}}$ as it knows the parties' inputs $(x_i)_{i \in [n]}$. By inspection, the simulation performed by B is perfect in the sense that when the challenger encrypts the messages $m_{k+1,j'}^{(r)}$ the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $\mathbf{H}_{3,k+1}(\lambda)$. Similarly, when the challenger encrypts the dummy messages $0^{|m_{k+1,j'}^{(r)}|}$ the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $\mathbf{H}_{3,k}(\lambda)$. Hence, B breaks semantic security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with non-negligible probability, concluding the proof. □

Lemma 5. $\{\mathbf{H}_2(\lambda)\}_{\lambda \in \mathbb{N}} \approx_c \{\mathbf{H}_1(\lambda)\}_{\lambda \in \mathbb{N}}$.

Proof. Let $\mathbf{BAD}$ be the event that an artificial abort happens in $\mathbf{H}_1(\lambda)$. Note that this means that, for some $j \in \mathcal{H}$, the attacker A^* replaces one of the ciphertexts $c_{j,i}^{(r)}$ that P_j would send to P_i in the real protocol with a different ciphertext $\tilde{c}_{j,i}^{(r)}$, in such a way that the corresponding signature $\tilde{\sigma}_{j,i}^{(r)}$ is still accepting. Clearly, the experiments $\mathbf{H}_2(\lambda)$ and $\mathbf{H}_1(\lambda)$ are identical conditioning on $\mathbf{BAD}$ not happening, and does it suffices to show that $\mathbb{P}[\mathbf{BAD}]$ is negligible.

Given a PPT distinguisher D^* and a PPT attacker A^* such that A^* provokes event $\mathbf{BAD}$ in a run of $\mathbf{H}_2(\lambda)$ with non-negligible probability, we can construct a PPT attacker B breaking security of the signature scheme $(\mathsf{Gen}', \mathsf{Sign}, \mathsf{Verify})$. The reduction works as follows:

- Receive the target public verification key vk^* from the challenger.

- Choose a random j^* as a guess for the index corresponding to the honest party for which A^* provokes the bad event.
- Run D^*, receiving the set of corrupted parties $\mathcal{I}$, the inputs $(x_i)_{i\in[n]}$, and the auxiliary input z, then pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$ to A^*.
- Interact with A^* as described in $\mathbf{H}_2(\lambda)$, except that:
  - The public verification key for player P_{j^*} is set to be the target public key vk^*.
  - When it comes to simulating the round-r messages from party P_{j^*}, generate the ciphertexts $(c^{(r)}_{j^*,i})_{i\in[n]\setminus\{j^*\}}$ as done in $\mathbf{H}_2(\lambda)$, and then forward each of $c^{(r)}_{j^*,i}$ to the challenger, obtaining the corresponding signature $\sigma^{(r)}_{j^*,i}$ that is needed in order to complete the simulation.
  - Keep updating the local state of P_{j^*} until an index $i \in [n]\setminus\{j^*\}$ is found such that the partial transcript α_{j^*} contains a pair $(\tilde{c}^{(r)}_{j^*,i}, \tilde{\sigma}^{(r)}_{j^*,i})$ such that $\mathsf{Verify}(vk_{j^*}, \tilde{c}^{(r)}_{j^*,i}, \tilde{\sigma}^{(r)}_{j^*,i}) = 1$ and $\tilde{c}^{(r)}_{j^*,i}$ is different from the original ciphertext $c^{(r)}_{j^*,i}$ previously sent on behalf of P_{j^*}.
- If no such pair is found, abort the simulation. Else, return $(\tilde{c}^{(r)}_{j^*,i}, \tilde{\sigma}^{(r)}_{j^*,i})$.

