The Rush Dilemma: Attacking and Repairing Smart Contracts on Forking Blockchains

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

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

October 24, 2019

Abstract

We investigate the security of smart contracts within a blockchain that can fork (as Bitcoin and Ethereum). In particular, we focus on multi-party computation (MPC) protocols run on-chain with the aid of smart contracts, and observe that honest players face the following dilemma: Should I rush sending protocol's messages based on the current view of the blockchain, or rather wait that a message is confirmed on the chain before sending the next one?

To the best of our knowledge, the (implicit) default option used in previous work is the second one and thus known on-chain MPC protocols take long time to be executed on those blockchains with a long confirmation time (e.g., 1 hour per transaction in Bitcoin). While the first option would clearly be preferable for efficiency, we show that this is not necessarily the case for security, as there are natural examples of on-chain MPC protocols that simply become insecure in presence of rushing players.

Our contributions are twofold:

- For the concrete case of fairly tossing multiple coins with penalties, we show that the lottery protocol of Andrychowicz *et al.* (S&P '14) becomes insecure in the presence of rushing players. In addition, we present a new protocol that instead retains security even if the players are rushing.
- We design a compiler that takes any on-chain MPC protocol and transforms it into another one (for the same task) that remains secure even in the presence of rushing players. The only (unavoidable) requirement is that honest players start to be rushing after the first round of the protocol (by all players) has been confirmed on the blockchain.

Our techniques are inspired by ideas on resettably secure computation (Goyal and Sahai, EUROCRYPT '09). We also provide a prototype implementation of our coin tossing protocol using Ethereum smart contracts, and instantiate our generic compiler in a concrete setting, showing that both our constructions yield considerable improvements in terms of efficiency.

Keywords: blockchain, forks, smart contracts, secure computation, resettability.

[*]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 2019

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 joint computation, especially in cases where features like public verifiability and automatic punishment are desired. Instead, 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 and the Double-Spending Problem

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.

The existence of transactions that appear and then disappear from a blockchain is the source of the famous double-spending attack. In such attack, the adversary performs a payment thorough a transaction on the blockchain in order to receive a service off-chain. If later on the transaction related to the payment disappears from the blockchain, due to the presence of forks, then the attacker gets the money back and can spend it for something else. Therefore, at the end of the day, the service was received for free. The crucial point of the double spending attack is that, while the payment transaction disappears, the obtained service is not canceled since it is not linked to the payment transaction happening on chain.

The solution to the double spending problem is pretty harsh: the receiver of a payment will have to wait long time—essentially until the transaction is confirmed and somehow 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.

Interestingly, the double spending problem seems to disappear when instead the service consists of another on-chain transaction that is connected to the payment transaction. Indeed, in this case, if as consequence of a fork the payment transaction disappears, then the service transaction disappears too. This chaining of transactions related to the same process can be easily implemented through smart contracts. Indeed, a smart contract can have an initial state s_1 that is updated transaction by transaction obtaining s_2, s_3, and so on. Let t_i be the transaction that changes the state from s_i to s_{i+1}. If because of a fork the state goes back to s_i from a state s_j, only t_i is again applicable to s_i, while instead all transactions $t_{i+1}, \ldots, t_{j-1}$ are not applicable to state s_i. Therefore, by invalidating t_i (similarly to the double spending problem where money is used in a new transaction making t_i invalid), then $t_{i+1}, \ldots, t_{j-1}$ will be invalidated too.

This motivates the possibility of running smart contracts efficiently, without waiting that every single transaction is confirmed before broadcasting the next one. We have therefore the following rush dilemma: to rush or not to rush?

1.2 Subtle Attacks to Smart Contracts: Efficiency vs Security

Since transactions take long time to be confirmed in a forking blockchain, the full execution of a smart contract with multiple sequential transactions might take too long. It would thus

[1]Throughout the paper, we use the terms "blockchain" and "distributed ledger" interchangeably.
[2]We will often use the two terms "party" and "player" as synonyms.

be natural to speed up the execution of smart contracts by rushing and playing messages immediately. Indeed, as mentioned above, by appropriately chaining the transactions of a smart contract, attacks consisting in exploiting the cancellation of a transaction like the double-spending attack are not effective, and therefore rushing could be a valid option.

However, we notice that forks can help an adversary to mount more subtle attacks. For example, a player could answer to some transaction A by sending another transaction B as soon as A appears on the blockchain. Obviously, in case of forks, the transaction A could appear on the blockchain in different branches, and then multiple copies of B would follow A. While at first sight this seems to be fine, an adversary computing A can exploit his view of B in a branch of a fork to play adaptively a message different than A in another branch, invalidating some security property of the smart contract. Indeed, different transactions A1 and A2 could be played in the two branches of a fork, and (potentially different) transactions B1 and B2 might be required and played as answers. Notice that the honest player could become aware of the fork only after the fact, i.e., after A1 and B1 have been played already. Indeed, because of a fork, transactions A1 and B1 would just disappear, and transaction A2 might appear instead. The honest player therefore will have to compute B2 to continue the execution of the smart contract. The fact that the adversary can play A2 adaptively after having seen B1 can produce a deviation from the expected behavior of the smart contract, therefore compromising the appealing transparency and robustness guarantees of this technology.

In addition, when the transactions represent an on-chain execution of an MPC protocol, the above scenario can be a serious threat for confidential data of honest players.

1.3 Our Contributions

Motivated by the dilemma of rushing risking security or waiting paying on efficiency, we investigate the security of smart contracts that leverage a forking blockchain. Our main contributions are outlined below.

Insecurity of smart contracts with rushing players. Consider a simple 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 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. 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 the smart contract. Such slowness could be unacceptable in some applications.

As our first result, we analyze a variant of the above attack to the well-known smart contract[3] of Andrychowicz *et al.* [ADMM14, ADMM16], for securely realizing multi-party lotteries. The main difference with the toy example from above is that 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

[3]The protocol of [ADMM14, ADMM16] is actually based on Bitcoin, but this makes no difference for our attack.

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 so-called *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). Such a feature is particularly important in light of the negative result by Cleve [Cle86] on achieving fairness without honest majority.

We note that in the protocol of Andrychowicz *et al.* it is vital that players are non-rushing, and therefore post new transactions only after the previous transactions in the protocol are already confirmed on the blockchain. Indeed, in the presence of rushing players, a simple variant of the attack described above would allow a malicious party to 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.

On-chain parallel coin tossing. As our second contribution, we go beyond the limits of the protocol of [ADMM14, ADMM16], and present a smart contract that implements such functionality and remains secure even if the players are rushing. It is important to note that our solution is not a remedy to the [ADMM14, ADMM16] protocol, because we don't have an interest in having a protocol that respects fairness with penalties, but we want to be sure that the output distribution in case of forks and rushing players is unpredictable in any case. 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).

The main idea in our construction consists of using verifiable unpredictable functions (VUF) [MRV99] in order to implement a commitment scheme that simultaneously: (i) has an unpredictable opening for the receiver even in case of multiple openings of the commitment through multiple evaluations of the VUF with the same key but with different inputs, and (ii) achieves binding for the sender through the uniqueness of the VUF. We put together all the openings into a long string that remains unpredictable as long as there is a single honest player. This long string is then given in input to a random oracle to complete the generation of a random string that can be used in various applications (e.g., to run a multi-party lottery).

Notice that this result makes no use of finality of transactions on a blockchain (i.e., we do not need to know after how many blocks a transaction can be considered permanent). 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 on each fork to then select the one that produced the output that she likes the most. This is unavoidable when there is no use of finality of transactions, and we tackle this in our next result that instead takes finality of transactions into account.

On-chain MPC with rushing players. As final contribution, we show a general transform to design smart contracts that retain security in the presence of forks (and thus, when properly instantiated, can be more efficient since players are allowed to rush). Our transform is inspired by previous works that show how to make MPC protocols secure against *reset attacks* [GS09, GM11]. The main idea consists of asking each player to first commit to its input and randomness, and then to run the underlying MPC protocol by additionally proving in zero knowledge[4] that

[4]Technically, the zero-knowledge proof must also be secure w.r.t. reset attacks. Moreover, the fact that such proofs come from different players requires to protect them w.r.t. man-in-the-middle attacks. To fulfill both

each message has been computed correctly w.r.t. the initial commitment.

In addition to connecting the problems of security w.r.t. fork attacks with previous work on resettable security, we actually consider a slightly different setting that remained so far unexplored in the literature. Indeed, in order to model forks we notice that: (i) honest rushing players would use the same input on both branches of a fork, but the randomness would be fresh (since it would be sampled multiple times and the random tape does not move back); (ii) while reset attacks are generically allowed to any previous point of the computation, we can leverage on the properties of a blockchain in order to make sure that there is a bound beyond which one can consider a transaction finalized (e.g., 6 is the commonly recommended number of blocks after which a transaction is considered irreversible in then Bitcoin blockchain), and therefore a transaction can be considered confirmed at some point. Our transform takes advantage of the above two simplifications by requiring honest players to wait for the confirmation of the first transaction of each other player, and by avoiding the use of a pseudo-random function that is traditionally used to change randomness after a reset [GS09].

Note that here we make a minimal use of the finality guarantee of a blockchain, since we ask to wait for the confirmation of only the first message of each player. Waiting for a single confirmation is optimal in light of known impossibility results for several interesting functionalities[5] in the presence of arbitrary reset attacks (for us, honest players that because of forks end up playing multiple times the same step of a smart contract or of an MPC protocol).