Note that the simulation performed by B is perfect, in that the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in a run of $\mathbf{H}_2(\lambda)$. Moreover, conditioning on B guessing the index j^* correctly, the reduction is successful in breaking security of the signature scheme exactly with probability at least $\mathbb{P}[\mathbf{BAD}]$, which is non-negligible. Since the former event also happens with non-negligible probability, this concludes the proof. $\square$

Lemma 6. $\{\mathbf{H}_1(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{\mathbf{H}_0(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. The proof is by reduction to UC-security of the underlying protocol π. By contradiction, assume that there exists a PPT adversary A^* and a non-uniform PPT distinguisher D^* that can distinguish between $\mathbf{H}_1(\lambda)$ and $\mathbf{H}_0(\lambda)$ with non-negligible probability. Consider the following PPT attacker A, initialized with a set of corrupted parties $\mathcal{I}$, inputs $(x_i)_{i\in\mathcal{I}}$ for the malicious players, and auxiliary input $z = (z^*, (ek_i, dk_i)_{i\in[n]}, (vk_i, sk_i)_{i\in[n]})$ which will be specified later:

- Pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z^*)$ to A^*.
- For each $i \in \mathcal{I}$, upon receiving the round-r messages $(m^{(r)}_{j,i})_{j\in\mathcal{H}}$ from the honest players to the malicious players, let $c^{(r)}_{j,i} \leftarrow\$\ \mathsf{Enc}(ek_i, m^{(r)}_{j,i})$ and $\sigma^{(r)}_{j,i} = \mathsf{Sign}(sk_j, c^{(r)}_{j,i})$, and emulate broadcasting $(c^{(r)}_{j,i}, \sigma^{(r)}_{j,i})_{j\in\mathcal{H},i\in\mathcal{I}}$ via the blockchain protocol.
- For each $j \in \mathcal{H}$, upon receiving the round-r messages $(m^{(r)}_{i,j})_{i\in\mathcal{I}}$ that A^* wants to send to the honest parties, let $c^{(r)}_{i,j} \leftarrow\$\ \mathsf{Enc}(ek_j, m^{(r)}_{i,j})$ and $\sigma^{(r)}_{i,j} = \mathsf{Sign}(sk_i, c^{(r)}_{i,j})$, and emulate broadcasting $(c^{(r)}_{i,j}, \sigma^{(r)}_{i,j})_{i\in\mathcal{I},j\in\mathcal{H}}$ via the blockchain protocol.
- For each $j \in \mathcal{H}$, compute the messages $(m^{(r)}_{j,j'})_{j'\in\mathcal{H}\setminus\{j\}}$ exchanged between honest parties as done in $\mathbf{H}_0$ (which is the same in $\mathbf{H}_1(\lambda)$), let $c^{(r)}_{j,j'} \leftarrow\$\ \mathsf{Enc}(ek_{j'}, m^{(r)}_{j,j'})$ and $\sigma^{(r)}_{j,j'} = \mathsf{Sign}(sk_j, c^{(r)}_{j,j'})$, and emulate broadcasting $(c^{(r)}_{j,j'}, \sigma^{(r)}_{j,j'})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ via the blockchain protocol.
- In case a fork appears during the simulation of the underlying blockchain protocol, replicate the messages from the honest players as done in the other branches (using exactly the same randomness). On the other hand, if the messages from A^* differ from those sent on the simulation of a previous branch, simulate A^* aborting and terminate.
- Output whatever A^* outputs.

Additionally, let Z be the following PPT distinguisher:

- Run D^*, receiving the set of corrupted parties $\mathcal{I}$, the inputs $(x_i)_{i\in[n]}$, and the auxiliary input z^*, then sample (ek_i, dk_i) and (vk_i, sk_i) for all $i \in [n]$, and pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$ to the above defined attacker A, where $z = (z^*, (ek_i, dk_i)_{i\in[n]}, (vk_i, sk_i)_{i\in[n]})$.

- Upon receiving the final output from A, pass it to D^* and output whatever D^* outputs.

By inspection, in case the attacker A is playing in a real execution of protocol π, the view of D^* is identical to that in an execution of $\mathbf{H}_0(\lambda)$ with A^* controlling the malicious parties. Similarly, in case the view of A is emulated using the simulator S (corrupting the dummy parties controlled by A) of protocol π, the view of D^* is identical to that in an execution of $\mathbf{H}_1(\lambda)$ with A^* controlling the malicious parties. It follows that Z can distinguish between $\mathbf{REAL}_{\pi,\mathsf{A},\mathsf{Z}}(\lambda)$ and $\mathbf{IDEAL}_{f_\perp,\mathsf{S},\mathsf{Z}}(\lambda)$ with non-negligible probability, a contradiction. $\square$

The theorem now follows directly by combining the above lemmas. $\square$

5.3 On Fairness with Penalties

Our compiler described in Sec. 5.1 does not directly achieve fairness with penalties but it is possible to obtain fairness with penalties following in part the outline of [KB14, BK14].

Let us now assume the existence of an (n, n)-secret sharing scheme $(\mathsf{Share}, \mathsf{Recon})$, non-interactive commitment schemes (see Sec. 2.2 for the formal definitions), and consider a functionality f' that first calculates $y \leftarrow f(x_1, \ldots, x_n)$, where each party P_i holds x_i, and then calculates the shares of the output $(\sigma_1, \ldots, \sigma_n) \leftarrow\!\!\$\ \mathsf{Share}(y)$, the commitments $C = (\gamma_1, \ldots, \gamma_n)$, where $\gamma_i \leftarrow\!\!\$\ \mathsf{Commit}(\sigma_i)$, and outputs (C, σ_i) to each player P_i. Let us call π' the protocol realizing f', we can apply our generic compiler to π' to obtain a protocol π'_{bc} that can be run in the blockchain. Our protocol π_{fair}, running with players $\mathsf{P}_1 \ldots, \mathsf{P}_n$ works as follows

(i) *Protocol execution*: All the players engage in the protocol π'_{bc}. A party P_i aborts the execution if π'_{bc} aborts. Otherwise, obtains (C, σ_i) in the last round.

(ii) *Smart contract*: P_1 publishes the smart contract depicted in Fig. 5.

(iii) *Commitment phase*: For each $i \in [n]$, P_i triggers `deposit`(C_i) together with d coins, where d is a fixed deposit. If some player does not publish his commitments with the deposit or there is a disagreement on the commitments within `time1` (i.e., a player P_j sends $C_j \neq C_i$ for some $P_{i \neq j}$, or deposits a value $d_i < d$, P_i abort the execution. Recall that abort in this phase is still fine, since no information about the output y is released. Otherwise, if `time1` has passed, go to the Opening Phase.

(iv) *Opening phase*: For each $i \in [n]$, P_i opens his commitment by sending `openCom`(i, σ_i), thus receiving back his d coins, wait that all the openings are published in the smart contract (within `time2`) and calculates $y \leftarrow \mathsf{Recon}(\sigma_1, \ldots, \sigma_n)$. If, after `time2`, some share is missing, P_i aborts the execution.

During the last phase, if some player did not open the commitment or sent an incorrect value, the smart contract will penalize him by freezing his deposit. Thus, the adversary is not incentivized to send an incorrect share.

This attempt to add fairness with penalties, however, introduces an attack. Given an n-party protocol $\pi^{\Gamma}_{f'}$ obtained by the compiler described in Sec. 5.1 applied to $\pi_{f'}$, with the addition of the smart contract, commit and opening phases described above, we have the following scenario:

- For all $i \in [n]$, party P_i runs π_{fair} obtaining (C, σ_i).
- For all $i \in [n]$, party P_i triggers `deposit`(C) together with d coins to the smart contract.
- For all $i \in [n-1]$, party P_i opens his commitment by triggering `openCom`(σ_i).
- Wlog., we say that P_n is an adversary. P_n compute the output y. If P_n does not like y in the current branch, P_n can try to exploit a fork happening during the execution of π_{fair} to change the in a different branch to obtain a new couple (C', σ'_n).
- The honest parties $\mathsf{P}_1, \ldots, \mathsf{P}_{n-1}$ notice that there is a message published by P_n that differs from the value previously received always by P_n. Since the transcript obtained from the blockchain differs from the transcript stored in their local state, they will abort.

The **General Compiler Smart Contract** runs with players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ and consists of two main functions `deposit` and `openCom` and two fixed timestamps `time1`,`time2`.

Commitment Phase: In round t_1, when `deposit`(C_i) together with d coins is triggered from a party P_i, store (i, C_i). Then, if $\forall i, j : C_i = C_j$ proceed to the Opening Phase. Otherwise, for all i, if the message (i, C_i) has been stored, send message d coins to P_i and terminate.

Opening Phase: In round t_2, when `openCom`(i, σ_i) is triggered from P_i, check if $\mathsf{Commit}(\sigma_i) = \gamma_i$, where γ_i is obtained from parsing $C_i = (\gamma_1, \ldots, \gamma_n)$ (recall that all the C_i are the same), and send d coins back to P_i.

Figure 5: General compiler smart contract.