We also discuss a concrete instantiation of our transform on a natural smart contract that would be insecure in the presence of rushing players. Besides achieving security in case of forks, the final protocol features an improved efficiency w.r.t. time to completion, at the price of a small overhead in the communication complexity.

1.4 Related Work

Following [ADMM14, ADMM16], several other works focus on achieving fairness with penalties for different applications of interest, including lotteries [BK14, BZ17], decentralized poker [KMB15, BKM17], and general-purpose computation [BK14, KMS$^+$16, KB16, KVV16]. In particular, the line of works by Kumaresan *et al.* relies on an elegant paradigm working in two phases: During the first phase, the 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. During the second phase, the output is then reconstructed in a fair manner on chain.

Unfortunately, the security of this paradigm in the presence of rushing players is difficult to assess, as it relies on intermediate ideal functionalities (such as the "claim-or-refund" and "multi-lock" functionality [BK14, KB14]) that, while they can be implemented using Bitcoin or Ethereum, offer a-priori no security guarantee in the presence of reset attacks due to blockchain forks. Moreover, known results about designing protocols in a hybrid model that allows to make calls to a functionality are applicable only to the classical setting where resets 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 run 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, BGM$^+$18, SSV19, CGJ19], of which Bitcoin and Ethereum

requirements we use proofs that are non-interactive and non-malleable. Moreover, for simplicity, we will use the term "proof" rather than "argument" even when computational soundness is sufficient.

[5]E.g., in oblivious transfer a malicious receiver by playing again his very first message would be allowed to obtain both inputs of the sender.

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-rushing.

2 Preliminaries

2.1 Notation

Given an integer n, we let $[n] = \{1, \ldots, n\}$. 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

Verifiable unpredictable functions. A verifiable unpredictable function (VUF) [MRV99] is a tuple of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Ver})$ 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 vk; (ii) The deterministic algorithm Eval takes as input the secret key sk and an input x within domain $\mathcal{D}$, and outputs a value y belonging to some range $\mathcal{R}$; (iii) The deterministic algorithm Prove takes as input the secret key sk and an input $x \in \mathcal{D}$, and outputs a proof $\phi \in \{0,1\}^*$; (iv) The randomized algorithm Ver takes as an input the verification key vk, an input/output pair (x, y), and a proof ϕ, and outputs a decision bit. Correctness says that for sufficiently large $\lambda \in \mathbb{N}$, with overwhelming probability over the choice of $(sk, vk) \leftarrow\$ \mathsf{Gen}(1^\lambda)$, for all $x \in \{0,1\}^\ell$ it holds that $\mathsf{Ver}(vk, x, \mathsf{Eval}(sk, x), \mathsf{Prove}(sk, x)) = 1$.

As for security, we require two properties known as unique provability and unpredictability.

Definition 1 (Unique provability)**.** We say that $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Ver})$ satisfies unique provability if no values $(vk, x, y_0, y_1, \phi_0, \phi_1)$ with $y_0 \neq y_1$, even maliciously generated, can satisfy $\mathsf{Ver}(vk, x, y_0, \phi_0) = \mathsf{Ver}(vk, x, y_1, \phi_1)$. In words, for every string vk and every $x \in \mathcal{D}$, there exists at most one value $y \in \mathcal{R}$ for which there is an accepting proof.

Definition 2 (Unpredictability)**.** We say that $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Ver})$ satisfies computational unpredictability if for all non-uniform *valid* PPT attackers A the following quantity is negligible:

$$\mathbb{P}\left[\mathsf{Eval}(sk, x) = y : \begin{array}{c} (vk, sk) \leftarrow\$ \mathsf{Gen}(1^\lambda) \\ (x, y) \leftarrow\$ \mathsf{A}^{\mathcal{O}(sk, \cdot)}(vk) \end{array}\right],$$

where $\mathcal{O}(sk, \cdot) \equiv \{\mathsf{Eval}(sk, \cdot), \mathsf{Prove}(sk, \cdot)\}$, and attacker A is called *valid* if it never queries x to its oracles.

Verifiable unpredictable functions (a.k.a. *unique signatures*) exist based on a variety of assumptions [BR96, MRV99, Lys02, Dod03, DY05, Jag15, HJ16, Bit17].

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 3 (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 4 (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 $\mathsf{Commit}(m_b)$.

Non-interactive commitments satisfying the above properties can be obtained under the assumption of injective one-way functions [GMW91].

Non-interactive zero knowledge. Let R be a relation, corresponding to an NP language L. A non-interactive proof system for R is a tuple of efficient algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ specified as follows. (i) The randomized algorithm Setup takes as input the security parameter and outputs a common reference string σ; (ii) The randomized algorithm $\mathsf{Prove}(\sigma, (x, w))$, given a pair (x, w) s.t. $R(x, w) = 1$ outputs a proof ϕ; (iii) The deterministic algorithm $\mathsf{Ver}(\sigma, (x, \phi))$, given an instance x and a proof ϕ outputs either 0 (for "reject") or 1 (for "accept"). We say that a NIZK for relation R is *correct* if for every $\lambda \in \mathbb{N}$, all σ as output by $\mathsf{Setup}(1^\lambda)$, and any (x, w) s.t. $R(x, w) = 1$, we have that $\mathsf{Ver}(\sigma, (x, \mathsf{Prove}(\sigma, (x, w)))) = 1$.

We define two properties of a non-interactive proof system. The first property says that honest proofs do not reveal anything beyond the fact that $x \in L$.

Definition 5 (Adaptive multi-theorem zero-knowledge [Gro06])**.** A non-interactive proof system $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ for a relation R satisfies adaptive multi-theorem zero-knowledge if there exists a PPT simulator $\mathsf{ZKSim} := (\mathsf{ZKSim}_0, \mathsf{ZKSim}_1)$ such that for all PPT non-uniform adversaries A the following quantity is negligible:

$$\Big|\mathbb{P}\left[\mathsf{A}^{\mathsf{Prove}'(\sigma,\cdot,\cdot)}(\sigma) = 1 : \sigma \leftarrow\$ \mathsf{Setup}(1^\lambda)\right] - \mathbb{P}\left[\mathsf{A}^{\mathsf{ZKSim}_1'(\zeta,\cdot,\cdot)}(\sigma) = 1 : (\sigma, \zeta) \leftarrow\$ \mathsf{ZKSim}_0(1^\lambda)\right]\Big|,$$

where $\mathsf{ZKSim}_1'(\zeta, x, w) = \mathsf{ZKSim}_1(\zeta, x)$ if $R(x, w) = 1$, and both oracles ZKSim_1' and Prove' output $\perp$ if $R(x, w) = 0$. Sometimes we call a non-interactive proof system satisfying adaptive multi-theorem zero knowledge a *NIZK proof system.*

The second property states that knowledge soundness holds even if the adversary can see simulated proofs for possibly false statements of its choice.

Definition 6 (Simulation extractability [Gro06])**.** A NIZK proof system $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ with zero-knowledge simulator $(\mathsf{ZKSim}_0, \mathsf{ZKSim}_1)$ for a relation R satisfies simulation extractability if there exists a PPT algorithm $\mathsf{KExt} = (\mathsf{KExt}_0, \mathsf{KExt}_1)$ such that the output of KExt_0 is identical to that of ZKSim_0 when restricted to the first two components, and moreover for every non-uniform PPT adversary A the following quantity is negligible:

$$\mathbb{P}\left[\begin{array}{c} \mathsf{Ver}(\sigma, x^*, \phi^*) = 1 \wedge \\ (x^*, \phi^*) \notin \mathcal{Q} \wedge \\ R(x^*, w^*) = 0 \end{array} : \begin{array}{c} (\sigma, \zeta, \xi) \leftarrow\$ \mathsf{KExt}_0(1^\lambda) \\ (x^*, \phi^*) \leftarrow\$ \mathsf{A}^{\mathsf{ZKSim}_1(\zeta,\cdot,\cdot)}(\sigma) \\ w^* \leftarrow\$ \mathsf{KExt}_1(\xi, x^*, \phi^*) \end{array}\right],$$

where $\mathcal{Q}$ is the list of all queries to oracle $\mathsf{ZKSim}_1(\zeta, \cdot, \cdot)$.

3 Parallel Coin Tossing

In this section we focus on blockchain-aided MPC for parallel coin tossing. Roughly, such protocols allows a set of players to agree on a uniformly random string, and have many important applications. (For instance, they trivially imply a fair lottery.) After recalling the lottery protocol by Andrychowicz *et al.* [ADMM14, ADMM16], we show that this construction is not secure in the presence of rushing players. We then propose a new protocol which achieves security in the presence of rushing players, leveraging the power of smart contracts, but as we noted before we lose the fairness guaranteed by the lottery protocol by Andrychowicz *et al.*.

3.1 The Protocol of Andrychowicz *et al.*

Background on Bitcoin. 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 the proof of work (PoW) [DN92]; users willing to add a transaction to the ledger forward it to all the miners, which will try to include it in the next minted block.

Due to the PoW consensus mechanism, each miner could have a different view of the ledger. The *common-prefix property* [GKL15] roughly states that all the miners have the same view of the blockchain up to a certain number k of blocks (before the last block); this guarantees long-term consistency of the transactions in the ledger. Each view of the blockchain is called a *fork*; after k blocks, with noticeable probability, one of these forks will be part of the common prefix.

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., at least k blocks have passed). Each transaction Tx contains the following information:

- 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).

A transaction is called *standard* if its output script contains only the signature of the account's owner (i.e., it can be redeemed by the owner by simply signing it).