The protocol described is not fair, since we can construct a counterexample that prove that the unfair party P_n can obtain the output without being penalized.

$\mathsf{P}_1, \ldots, \mathsf{P}_n$ will play $\pi_{f'}^{\Gamma}$ to obtain (C, σ_i). As the smart contract is published, P_n will trigger `deposit`(C) together with d coins. At this point, P_n waits that all P_i, with $i \in [n]$ publish the opening σ_i.

When P_n see $\sigma_1, \ldots, \sigma_{n-1}$ on the blockchain, computes the output. If P_n dislikes the output, then he tries to exploit a branch created before the end of the execution of π_{fair} to change messages in that branch to obtain an advantage. Since P_n publishes different messages on different branches of the blockchain, there exist some party P_i, with $i \in [n]$ that will notice it, causing an abort in the protocol.

Let us call b_1 the branch where P_n learned the output and b_2 the branch exploited to change the execution of π_{fair}. We have two cases:

- If b_1 is the branch that will be confirmed on the blockchain, P_n will be penalized.
- If b_2 is the branch that will be confirmed on the blockchain, P_n will cause an abort in the protocol before that the commitment phase starts. In this case he does not get penalized for learning the output.

With this counterexample we show that the proposed solution is not enough to obtain fairness with penalties, since P_n has the possibility to learn the output without incurring in any punishment. It is possible to obtain fairness with penalties with our general compiler, and waiting that the commitment phase is confirmed on the ledger. Since in this case the commitment phase is finalized, an adversary A cannot learn y unless she decides to lose the d coins deposited in the commitment phase. Since the commitment phase is confirmed on the ledger, A cannot find a fork to exploit the execution of the protocol on another branch. Yet, A can cause an abort in the protocol, but if it happens before the commitment phase she will not learn the output y. If the abort happens after the commitment phase, A will learn the output but will be penalized.

Theorem 4. *Let us assume the existence of non-interactive commitment schemes and (n, n)-secret sharing schemes. Let π'_{bc} be an n-party ρ-round protocol realizing f' in the presence of hasty players. Then, the protocol π_{fair} described above securely realizes f satisfying fairness with penalties in the presence of $(\rho, 1)$-hasty players.*

Proof sketch. We can claim security of the compiled protocol π'_{bc} obtained by applying the general compiler to π', by referring to the same proof of Theorem. 3. Now, we argue that the overall protocol π_{fair} achieves fairness with penalties. As mentioned before, aborts during the execution of π'_{bc} are acceptable, since the adversary cannot learn any information about the output. After the committing phase, that is finalized, the adversary could try to exploit different branches to send different openings of his commitments. We have the following timeline: The execution started in a branch b_1 and a forks happens right after the committing phase, generating a branch b_2. Wlog. we can extend this argument to multiple parallel executions in different branches. We have the following scenarios:

Scenario 1: A corrupted player abort in both branches. Since the commitments are finalized, fairness with penalties follows in a straightforward manner, since he did not open his commitments in each branch, and so also in the confirmed one.

Scenario 2: A corrupted player opens his share in b_1 and aborts the execution in b_2 (after the commitment phase). If b_1 gets confirmed, the honest parties will learn the output. If b_2 gets confirmed, it automatically boils down to Scenario 1.

Scenario 3: A corrupted player P_i opens his share in b_1 and tries to open on a different share in b_2. Since the commitment is always confirmed, the adversary cannot try to change his commitment by exploiting forks. If A is able to open the commitment by providing two different shares, then we can define an adversary $\mathsf{A}_{\mathsf{com}}$ breaking the binding property of the underlying commitment scheme with non-negligible probability. That means that at least in one of the two branches P_i gets penalized, and if he provides the correct opening in one of the branches and it gets confirmed, honest players will learn the output.

□

Remark on DoS attacks. Note that in our construction deposits can be made at the end of step (ii) since adversaries trying to violate fairness can be spotted only during step (iii). Therefore an adversary can freely abort the execution before step (iii). Intuitively, by taking as input a protocol achieving identifiable abort [IOZ14] that is publicly verifiable[15], a player cheating in any point of the protocol execution can be successfully spotted and punished. This can be done by making a player posting a smart contract that will act as an external judge that exploits public verifiability of the underlying protocol. Unfortunately, protocols compiled with our construction would lose the identifiable abort property. This is due to the fact that the adversary can make honest players aborting by running two correct executions of the underlying protocol on two branches but using different messages. In such case, the two executions would be both considered valid in both branches by the smart contract mentioned above.

Efficiency analysis. We first analyze the case of the compiler without taking into account the discussion to obtain fairness and then we separately analyze the compiler that produces a protocol that obtains fairness through penalties.

Given an n-party ρ-round UC-secure MPC protocol π, we evaluate the efficiency of a protocol π^* obtained compiling π with the compiler described in Fig. 4. Our compiler adds only one round to the execution of the protocol, in which the parties publish their encryption keys of the underlying encryption scheme and signature keys of the signature scheme. We analyze the number of blocks needed to end π^* in case of standard MPC with aborts.

Since in the setting of MPC with aborts the protocol can be run a full-hasty mode, if the underlying protocol π has ρ rounds, then π^* will have $\rho + 1$ rounds and the number of blocks needed to end the computation is $\rho + 1$.

Let us consider now the case of fairness with penalties. As we noted in Sec. 5.3, to obtain fairness with penalties in a protocol π, the first step to perform is to generate the new protocol π'. Since we assume that the input of the compiler is π', we will evaluate the number of round

[15] An efficient construction can be found at [BOSS20].

added by the compiler to π' and not to π. The protocol π'_{bc} adds only one round to π' and this round does not need to be confirmed. From π'_{bc} we obtain π_{fair} adding other three phases: the publication of the smart contract of Fig 5; the commitment phase and the opening phase. The commitment phase is the only round that needs to be confirmed. Moreover, we notice that the smart contract does not need to be published after the execution of π'_{bc}, then we can add the publication of the smart contract to the first round of π'_{bc}.

If ρ is the number of blocks needed by π' under the assumption listed in Sec. 4.6, π_{fair} needs $\rho + 2 + w$ blocks to terminate, where $w \geq k$ is the liveness parameter. Since we assume that in the ideal conditions all the players broadcast the commitment phase at the same time, this round will be posted and confirmed after at most w blocks.

6 Conclusions

We have focused on MPC protocols implemented on forking blockchains using smart contracts, and have shown how to design such protocols allowing players to be hasty (i.e., without being delayed by finality limitations).

Our work shows that, beyond the double-spending attack, there are other issues that can affect both security and privacy of MPC protocols implemented by a smart contract. On the negative side, we showed that a well-known MPC protocol implemented via smart contracts becomes insecure in the presence of forks and hasty players (because the adversary can play adaptively on a branch of a fork depending on the information observed on the other branch). On the positive side, we have shown smart contracts within on-chain MPC protocols that remain secure even when there are forks and players are hasty.

Moreover we have also discussed how to get fairness with penalties. This allows us to get smart contracts that are simultaneously safe, fair and fast. We have also provided in Sec. 4.6 some experiments to show noticeable improvements of our PCT protocol with respect to the lottery protocol of Andrychowicz et al. in terms of number of blocks needed for completion of the protocol and gas consumption of the smart contracts.

Acknowledgments. We thank Michele Ciampi, Fabio Massacci, Mark Simkin and Roberto Zunino for remarkable comments on this work. We also thank Andrew Miller for useful feedback on a previous version of our paper. Research supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 780477 (project PRIViLEDGE), and in part by GNCS–INdAM, Region Campania (Italy), and the research project SPECTRA funded by Sapienza University of Rome.

References

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 443–458. IEEE Computer Society, 2014.

[ADMM16] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. *Commun. ACM*, 59(4):76–84, 2016.

[BDD20] Carsten Baum, Bernardo David, and Rafael Dowsley. Insured MPC: efficient secure computation with financial penalties. In *Financial Cryptography and Data Security*

- *24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2020.

[BFVV21] Vincenzo Botta, Daniele Friolo, Daniele Venturi, and Ivan Visconti. Shielded computations in smart contracts overcoming forks. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual, March 1-5, 2021*, volume to appear, 2021.

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 421–439. Springer, 2014.

[BKM17] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 410–440. Springer, 2017.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356. Springer, 2017.

[BOSS20] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, pages 562–592, 2020.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with RSA and rabin. In *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 1996.

[CGJ19] Arka Rai Choudhuri, Vipul Goyal, and Abhishek Jain. Founding secure computation on blockchains. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 2019.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986.