The protocol. 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 $\mathsf{PayDeposit}^j_i$ redeeming Commit^j_i, sends it off-chain to each $\mathsf{P}_{i\neq j}$, and finally posts all the Commit^j_i transactions on the ledger. In case P_j does not open the commitment before time t, then each recipient of a $\mathsf{PayDeposit}^j_i$ transaction can sign it and post it on the ledger; since time t has passed, the miners will now accept the transaction as a valid transaction redeeming Commit^j_i.

More in details, the protocol works as follows.

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 $\mathsf{PayDeposit}^j_i$ transactions (with time-lock t) to each $\mathsf{P}_{i\neq j}$, and posts the Commit^j_i transactions on the ledger.

Betting phase: P_j bets one coin in the form of a standard 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 t, in case some player P_j did not send all of its $\{\mathsf{Open}^j_i\}_{i\in[n],i\neq j}$ transactions, all the other players $\mathsf{P}_{i\neq j}$ can post the $\mathsf{PayDeposit}^j_i$ transaction on the ledger, thus obtaining at least a certain number of coins as compensation.

3.2 A Simple Attack

The main idea behind our attack is that, in the presence of rushing players, the protocol's messages can end-up on unconfirmed blocks. By looking at different forks, an attacker can try to change an old unconfirmed transaction by re-posting it, with the hope that it will end-up on a different fork and become part of the common prefix. This essentially corresponds to a reset attack on the protocol.

The construction described in the previous section relies on the (implicit) assumption that the players are non-rushing. In particular, each player P_j should wait to post its $\mathsf{PutMoney}^j_i$ transaction only after all the Commit^j_i transactions are confirmed on the ledger, in such a way that all players are aligned on the same fork (and so the miners have the $\{\mathsf{Commit}^j_i\}_{i\in[n],j\neq i}$ transactions in their common prefix).

In the case of rushing players, when a fork occurs, an attacker can take advantage of 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 $w = x_1 + \ldots + x_n \bmod n+1$. Consider the following scenario:

- The (rushing) players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ run a full instance of the lottery protocol; note that this requires at least 3 blocks.
- The attacker P_n hopes to see a fork containing all the $\{\mathsf{Commit}^i_j\}$ transactions of the other $n-1$ players.
- Since the attacker P_n now knows the opening $x_1, \ldots, x_{n-1}$ of the other participants, 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 fork which is finally included in the common prefix, P_n wins the lottery.

In the above scenario, the attacker can freely bias the result of the lottery. In the next section, we propose a new protocol (running on Ethereum) that does not suffer from this problem.

3.3 Our New Lottery Protocol: An Ethereum Smart Contract

We now present a smart contract for Ethereum that allows to run a parallel coin-tossing protocol with rushing players. The main challenge is that the protocol must prevent that an adversary chooses adaptively in a branch of a fork her contribution to the coin tossing after having seen the contributions of others in other branches. We tackle this problem by requiring that honest players compute their contributions evaluating a verifiable unpredictable function (VUF) on input the public keys of all players. Notice that if the adversary sees some evaluations of the VUF in a branch, and changes her public key in another branch, then the VUF evaluations of the honest players on this other branch are unpredictable, and thus the adversary will not manage to control the final output. Moreover, changing the public key is the only possibility for the adversary because the VUF has a uniqueness property, and thus, once the public key is selected, the adversary can play a single message that is accepted as correct by the honest players.

Informally, the protocol works as follows:

- A contract is published on the blockchain with the unique identifier sid.
- Every player that wants to play in the protocol generates the public and private keys for the VUF as $(pk, sk) \leftarrow\$ \mathsf{Gen}(1^\lambda)$, and shares pk with all the protocol participants publishing it in on the contract.
- For all $i \in [n]$, using the VUF, P_i evaluates $y_i = \mathsf{Eval}(sk_i, \mathsf{Hash}(pk_1||...||pk_n||sid))$ and its proof ϕ_i, and publishes (y_i, ϕ_i) on the smart contract.
- The smart contract checks for all $i \in [n]$ that y_i is obtained using a VUF with input $\mathsf{Hash}(pk_1||...||pk_n||sid)$, where pk_i is the address of the i-th player. Then the smart contract defines the output as $\mathsf{Hash}(y_1||...||y_n)$.

Intuitively, the adversary can not bias the output of the protocol because, as long as the input of Hash is unpredictable, the final output will be random, since we model Hash as a random oracle[6]. Moreover, unpredictability of the input to Hash comes from the unpredictability of the output of a VUF of at least one honest player for any sequence of public keys, and from the uniqueness of the output of the adversary once her public keys are established.

[6]Note that even in [ADMM14, ADMM16] the Hash function is implemented as a random oracle to have a non-malleable commitment scheme.

Background on Ethereum. Ethereum transactions includes two types of accounts: *externally-owned accounts* (EOAs), controlled by the private keys of the users, and *contract accounts* (CAs), controlled directly by their code [Woo19]. Both types of accounts have a balance in ETH (also denoted Ξ). The blockchain tracks the state of every account in its blocks. The transactions posted by an EOA consist of a destination address, a signature s, the amount of *wei* (subunits of Ξ), and an optional data field representing the inputs of a contract (which we consider as messages to a contract).[7]

Each time an EOA creates a transaction, it is replicated to all the miners of the blockchain. If such a transaction contains inputs to some contract, it triggers the specified contract via a function call: each miner, when receiving a new transaction during the replication process, checks for the triggered contract on the blockchain, and starts to run it using its current state α, and the inputs specified in the transaction. Contracts can send messages (i.e., transactions) to the EOAs or other contracts, which means that when a miner runs the code of a contract with some prescribed input, the output contains messages (i.e., transactions) directed to some other entity. We can model both EOAs transactions and messages from/to a contract as different types of transactions. In particular, a block might contain the following elements.

Message transactions: Represented as a tuple $\mathsf{Tx}_{\mathsf{msg}} = (\mathcal{T}, s, pk, \vec{x}, \mathtt{data})$, where $\mathcal{T}$ is the set of transactions from which $\mathsf{Tx}_{\mathsf{msg}}$ is redeeming,[8] s is the signature of the owner, pk is the public key of the receiver (either a contract or an EOA), $\vec{x}$ is the vector of inputs to the contracts (in case the receiver is a contract), and `data` represents some extra data.

Contract transactions: Represented as a tuple: $\mathsf{Tx}_{\mathsf{cnt}} = (\mathcal{T}, s, \beta, \Psi)$, where β is the initial balance of the contract, Ψ is the code of the contract, and $\mathcal{T}$, s are the same as in message transactions. Each time a new transaction triggers some contract, the miners check that it is valid (e.g., that the inputs are correctly formed, and that the balance is enough). The program Ψ takes as input the current state of the contract α and a vector $\vec{x}$, and outputs a sequence of transactions $(\mathsf{Tx}_1, \mathsf{Tx}_2, \ldots)$ to possibly different recipients and a new state α'; when a miner includes a message transaction Tx_i in a new block, it also updates the current state of the contract.

Each contract has associated a sequence of states $(\alpha_1, \alpha_2, \cdots)$, where α_i is the i-th state of the contract.[9] Typically, the state includes the global variables of the contract plus all messages sent/received from/by the contract itself. Note that, in case of forks, different miners can see diverging states for the same contract, i.e., $\alpha = (\alpha_1, \ldots, \alpha_i, \alpha_{i+1}, \cdots)$ and $\alpha' = (\alpha_1, \ldots, \alpha_i, \alpha'_{i+1}, \cdots)$.

When interacting with a smart contract, a player sends message transactions without waiting the minimum number of blocks that guarantee all the miners have a consistent view of the contract's state. This, in particular, means that if we describe the lottery protocol of [ADMM14, ADMM16] using a single contract, our attack from §3.2 still works, since the protocol's messages sent by the committer to the contract would appear in different blocks of the blockchain, which makes the protocol insecure in the presence of rushing players.

Protocol description. We are now ready to describe our protocol for parallel coin tossing. Roughly, our construction follows the steps described below.

[7]For our purposes, we omit additional fields such as the `gasLimit`, `gasPrice`, and `nonce` values.

[8]The miners, in order to validate the transaction, must check that the signer is also the owner of the redeemed transactions, and that those have a high enough balance.

[9]In Ethereum, the state of all active contracts is stored efficiently using Patricia trees [Woo19]. For our purpose, it suffices to assume that a miner can determine the current state of an active contract by looking at past blocks included in the blockchain.

Setup phase: At the beginning, one of the players creates the smart contract described in Fig. 3 on page 27 specifying a minimum deposit amount $\bar{q}$ and timeouts t_1, t_2 (denoted as hard-wired constants `minDep`, `time1`, `time2` in the contract). Note that when the contract is posted on the blockchain, the function `beginCoinTossing` is triggered yielding a unique session identifier sid.

Deposit phase: An arbitrary number of players can decide to participate to the coin-tossing protocol by sending (via the function `deposit`) a safety deposit. The smart contract automatically keeps track of the address pk of each player and its deposit. After time t_1, the deposit phase ends and the redeem phase starts.[10]

Redeem phase: During this phase, each player can claim its deposit back by sending (via the function `claim`) to the smart contract a value y (denoted `rand` in the contract) along with a proof ϕ (denoted `proof` in the contract) showing that y was obtained by evaluating a VUF upon $\mathsf{Hash}(pk_1||\ldots||pk_n||sid)$, where pk_i is the address of the i-th player who deposited, and Hash is a cryptographic hash function.

After time t_2, the `checkClaimed` function checks whether some player did not claim its deposit back or provided an invalid proof, in which case its deposit is used in order to compensate all the honest players (i.e., those that followed the protocol correctly) via the function `penalize`. Note that in case all players behave honestly, each of them can compute $\mathsf{Hash}(y_1||\cdots||y_n)$ which is defined to be the output of the coin tossing protocol.

A more formal description of the protocol follows below.

Parallel Coin Tossing protocol π

Running with hard-wired parameters $\bar{q}, t_1, t_2$. Let $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Ver})$ be a VUF with range $\mathcal{R} \equiv \{0,1\}^\ell$, and $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function. After the smart contract of Fig. 3 has been created by one of the players, the protocol continues as follows:

- Any willing player deposits $d \geq \bar{q}$ coins and specifies a public key pk by triggering the `deposit` function, where $(pk, sk) \leftarrow\$ \mathsf{Gen}(1^\lambda)$.
- After time t_1, let n be the number of participants and denote by P_i the player that made the i-th deposit (specifying public key pk_i). Each player P_i, in an arbitrary order, executes the steps below:
  - Compute $y_i = \mathsf{Eval}(sk_i, x)$, $\phi_i = \mathsf{Prove}(sk_i, x, y_i)$, where $x = \mathsf{Hash}(pk_1||\ldots||pk_n||sid)$ and the values $(pk_j)_{j\neq i}$ and sid are retrieved from the state of the contract.
  - If the flag `redeemPhase` contained in the state of the contract is set to `true`, trigger the `claim` function with inputs (y_i, ϕ_i).
- After time t_2, upon receiving a compensation from the contract (meaning that some party aborted) stop the execution. Else, retrieve $y_1, \ldots, y_n$ from the contract and output $\mathsf{Hash}(y_1||\cdots||y_n)$.

Security. We establish the following result.[11]

[10] Technically, in Ethereum, an automatic service would trigger the `checkDeposit` function after time `time1`; similarly, the function `checkClaimed` would automatically be invoked after time `time2`.

[11] We state the theorem informally. A more formal statement would require modeling MPC with a forkable blockchain supporting smart contracts, and in the presence of rushing players. This is beyond the scope of this

Theorem 1. *Assuming that* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Ver})$ *is a VUF satisfying perfect uniqueness and computational unpredictability, and that* Hash *is a non-programmable random oracle, then in the protocol* π*, when no abort happens, the honest players output a string computationally close to uniform, even if the players are rushing and only one player is honest.*

Proof. We prove the security in the presence of a single honest player, and when the parties are rushing. Consider the following sequence of games.

Game $\mathbf{G}_0(\lambda)$: This is identical to an execution of protocol π in the presence of rushing players. Let us denote with m the total number of protocol's instances, where we uniquely identify an instance by the tuple $(pk_1^{(r)}, \ldots, pk_n^{(r)}, sid)$ for some $r \in [m]$. Note that, since the players are rushing, there are indeed many instances, and each instance can possibly be run multiple times. Given an instance $r \in [m]$, it is important to observe that the index i of the honest player and the number of parties n actually depend on r, i.e., $i := i^{(r)}$ and $n := n^{(r)}$; however, to simplify notation, we ignore this dependency. Our goal is to show that the output of honest players in each instance is computationally close to uniform.

Game $\mathbf{G}_1(\lambda)$: Identical to the previous game, except that now the experiment artificially stops if $\exists r^* \in [m]$ such that during the redeem phase of two protocol runs with the same instance $(pk_1^{(r^*)}, \ldots, pk_n^{(r^*)}, sid)$, a corrupted player $\mathsf{P}_{j\neq i}$ outputs values $(y_{0,j}^{(r^*)}, \phi_{0,j}^{(r^*)})$ and $(y_{1,j}^{(r^*)}, \phi_{1,j}^{(r^*)})$ for which $y_{0,j}^{(r^*)} \neq y_{1,j}^{(r^*)}$ and $\mathsf{Ver}(pk_j^{(r^*)}, x^{(r^*)}, y_{0,j}^{(r^*)}, \phi_{0,j}^{(r^*)}) = \mathsf{Ver}(pk_j^{(r^*)}, x^{(r^*)}, y_{1,j}^{(r^*)}, \phi_{1,j}^{(r^*)}) = 1$, where $x^{(r^*)} = pk_1^{(r^*)}||\cdots||pk_n^{(r^*)}||sid$.

We claim that $\{\mathbf{G}_0(\lambda)\}_{\lambda\in\mathbb{N}} \equiv \{\mathbf{G}_1(\lambda)\}_{\lambda\in\mathbb{N}}$. This is because the only difference between the two games is when an artificial abort happens. However, any attacker A triggering such an abort can be turned into a non-uniform A' breaking perfect uniqueness. The reduction A' simply runs A until the artificial abort is provoked, and then outputs $(pk_j^{(r^*)}, y_{0,j}^{(r^*)}, \phi_{0,j}^{(r^*)}, y_{1,j}^{(r^*)}, \phi_{1,j}^{(r^*)})$. Since the reduction is straightforward, we omit further details.

Consider the following event $\mathbf{BAD}$, defined over the probability space of $\mathbf{G}_1(\lambda)$: The event becomes true if $\exists r^* \in [m]$ such that the attacker A is able to predict the input $y_1^{(r^*)}||\cdots||y_n^{(r^*)}$ to the random oracle Hash *before*[12] the honest party P_i reveals the VUF output $y_i^{(r^*)}$ during the redeem phase of instance $(pk_1^{(r^*)}, \ldots, pk_n^{(r^*)}, sid)$. In order to finish the proof, it suffices to show that for all PPT A playing in $\mathbf{G}_1(\lambda)$, it holds that $\mathbb{P}[\mathbf{BAD}] \in \mathtt{negl}(\lambda)$; in fact, assuming $\mathbf{BAD}$ does not happen, the final output of the protocol is determined by running the random oracle Hash on a fresh input, which yields a truly random output.

By contradiction, let us assume that there exists a PPT adversary A provoking event $\mathbf{BAD}$ with non-negligible probability. We construct a non-uniform PPT attacker A' breaking computational unpredictability of the VUF.

- At the beginning, A' receives a target public key pk from the challenger and samples[13] $r^* \leftarrow\$ [m]$.
- Hence, A' runs A and simulates the deposit phase as follows:
  - For each instance $r < r^*$ associated to a fork during an execution of the protocol, A' samples $(pk_i^{(r)}, sk_i^{(r)}) \leftarrow\$ \mathsf{Gen}(1^\lambda)$ and uses $pk_i^{(r)}$ as the public key of the honest player during the deposit phase.
  - For the instance $r = r^*$, A' uses the target pk as the public key of the honest player.

work.

[12]Of course, after the value $y_i^{(r^*)}$ is revealed, the attacker can compute the input to the random oracle.

[13]More precisely, A' samples $r^* \leftarrow\$ [m']$ where $m' < m$ is the total number of instances with a distinct public key for the honest player.

  - For each instance $r > r^*$, A' samples $(pk_i^{(r)}, sk_i^{(r)}) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$ and uses $pk_i^{(r)}$ as the public key of the honest player during the deposit phase, unless the instance r starts with prefix $(pk_1^{(r^*)}, \ldots, pk_i^{(r^*)}, *)$, in which case the target pk is used.

- During the redeem phase, A' proceeds as follows:
  - For each instance $(pk_1^{(r)}, \ldots, pk_n^{(r)}, sid)$ that does not include the target public key pk, run the protocol honestly. (This is possible since A' knows $sk_i^{(r)}$.)
  - For each instance $(pk_1^{(r)}, \ldots, pk_n^{(r)}, sid)$ such that $pk_i^{(r)} = pk$, query the target Eval and Prove oracles upon input $x^{(r)} = pk_1^{(r)}||\cdots||pk_n^{(r)}||sid$, and use the output from the oracles to simulate the values $(y_i^{(r)}, \phi_i^{(r)})$ used by the honest player during the redeem phase.
  - The first time that A outputs a guess y^* for the input to the random oracle, check that this happens w.r.t. to the instance r^*. If not, abort the simulation. Else, parse $y^* = y_1^{(r^*)}||\cdots||y_n^{(r^*)}$ and forward $(x^{(r^*)}, y_i^{(r^*)})$ to the challenger, where $x^{(r^*)} = pk_1^{(r^*)}||\cdots||pk_n^{(r^*)}||sid$.

By inspection, assuming that A' does not abort, the view of A is perfectly simulated. It follows that with non-negligible probability A will guess y^* correctly, which in particular means that $y_i^{(r^*)} = \mathsf{Eval}(sk, x^{(r^*)})$ and thus, since the value $x^{(r^*)}$ is fresh, A' breaks computational unpredictability of the VUF. This, together with the fact that the probability that A' does *not* abort is an inverse polynomial (as m is bounded by a polynomial in the security parameter), concludes the proof of the theorem. □

4 A Generic Compiler Exploiting Finality

In this section, we present a generic solution to the problem of rushing in the presence of forks for blockchain-aided MPC protocols. Our solution assumes that at some point a transaction can be considered permanently part of the blockchain. Of course, there is no issue related to forks if one always delays protocol messages waiting until the previous message becomes permanent. What we do here is to make a very mild use of this assumption in order to amortize the delay introduced by the wait for a confirmation. More specifically, we will require that players wait enough time to make sure that only the first message of the protocol is confirmed on the blockchain, but in all subsequent rounds players are allowed to rush without worrying about forks.

4.1 ABC Resettability

Consider n parties (indexed by $[n]$) running an interactive protocol π for computing some joint function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ of their private inputs. We denote by x_i and ω_i the input and randomness of the i-th player P_i. As usual in the setting of MPC, we define security by comparing the real world in which protocol π is executed, with an ideal world where a trusted party collects the inputs from all the players, computes f, and distributes the outputs to the parties.