[CPS18] T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. Cryptology ePrint Archive, Report 2018/981, 2018. `https://eprint.iacr.org/2018/981`.

[DMM+20] Thomas Dinsdale-Young, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort: A partially synchronous finality layer for blockchains. In *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 24–44. Springer, 2020.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, volume 3386 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 2005.

[Eth] Ethereum team. The ethereum average block time chart. `https://etherscan.io/chart/blocktime`. Accessed: 2020-06-11.

[GG17] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 529–561. Springer, 2017.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[GS09] Vipul Goyal and Amit Sahai. Resettably secure computation. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2009.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 369–386. Springer, 2014.

[KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 30–41. ACM, 2014.

[KB16] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 418–429. ACM, 2016.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 195–206. ACM, 2015.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 839–858. IEEE Computer Society, 2016.

[KVV16] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 406–417. ACM, 2016.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 705–734. Springer, 2016.

[Lys02] Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 597–612. Springer, 2002.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 120–130. IEEE Computer Society, 1999.

[PS17] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 39:1–39:16, 2017.

[PS18] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018.

[PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, 2017.

[RT] Randao's Team. Randao: A dao working as rng of ethereum. `https://github.com/randao/randao`.

[Sol] Solidity team. Solidity bls signatures. `https://github.com/kfichter/solidity-bls`. Smart contract implementation of BLS Signatures.

[SSV19] Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Publicly verifiable proofs from blockchains. In *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part I*, volume 11442 of *Lecture Notes in Computer Science*, pages 374–401. Springer, 2019.

A Smart Contracts

We describe below our smart contract implementations in details.

Lottery protocol by [ADMM16]. The smart contract execution is described as follows:

- *Setup phase:* A player publishes the smart contract in Fig. 7 on page 40, indicating in the constructor the addresses of the players' wallet, and the committing phase timeout `time1` and opening phase timeout `time2`.
- *Committing phase:* The players trigger the `commit` function upon input the commitment to some value and a deposit of `minDep` $= n(n-1)+1$, where $n(n-1)$ coins are used for the penalty mechanism and 1 coin is used to put money for the lottery. After `time1` blocks, the n commitments are collected and the committing phase ends.
- *Opening phase:* All the participants open their commitment by triggering the `openCom` function, and the winner can then claim his bet by triggering the `claimWinner` function of the smart contract (if all the parties have opened).
- *Compensation phase:* If, after `time2` blocks, some player did not open his commitment, the function `payDeposit` can be triggered, so that all player that has not open before `time2` will be penalized and the players who had opened receive their deposit back together with a fraction of the adversaries' deposit.

Our coin-tossing protocol. The description the smart contract execution of π^*_{pct} on Fig. 6 works as follows:

- *Setup phase:* At the beginning, one of the players creates the smart contract specifying a deposit amount `minDep` and timeout `time1`. When the contract is posted on the blockchain, the constructor automatically generates a unique session identifier *sid* by triggering `generateSid`.
- *Deposit phase:* For each $i \in [n]$, P_i can decide to participate to the PCT protocol by triggering the function `deposit` to send a safety deposit and his public key pk_i for an unique signature scheme. After that $(pk_1, \ldots, pk_n)$ are collected by the smart contract and `time1` blocks are passed, the deposit phase ends parties can start to redeem their deposit. [16]
- *Claim phase:* During this phase, each player P_i can claim his deposit back by triggering the function `claim` of the smart contract and sending a value y_i such that $\mathsf{Verify}(pk_i, x, y_i) = 1$, where $x = pk_1 || \cdots || pk_n || sid || bid$, pk_i is the public key of P_i. For the signature verification, we can use a unique signature scheme with fast verification like BLS Signatures [BLS01], invoked with `BLSVerify` in the code, or RSA-FDH [BR96].[17]

[16] Note that when some party P_i sends his deposit to the smart contract, the variable *bid* is updated with the hash of the last block of the contract state (uniquely identifying the branch in which the smart contract state is updated). This implies that *bid* is fixed after the last player sends his public key.