The real model. We assume a synchronous setting where protocol π proceeds in rounds. The protocol's execution is facilitated by the existence of a blockchain on which the players write protocol's messages. We will not need to specify the properties of the underlying blockchain formally, the only things that we need to assume are that: (i) the blockchain is forkable, in the sense that at a given round a player sees (or rather, obtains from the miners according to some rule) a version of the blockchain which might differ from the "main chain" in the last blocks; (ii) if a player waits long enough, it can be sure that a given message is confirmed on the "main chain", in the sense that such a message will be included in all future forks. We call a player *rushing* if it automatically writes protocol's messages based on its current view of the blockchain; note that by assumption (i) this has the effect that a given round could be repeated multiple times with slightly different versions of the blockchain. In contrast, a *non-rushing* player always waits that each message is confirmed on the "main chain" before sending the next message. A player is *c-rushing* if it is non-rushing up to round $r \leq c$, and it becomes rushing afterwards.

Consider now the execution of π with c-rushing players, and in the presence of an adversary A 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.[14] The parties then start running the protocol, with the honest players P_i behaving as prescribed in π (using input x_i), and with malicious parties behaving arbitrarily (directed by A). As usual, the attacker can delay sending the round-r messages of the corrupted parties until after the honest players send their round-r messages.

We abstract away the presence of the blockchain with c-rushing players, by simply allowing the players to access an authenticated broadcast channel[15] until round c, after which the attacker A is allowed to reset[16] an honest party to any round $r > c$; importantly, after a reset, the player continues to execute the protocol starting from round r but using *fresh* randomness. We call such an adversary *all-but-c (ABC) resetting.* At some point, A gives to D an arbitrary function of its view, and D additionally receives the outputs of the honest parties and must output a bit. We denote by $\mathbf{REAL}^c_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ the random variable corresponding to D's guess.

The ideal model. In the ideal world, a trusted third party evaluates the function f on behalf of the players. As in the real setting, 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 the ideal adversary S. 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 D an arbitrary function of its view, and D additionally receives the outputs of the honest parties and must output a bit. We denote by $\mathbf{IDEAL}_{f,\mathsf{S},\mathsf{D}}(\lambda)$ the random variable corresponding to D'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. 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 special message `continue` or `abort` to the trusted party; in the former case all the honest parties are given their output y_i, whereas in the latter case they receive an abort

[14]For simplicity, we only consider static corruptions (i.e., the distinguisher decides who is corrupt at the beginning of the protocol).

[15]By writing protocol's messages on the blockchain, indeed, non-rushing players realize a special broadcast channel that allows sleepy players (i.e. parties that are currently offline) to participating consistently to the protocol after being back online.

[16]This makes our model only stronger, as in practice the attacker might not be able to reset a player to a state too back in the past.

symbol $\bot$. We denote by $\mathbf{IDEAL}_{f_\bot,\mathsf{S},\mathsf{D}}(\lambda)$ the modified random variable corresponding to D's final guess.

The definition. We are now ready to define ABC resettable security.

Definition 7 (ABC resettability)**.** We say that π (t, c)-securely computes f in the presence of malicious adversaries if for any all-but-c resetting 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:

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

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

Remark 1 (On the parameter c)**.** *When $c = \infty$ (i.e., the attacker is not allowed to reset), we recover the standard notion of simulation-based security for multiparty protocols. On the other hand, when $c = 0$ (i.e., the attacker can always reset the protocol from the start)—as long as the distinguisher is allowed to choose different inputs after each reset—the real model specification becomes similar to that of resettably secure computation [GS09, GM11], the only difference being that in ABC resettability honest players use fresh randomness after each reset.*

In case of full resets, we must also give the simulator the possibility to reset the execution in the ideal world, which makes this notion meaningful only for certain functionalities. Notice that when $c = 0$ it is unavoidable that each branch might correspond to a different execution with different inputs of the same protocol. If the adversary has also some control over which branch becomes the one permanently stored in the blockchain, then of course she can select the execution of the protocol that has a better output (according to her taste).

4.2 Compiler Description

Let π be any MPC protocol for securely computing some function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$. It will be useful to describe π in terms of the next-message functions $\psi^\pi = (\psi^\pi_1, \ldots, \psi^\pi_n)$:

$$\forall i \in [n], r \geq 0 : \ \psi^\pi_i(x_i, \omega_i, \tau^{(r)}_i) = m^{(r+1)}_i,$$

where x_i (resp. ω_i) is the input (resp. random tape) of P_i, $m^{(r+1)}_i$ is the message that P_i writes on the blockchain at round $r + 1$, and $\tau^{(r)}_i$ is the collection of all protocol's messages received by P_i up to round r (with $\tau^{(0)}_i = \varepsilon$ being the empty string). Sometimes, we will think of ψ^π as a single function further taking as input the index $i \in [n]$ of the player, i.e., $\psi^\pi(i, x_i, \omega_i, \tau^{(r)}_i) = \psi^\pi_i(x_i, \omega_i, \tau^{(r)}_i)$.

The basic idea of our compiler is to add an initial round in which all the players commit to their input x_i and random tape ω_i. Hence, each move of the original protocol π is augmented with a non-interactive zero-knowledge proof showing that the last message has been computed correctly using the same input and random tape included in the commitment. We provide a formal specification in Fig. 1.

4.3 Security Analysis

The theorem below states that the final protocol is secure in the presence of 1-rushing players, i.e., as long as the parties wait for the end of the first round before starting to forward protocol's messages on different forks of the blockchain.

Compiled protocol π^*

Let Commit be a non-interactive commitment (cf. §2.2), and π be an n-party protocol with associated next-message functions $\psi^\pi = (\psi_1^\pi, \ldots, \psi_n^\pi)$. Further, let $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ be a non-interactive proof system (cf. §2.2) for the NP-relation R_π associated to the language:

$$L_\pi = \left\{ ((\gamma_i, m_i^{(r+1)}, \tau_i^{(r)}), (x_i, \omega_i, \delta_i)) : \ \gamma_i = \mathsf{Commit}(x_i || \omega_i; \delta_i) \wedge m_i^{(r+1)} = \psi^\pi(i, x_i, \omega_i, \tau_i^{(r)}) \right\}. \qquad (1)$$

The compiled protocol π^* proceed as follows.

Initialization: A trusted party samples $\sigma \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$ and distributes σ to each player.

Round $r = 0$: Party P_i samples the random tape ω_i required for a full run of π and broadcasts $\gamma_i = \mathsf{Commit}(x_i || \omega_i; \delta_i)$, where $\delta_i \leftarrow\!\!\$\ \{0,1\}^*$. Whenever P_i receives from $\mathsf{P}_{j \neq i}$ a commitment $\gamma_j = \gamma_i$, it outputs $\perp$ and stops the execution.

Round $r \geq 1$: During round r, each party P_i broadcasts a pair $(m_i^{(r)}, \phi_i^{(r)})$, where $m_i^{(r)}$ is its next message $m_i^{(r)} = \psi_i^\pi(x_i, \omega_i, \tau_i^{(r-1)})$, with $\tau_i^{(r-1)}$ being the transcript containing all π's messages received by P_i up to round $r-1$, and $\phi_i^{(r)} \leftarrow\!\!\$\ \mathsf{Prove}(\sigma, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), (x_i, \omega_i, \delta_i))$. Hence, each $\mathsf{P}_{j \neq i}$ checks that $\mathsf{Ver}(\sigma, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), \phi_i^{(r)}) = 1$; if not, it outputs $\perp$ and stops, and otherwise it proceeds to the next round.

Figure 1: Transformed protocol for in the presence of 1-rushing players.

Theorem 2. *Assume that* Commit *is perfectly binding and computationally hiding, that π securely computes f in the presence of semi-honest adversaries with malicious randomness (and aborts), and that* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ *is a simulation-extractable NIZK proof system for the relation R_π associated to the NP-language of Eq.* (1). *Then, π^* $(t, 1)$-securely computes f in the presence of malicious adversaries (with aborts).*

Remark 2 (Concrete instantiations)**.** *We can instantiate our compiler using the simulation-extractable NIZK proof system for NP from [SCO$^+$01], or simulation-extractable SNARKs [GM17] (under non-falsifiable assumptions).*

Proof of Thm. 2. For simplicity, we give the proof for the case of full security (without aborts). The proof for the case of security with aborts follows similarly. Let S be the PPT simulator guaranteed by (t, ∞)-security of π, and $(\mathsf{ZKSim}_0, \mathsf{ZKSim}_1)$, $(\mathsf{KExt}_0, \mathsf{KExt}_1)$ be the NIZK simulator and the knowledge extractor for $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$. Consider the following derived PPT simulator S^*:

- At the beginning, wait to receive the set $\mathcal{I}$ and the inputs $(x_i)_{i \in \mathcal{I}}$ for the corrupted players in the ideal execution.
- Sample $(\sigma, \zeta, \xi) \leftarrow\!\!\$\ \mathsf{KExt}_0(1^\lambda)$, and forward σ to each player controlled by A^*.
- For each $j \in [n]$ such that $j \notin \mathcal{I}$, compute $\gamma_j \leftarrow\!\!\$\ \mathsf{Commit}(0^\ell)$—where ℓ is the bit-length of the plaintext space for the commitment scheme—and forward $(\gamma_j)_{j \notin \mathcal{I}}$ to A^* on behalf of each honest player. Wait to receive $(\gamma_i)_{i \in \mathcal{I}}$ from A^*; if $\exists i, j \in [n]$ such that $\gamma_i = \gamma_j$, simulate A^* aborting and terminate.
- Emulate the rest of the protocol by running S on a uniformly random tape. In particular:
  - Whenever S outputs a simulated message $m_j^{(r)}$ on behalf of an honest player, forward $(m_j^{(r)}, \phi_j^{(r)})$ to A^*, where $\phi_j^{(r)} \leftarrow\!\!\$\ \mathsf{ZKSim}_1(\zeta, (\gamma_j, m_j^{(r)}, \tau_j^{(r-1)}))$ and $\tau_j^{(r-1)}$ is the simulated transcript consisting of all messages received by $\mathsf{P}_{j \notin \mathcal{I}}$ during the simulation up to round $r-1$.

– Upon receiving $(m_i^{(r)}, \phi_i^{(r)})$ from A^* on behalf of a corrupted party P_i, run $\mathsf{Ver}(\sigma, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1), \phi_i^{(r)}}), \phi_i^{(r)})$ and if the output is 0 simulate A^* aborting and terminate.
– Whenever A^* wants to reset the protocol execution at round $r \geq 2$, rewind the simulation to round r and continue from there using the same random tape for the simulator S.
– Finally, whenever S outputs $(x'_i)_{i \in \mathcal{I}}$, forward these values to the trusted party.

Towards proving the theorem, we consider a sequence of hybrid experiments and argue that each pair of hybrids is computationally close thanks to the properties of the underlying cryptographic primitives.

Hybrid $\mathbf{H}_0(\lambda)$: This is identical to $\mathbf{REAL}^1_{\pi^*, \mathsf{A}^*, \mathsf{D}^*}(\lambda)$.
Hybrid $\mathbf{H}_1(\lambda)$: Identical to the previous experiment, except that we now generate the CRS by running $(\sigma, \zeta) \leftarrow\!\!\$\ \mathsf{ZKSim}_0(1^\lambda)$; additionally, for each round $r > 0$ we now replace $\phi_i^{(r)}$ with a simulated[17] proof $\phi_i^{(r)} \leftarrow\!\!\$\ \mathsf{ZKSim}_1(\zeta, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}))$.
Hybrid $\mathbf{H}_2(\lambda)$: Identical to the previous experiment, except that we replace the commitments $(\gamma_j)_{j \notin \mathcal{I}}$ with $(\mathsf{Commit}(0^\ell))_{j \notin \mathcal{I}}$.
Hybrid $\mathbf{H}_3(\lambda)$: Identical to the previous experiment, except that we now generate the CRS by running $(\sigma, \zeta, \xi) \leftarrow\!\!\$\ \mathsf{KExt}_0(1^\lambda)$; additionally, every time the attacker sends a pair $(m_i^{(r)}, \phi_i^{(r)})$ such that the proof $\phi_i^{(r)}$ is accepting, we run $(x_i, \omega_i, \delta_i) \leftarrow\!\!\$\ \mathsf{KExt}_1(\xi, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), \phi_i^{(r)})$ and artificially abort in case $R_\pi((\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), (x_i, \omega_i, \delta_i)) = 0$.
Hybrid $\mathbf{H}_4(\lambda)$: This is identical to $\mathbf{IDEAL}_{f, \mathsf{S}^*, \mathsf{D}^*}(\lambda)$.

Lemma 1. $\{\mathbf{H}_0(\lambda)\}_{\lambda \in \mathbb{N}} \approx_c \{\mathbf{H}_1(\lambda)\}_{\lambda \in \mathbb{N}}$.

Proof. The proof is by a standard reduction to adaptive multi-theorem zero-knowledge of $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$. By contradiction, assume that there exists a PPT adversary A^* and a non-uniform PPT distinguisher D^* that can distinguish the two hybrids with non-negligible probability. Consider the following non-uniform PPT attacker A, playing the game of Def. 5.

- Receive the CRS σ from the challenger.
- Run $\mathsf{D}^*(\sigma)$ to obtain $\mathcal{I}$, $(x_i)_{i \in [n]}$, and $z \in \{0,1\}^*$, and forward $\mathcal{I}$, $(x_i)_{i \in \mathcal{I}}$, and z to A^*.
- Execute the protocol π^* of Fig. 1 with A^*, playing the role of the honest parties $\mathsf{P}_j(\sigma, x_j)$ for each $j \notin \mathcal{I}$, with the only difference that the proofs $\phi_j^{(r)}$ are obtained by forwarding $(\gamma_j, m_j^{(r)}, \tau_j^{(r-1)}), (x_j, \omega_j, \delta_j)$ to the target oracle (i.e., either $\mathsf{Prove}'(\sigma, \cdot, \cdot)$ or $\mathsf{ZKSim}'_1(\zeta, \cdot, \cdot)$).
- Output whatever D^* outputs.

By inspection, the simulation performed by A is perfect in the sense that when the challenger generates the CRS using Setup and computes the proofs using Prove, the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $\mathbf{H}_0(\lambda)$. Similarly, when the challenger generates the CRS using ZKSim_0 and computes the proofs using ZKSim_1, the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $\mathbf{H}_1(\lambda)$. Hence, A breaks adaptive multi-theorem zero knowledge with non-negligible probability, concluding the proof. □

Lemma 2. $\{\mathbf{H}_1(\lambda)\}_{\lambda \in \mathbb{N}} \approx_c \{\mathbf{H}_2(\lambda)\}_{\lambda \in \mathbb{N}}$.

Proof. The proof is by a standard reduction to the computational hiding property of Commit. By contradiction, assume that there exists a PPT adversary A^* and a non-uniform PPT distinguisher D^* that can distinguish the two hybrids with non-negligible probability. Consider the following non-uniform PPT attacker A, playing the game of Def. 4.

[17]Note that the simulator is proving a false statement.

- Generate $(\sigma, \zeta) \leftarrow\$ \mathsf{ZKSim}_0(1^\lambda)$, run $\mathsf{D}^*(\sigma)$ to obtain $\mathcal{I}$, $(x_i)_{i\in[n]}$, and $z \in \{0,1\}^*$, and forward $\mathcal{I}$, $(x_i)_{i\in\mathcal{I}}$, and z to A^*.
- Execute the protocol π^* with A^* exactly as described in $\mathbf{H}_1(\lambda)$, playing the role of the honest parties $\mathsf{P}_j(\sigma, x_j)$ for each $j \notin \mathcal{I}$, with the only difference that the commitments γ_j are obtained by forwarding $(x_j||\omega_j, 0^\ell)$ to the $\mathsf{LR}(\cdot,\cdot)$ oracle.
- Output whatever D^* outputs.

By inspection, the simulation performed by A is perfect in the sense that when the challenger generates the commitments using $x_j||\omega_j$, the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $\mathbf{H}_1(\lambda)$. Similarly, when the challenger generates the commitments using 0^ℓ, the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $\mathbf{H}_2(\lambda)$. Hence, A breaks the computational hiding property with non-negligible probability, concluding the proof. □

Lemma 3. $\{\mathbf{H}_2(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{\mathbf{H}_3(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. The proof is down to simulation extractability of $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$. Denote by $\mathbf{BAD}$ the event that an artificial abort happens in $\mathbf{H}_3(\lambda)$, namely there exists at least one round r in which the attacker forwards a valid proof $\phi_i^{(r)}$ for which the extractor fails to extract a valid witness w.r.t. the relation of Eq. (1). Since the two experiments are identical conditioned on $\mathbf{BAD}$ not happening, all we need to prove is that $\mathbb{P}[\mathbf{BAD}] \in \mathtt{negl}(\lambda)$. By contradiction, assume that there exists a PPT adversary A^* and a non-uniform PPT distinguisher D^* provoking event $\mathbf{BAD}$. Consider the following attacker A, playing the game of Def. 6.

- Receive the CRS σ from the challenger.
- Run $\mathsf{D}^*(\sigma)$ to obtain $\mathcal{I}$, $(x_i)_{i\in[n]}$, and $z \in \{0,1\}^*$, and forward $\mathcal{I}$, $(x_i)_{i\in\mathcal{I}}$, and z to A^*.
- Execute the protocol π^* with A^* exactly as described in $\mathbf{H}_2(\lambda)$, playing the role of the honest parties $\mathsf{P}_j(\sigma, x_j)$ for each $j \notin \mathcal{I}$, with the only difference that the proofs $\phi_j^{(r)}$ are obtained by forwarding $(\gamma_j, m_j^{(r)}, \tau_j^{(r-1)})$ to the $\mathsf{ZKSim}_1(\zeta, \cdot)$ oracle.
- At the end of the protocol execution, pick random $i^* \leftarrow\$ \mathcal{I}$ and $r^* \leftarrow\$ [r_{\mathsf{max}}]$, where r_{max} denotes the total number of rounds. Hence, return $((\gamma_{i^*}, m_{i^*}^{(r^*)}, \tau_{i^*}^{(r^*-1)}), \phi_{i^*}^{(r^*)})$ to the challenger.

By inspection, A perfectly emulates the view in an execution of $\mathbf{H}_2(\lambda)$. In fact, the challenger samples $(\sigma, \zeta, \xi) \leftarrow\$ \mathsf{KExt}_0(1^\lambda)$, where the distribution of (σ, ζ) is identical to that produced using $\mathsf{ZKSim}_0(1^\lambda)$. Moreover, honest proofs are simulated using $\mathsf{ZKSim}_1(\zeta, \cdot)$ exactly as done in $\mathbf{H}_2(\lambda)$. It follows, that there exists a polynomial $p(\lambda) \in \mathtt{poly}(\lambda)$ such that A^* as run by A provokes event $\mathbf{BAD}$ with probability at least $1/p(\lambda)$.