[17] We do not explicitly implement this signature, but solidity implementations are available for testing [Sol].

```
pragma solidity ^0.4.0;

contract ParallelCoinTossing {
  struct Player {
    bool isPlaying;
    bool hasClaimed;
    string pk;
    uint d; //Player's deposit
    uint c; //Player's claim
    }
  address[] playersAddr;
  mapping(address => Player) players;
  uint sid, time1, time2;
  bytes32 bid;

  //flags
  bool claimPhase = false; //true if the claimPhase starts

  //common message to be signed
  uint x;

  constructor(uint _time1, uint _time2) public  {
    sid = generateSid(); //session id
    time1 = _time1; //first timelock
    time2 = _time2; //second timelock
  }
  function deposit(string pubKey) public payable {
    require (msg.sender.balance >= minDep && msg.value >= minDep && players[msg.sender].d == 0 && now < time1);
    playersAddr.push(msg.sender); //add the public key of the current sender
    Player p = players[msg.sender];
    p.isPlaying = true;
    p.pk = pubKey;
    p.d = msg.value;  //msg.value is the deposit value of the player
    bid = block.blockhash(now); //Every time he receives a public key, it updates the blockhash, so that the correct bid is the blockchain state of the last public key deposited.
    }
  function claim(uint y) public {
    require (claimPhase && now < time2 && players[msg.sender].isPlaying && !players[msg.sender].hasClaimed && BLSVerify(players[msg.sender].pk,x,y));
    Player p = players[msg.sender];
    p.c = y;
    p.hasClaimed = true;
    msg.sender.transfer(p.d);
}

  //automatic check functions run after a certain time
  function checkDeposit() public {
    require (!claimPhase && now >= time1);
    uint n = playersAddr.length;
    x = sha3(player[playersAddr[0]].pk||...||players[playersAddr[n-1]].pk||sid||bid);
    claimPhase = true;
    }
}
```

Figure 6: Pseudocode implementation of our smart contract for realizing parallel coin tossing. For simplicity, we omit an explicit definition of the concatenation function in the computation of x.

```solidity
pragma solidity >=0.4.21 <0.6.0;

contract FairLottery {
  struct Player {
    address addr;
    bool hasCommitted, hasOpened, isPlaying;
    uint balance, index;
    bytes32 com;
    int opn;
    }
  uint public n, time1, time2;
  address[] addresses;
  mapping (address => Player) players;

constructor(address[] _addresses, uint _time1, uint _time2) public { //creates a new instance of the lottery for a set of prescribed players
  addresses = _addresses;
  for (uint i = 0; i < addresses.length; i++) {
    Player p = players[addresses[i]];
    p.isPlaying = true;
    p.index = i;
    }
  n = addresses.length;
  time1 = _time1;
  time2 = _time2;
  }
  function commit(bytes32 _com) public payable { //sha3 value commit (n*(n-1) coins + 1 coin for bet)
    require(msg.value >= (n*(n-1)+1)) &&  players[msg.sender].isPlaying && !players[msg.sender].hasCommitted);
    Player p = players[msg.sender];
    p.com = _com;
    p.hasCommitted = true;
    p.balance = msg.value;
  }
  function openCom(int openVar) public  { //opening of the commitment
    require(players[msg.sender].hasCommitted && now > time1 && now < time2 && !players[msg.sender].hasOpened && sha3(openVar) == players[msg.sender].com);
    Player p = players[msg.sender];
    p.hasOpened = true;
    msg.sender.transfer(n*(n-1)); //pays the sender back
  }
  function payDeposit() public { //compensation function
    require(players[msg.sender].isPlaying && now >= time2);
    uint index = players[msg.sender].index;
      for (uint i = 0; i < n; i++)
        if (i != index && !players[addresses[i]].hasOpened) msg.sender.transfer(players[msg.sender].balance/n); //the player msg.sender get is compensation of n coins
  }
  function claimWinner(uint[] secrets) public { //function triggered by the winner
    require (secrets.length == n && checkWinner(secrets,msg.sender));
    msg.sender.transfer(n);  //redeem the won coins
}
//private local functions
  function checkWinner(uint[] secrets, address _sender) private returns (bool) {
    int sum = 0;
      for (uint i = 0; i < secrets.length; i++) {
        if (sha3(secrets[i] != players[addresses[i]].com) return false;
        sum += secrets[i];
      }
    if ((sum%n != players[_sender].index))
      return false;
    return true;
    }
  }
```

Figure 7: Pseudocode implementation of the lottery protocol by Andrychowicz et al. [ADMM16], when using smart contracts.