Finally, since the statement/proof pair for which event $\mathbf{BAD}$ is provoked does not belong to the list $\mathcal{Q}$ of simulated proofs—as the commitments output by A must be different from the ones computed by the honest players—A succeeds in breaking simulation extractability with probability at least $1/(t \cdot r_{\mathsf{max}}) \cdot 1/p(\lambda)$ which is still non-negligible. This finishes the proof. □

Lemma 4. $\{\mathbf{H}_3(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{\mathbf{H}_4(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. The proof is down to the 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 the two experiments with non-negligible probability. We build a PPT adversary A and a non-uniform PPT distinguisher D that can distinguish between $\mathbf{REAL}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ and $\mathbf{IDEAL}_{f,\mathsf{S},\mathsf{D}}(\lambda)$.

- At the beginning, D samples $(\sigma, \zeta, \xi) \leftarrow\$ \mathsf{KExt}_0(1^\lambda)$, runs $\mathsf{D}^*(\sigma)$ obtaining $\mathcal{I}$, $(x_i)_{i\in[n]}$, and $z \in \{0,1\}^*$, and outputs $\mathcal{I}$, $(x_i)_{i\in\mathcal{I}}$, and z.

- Attacker A runs $\mathsf{A}^*(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$ and interacts with it as follows:
  - At round $r = 0$, forward $(\mathsf{Commit}(0^\ell))_{j\notin\mathcal{I}}$ to A^* and wait to receive $(\gamma_i)_{i\in\mathcal{I}}$ from A^*. In case $\exists i, j \in [n]$ such that $\gamma_i = \gamma_j$, simulate A^* aborting and terminate.
  - At round $r \geq 1$, upon receiving a message $m_j^{(r)}$ from an honest player, forward $(m_j^{(r)}, \phi_j^{(r)})$ to A^*, where $\phi_j^{(r)} \leftarrow\!\!\$\ \mathsf{ZKSim}_1(\zeta, (\gamma_j, m_j^{(r)}, \tau_j^{(r-1)}))$. Whenever A^* forwards a pair $(m_i^{(r)}, \phi_i^{(r)})$, check that $\mathsf{Ver}(\sigma, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), \phi_i^{(r)}) = 1$ and if the verification succeeds run $(x_i, \omega_i, \delta_i) \leftarrow\!\!\$\ \mathsf{KExt}_1(\xi, (\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), \phi_i^{(r)})$ and check that $R_\pi((\gamma_i, m_i^{(r)}, \tau_i^{(r-1)}), (x_i, \omega_i, \delta_i)) = 1$. In case any of the above two checks fails, simulate A^* aborting and terminate, and otherwise output $m_i^{(r)}$.
  - Whenever A^* resets the protocol at round $r \geq 1$, repeat the simulation as described above starting from round r, i.e., running A^* on the same messages $m_j^{(r)}$, but using fresh randomness for computing the proofs $\phi_j^{(r)}$.
  - At the end of the last round, pass to D the same output that A^* would return given the above simulated view.
- Finally, D^* outputs the same guess as that of D.

By inspection, the view of D^* when interacting with A^* as emulated by A during a run of $\mathbf{REAL}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ is identical to that in $\mathbf{H}_3(\lambda)$. This is because, by the perfect binding property of the non-interactive commitment (cf. Def. 3), whenever A^* produces a proof $\phi_i^{(r)}$ that is accepting for $(\gamma_i, m_i^{(r)}, \tau_i^{(r-1)})$, the extracted input x_i and random tape ω_i must be identical to those contained in the initial commitments γ_i that A^* outputs at round $r = 0$, which also fixes the message $m_i^{(r)}$ to be the next message $\psi_i^\pi(x_i, \omega_i, \tau_i^{(r-1)})$ that P_i would output in an honest execution of protocol π, and the latter continues to hold after each reset takes place.

Analogously, the view of D^* when interacting with A^* as emulated by A during a run of $\mathbf{IDEAL}_{f,\mathsf{S},\mathsf{D}}(\lambda)$ is identical to that of $\mathbf{H}_4(\lambda) \equiv \mathbf{IDEAL}_{f,\mathsf{S}^*,\mathsf{D}^*}(\lambda)$. This concludes the proof of the lemma. □

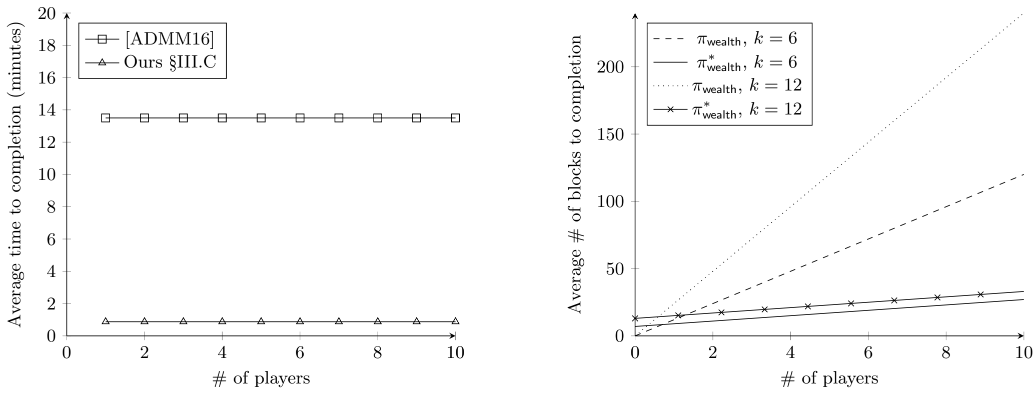
The theorem now follows directly by combining the above lemmas. □

Remark 3 (On simulation extractability)**.** *While our definition of simulation extractability (cf. Def. 6) requires straight-line extraction, the proof of Thm. 2 would go through even assuming that the extractor has black-box access to the adversary. This is because, when $c = 1$, the input of the adversary can not change as a consequence of a rewind, and thus a single extraction procedure is sufficient.*[18] *In such a case, we can instantiate the simulation-extractable NIZK in the random oracle model via the Fiat-Shamir transform [FS86, FKMV12].*

5 Efficiency Considerations

In this section, we analyze the efficiency of our protocols for what concerns both the time to completion and the communication complexity. In particular, we provide a prototype implementation of our coin tossing protocol from §3.3 using Ethereum smart contracts, and compare it to the classical construction by Andrychowicz *et al.* [ADMM14, ADMM16]. Additionally, we describe a concrete instantiation of our generic compiler from §4.2 in a specific setting, and analyze its performances.

[18]The situation is much more complicated when considering fully-fledged resettability, or even concurrent security, see, e.g., known negative results about concurrent zero knowledge [KPR98].



(a) Coin tossing (b) Generic compiler

Figure 2: Performances of our protocols in terms of time to completion. Protocol π_{wealth} is a toy smart contract for establishing the wealthier player, whereas π^*_{wealth} is the compiled version of the same protocol after applying our transform from §4.2.

5.1 Setting the Stage

Coin tossing. To allow for a fair comparison between our coin tossing protocol and the protocol from [ADMM14, ADMM16], we implemented both of them in Solidity using Ethereum smart contracts. See Fig. 3 on page 27 and Fig. 4 on page 28 in the appendix for the corresponding code. The VUF is instantiated using RSA-FDH [BR96].

Very roughly, the implementation of [ADMM14, ADMM16] works as follows. In the committing phase, the players trigger the `commit` function upon input the commitment to some value, and a deposit of `minDep` $= n(n-1)$. Hence, each player can open the commitment within the next 36 blocks.[19] At the end of this timeout, if a player did not send its opening, the other players can trigger the function `payDeposit` to receive the compensation back. After the commitment phase, the players trigger the function `putMoney` upon input a value-one coin to participate in the lottery; the pot can be claimed by the winner after all the participants opened their commitment, by triggering the `claimWinner` function of the smart contract.

Generic compiler. In order to analyze the efficiency of the compiler we describe a toy example along with its instantiation for the following simple n-party MPC protocol π_{wealth} to establish who is the wealthier:

1. Sequentially,[20] each player P_i publishes a commitment γ_i to its wealth x_i using randomness ω_i.
2. After all commitments are published, the players open their commitments in *reverse order* (i.e., P_i opens before P_j iff $i > j$).
3. Each player checks that all openings are correct, and if this is the case then outputs $\max_i x_i$.

The reason behind the ordering of the messages is to tackle malleability attacks. Indeed, if a player P_i commits after P_j, then the commitment of P_i could be related to the commitment of P_j. However, the fact that P_i has to open first, guarantees the independence of the two committed values, as otherwise P_i could break the hiding property of the commitment.

[19]Since the players cannot rush, in each phase of the protocol they need to wait 12 blocks before running the next phase, for a total of 36 blocks.

[20]Wlog. we can assume the order is identified by the index of each player.

By applying our generic compiler to the above construction, we obtain a protocol π^*_{wealth} that we describe in full in §A of the appendix. In particular, we instantiate the commitment with the scheme by Micciancio and Petrank [MP03] (based on the DDH assumption), and the simulation-extractable NIZK proof by applying the Fiat-Shamir transform (cf. Rmk. 3) to concrete Σ-protocols for the language of Eq. (1).

5.2 Time to Completion

Coin tossing. Fig. 2a shows that the average time to completion of our smart contract for coin tossing, as a function of the number of players, is less than 2 minutes. In contrast, the smart contract corresponding to the protocol in [ADMM14, ADMM16] takes about 15 minutes, on average, before completion. This discrepancy is due to the fact that in our protocol players can be rushing, while this is not possible in the protocol from [ADMM14, ADMM16] where the players need to wait 12 blocks before transitioning to the next stage of the protocol.

We run the experiment using the Ropsten testnet [tes]. Each point in the plot is averaged over 20 runs in order to smooth out the delays due to the network. In both implementations, we assume that the players start immediately running the protocol as soon as the corresponding smart contract is made available on the blockchain.

Generic compiler. Fig. 2b shows the minimum number of expected blocks that the players need to wait before protocols π_{wealth} and π^*_{wealth} terminate, as a function of the number n of players and for different values of the number k of blocks that are needed in order to be sure that a given transaction is confirmed on the ledger. Given that the players must be non-rushing, the original protocol π_{wealth} requires at least $2n \cdot k$ blocks before its completion. On the other hand, the compiled protocol π^*_{wealth} takes at least $1 + k + 2n$ blocks (i.e., one block for publishing in parallel all the initial commitments, k blocks for waiting that these values are confirmed on the chain, and finally $2n$ blocks for running the original protocol in the presence of rushing players).

5.3 Communication Complexity

Coin tossing. In the protocol from [ADMM14, ADMM16], we can instantiate the commitment heuristically via SHA-3-256 (i.e. the `commit` function outputs 256 bits and takes as input an arbitrary number of bits). In contrast, in our protocol the output size of the VUF is 4096 bit when using RSA-FDH at 80 bits of security. This can be improved by replacing RSA-FDH with the unique signature of Boneh, Lynn, and Shacham [BLS01], which yields signatures of size 160 bits at 80 bits of security.

Generic compiler. Regarding the communication complexity, we note that the compiler adds a constant overhead of $18 = O(1)$ group elements to the original protocol (i.e., 4 group elements for publishing the initial commitments, and 14 group elements for the NIZK proofs).

6 Conclusions

We have discussed attacks to smart contracts and on-chain MPC protocols on forking blockchains when honest players are rushing (i.e., they post the next message/transaction without waiting that the previous one is confirmed). In particular, our work uncovers that there are issues way different than the double-spending attack, and such issues might affect both security and privacy of the smart contract/MPC protocol in question. Indeed, on the negative side, we showed that a well-known MPC protocol based on smart contracts becomes insecure in presence of forks and rushing players because the adversary can play adaptively on a branch of a fork depending on the information observed on the other branch. Instead, on the positive side, we have shown smart contracts within on-chain MPC protocols (for both concrete and generic tasks) that remain secure even when there are forks and players rush. This, somehow, allows us to get the best of both worlds, i.e., having smart contracts that are both safe and fast.

Interesting avenues for future research include designing new ad-hoc protocols that remain secure in the presence of rushing players for other concrete settings of interest (as, e.g., decentralized poker [KMB15, BKM17]), and investigating a composable treatment of forking blockchains and their applications to fair MPC with penalties [KZZ16, BMTZ17]. Indeed, all those very useful prior works rely on players waiting for confirmations, and as such bring a significant price to pay in terms of efficiency.

References

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE, 2014.

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

[BGM$^+$18] Christian Badertscher, Juan A. Garay, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. But why does it work? A rational protocol design treatment of bitcoin. In *EUROCRYPT*, pages 34–65, 2018.

[Bit17] Nir Bitansky. Verifiable random functions from non-interactive witness-indistinguishable proofs. In *TCC*, pages 567–594, 2017.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, pages 421–439, 2014.

[BKM17] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In *ASIACRYPT*, pages 410–440, 2017.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, pages 514–532, 2001.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, pages 324–356, 2017.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with RSA and Rabin. In *EUROCRYPT*, pages 399–416, 1996.

[BZ17] Massimo Bartoletti and Roberto Zunino. Constant-deposit multiparty lotteries on bitcoin. In *Financial Cryptography and Data Security*, pages 231–247, 2017.

[CGJ19] Arka Rai Choudhuri, Vipul Goyal, and Abhishek Jain. Founding secure computation on blockchains. In *EUROCRYPT*, pages 351–380, 2019.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *ACM STOC*, pages 364–369, 1986.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, pages 139–147, 1992.

[Dod03] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In *PKC*, pages 1–17, 2003.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *PKC*, pages 416–431, 2005.

[FKMV12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the fiat-shamir transform. In *INDOCRYPT*, pages 60–79, 2012.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[GG17] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In *TCC*, pages 529–561, 2017.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.

[GM11] Vipul Goyal and Hemanta K. Maji. Stateless cryptographic protocols. In *IEEE FOCS*, pages 678–687, 2011.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In *CRYPTO*, pages 581–612, 2017.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *ASIACRYPT*, pages 444–459, 2006.

[GS09] Vipul Goyal and Amit Sahai. Resettably secure computation. In *EUROCRYPT*, pages 54–71, 2009.

[HJ16] Dennis Hofheinz and Tibor Jager. Verifiable random functions from standard assumptions. In *TCC*, pages 336–362, 2016.

[Jag15] Tibor Jager. Verifiable random functions from weaker assumptions. In *TCC*, pages 121–143, 2015.

[KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *ACM CCS*, pages 30–41, 2014.

[KB16] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *ACM CCS*, pages 418–429, 2016.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *ACM CCS*, pages 195–206, 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 S&P*, pages 839–858, 2016.

[KPR98] Joe Kilian, Erez Petrank, and Charles Rackoff. Lower bounds for zero knowledge on the internet. In *IEEE FOCS*, pages 484–492, 1998.

[KVV16] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In *ACM CCS*, pages 406–417, 2016.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT*, pages 705–734, 2016.

[Lys02] Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *CRYPTO*, pages 597–612, 2002.

[MP03] Daniele Micciancio and Erez Petrank. Simulatable commitments and efficient concurrent zero-knowledge. In *EUROCRYPT*, pages 140–159, 2003.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *IEEE FOCS*, pages 120–130, 1999.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[SCO$^+$01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In *CRYPTO*, pages 566–598, 2001.

[SSV19] Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Publicly verifiable proofs from blockchains. In *PKC*, pages 374–401, 2019.

[tes] Etherscan. `https://ropsten.etherscan.io`. Block Explorer and Analytics Platform for Ethereum.

[Woo19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical Report aeeda84, July 2019. Byzantium version.

A Instantiating the Compiler

We describe a concrete instantiation for the protocol generated by our generic compiler on input the protocol π_{wealth} for establishing the wealthiest party (cf. §5.1).

A.1 Description

Let $(\mathbb{G}, \cdot)$ be a cyclic group with prime order q, and let g, h be two random generators of $\mathbb{G}$. In what follows, we assume that $(\mathbb{G}, g, h, q)$ are publicly known.[21] For simplicity we will omit the modulus from the computations described below.

In order to commit to $x \in \mathbb{Z}_q$, the sender picks randomness $\omega \in \mathbb{Z}_q$ and outputs $\gamma = (g^{\omega}, h^{x+\omega})$. The receiver, given $\gamma = (\gamma_0, \gamma_1)$, and upon receiving (x, ω), checks that $\gamma_0 = g^{\omega}$, $\gamma_1 = h^{x+\omega}$. This scheme satisfies perfect binding and computational hiding (under the DDH assumption) [MP03].

[21] In the scheme from [MP03] all the above parameters are generated as part of the commitment; however, we can include these values in a common reference string if available. We can also get these parameters from a call to a random oracle, since they can all be extracted from a random string.

Let us now describe the output of the generic compiler from Fig. 1 on input π_{wealth}. At the beginning, each party P_i commits to its input $x_i \in \mathbb{Z}_q$ and random tape $\omega_i \in \mathbb{Z}_q$ by publishing γ_i', γ_i'', where $\gamma_i' = (\gamma_{0,i}', \gamma_{1,i}') = (g^{\omega_i'}, h^{x_i+\omega_i'})$ and $\gamma_i'' = (\gamma_{0,i}'', \gamma_{1,i}'') = (g^{\omega_i''}, h^{\omega_i+\omega_i''})$. Then, when P_i computes the commitment of π_{wealth} he publishes $\gamma_i = (\gamma_{0,i}, \gamma_{1,i}) = (g^{\omega_i}, h^{x_i+\omega_i})$ and then, as required by the compiler, it must additionally provide a simulatable-extractable NIZK proof that the message and randomness of this last commitment correspond to the input x_i and randomness ω_i to which he committed at the beginning of the protocol. In the interactive setting, the latter can be done efficiently using the well-known Σ-protocol for proving equality of discrete logs. Indeed, recall that the compiler requires to prove here that the committed message of π_{wealth} corresponds to the message committed initially. In order to do so, the prover divides $\gamma_{1,i}'$ by $\gamma_{1,i}$, obtaining $h^{\omega_i'-\omega_i}$ and $\gamma_{0,i}'$ by $\gamma_{0,i}$, obtaining $g^{\omega_i'-\omega_i}$. The proof that the two discrete logarithms to bases g and h are equal implies that to construct $\gamma_{1,i}'$ and $\gamma_{1,i}$ the prover used the same x_i. A similar trick can be done to prove consistency of the randomness and we omit this description.

Moreover, we also add Schnorr's Σ-protocol [Sch91] to prove knowledge of the discrete logarithms of γ_i. This is required to make sure that extraction will provide the committed message, as in the relation specified by the compiler. Obviously, the above composition of Σ-protocols is then transformed into a simulatable-extractable NIZK proof via the Fiat-Shamir transform.

Similarly, when each player P_i opens her commitment by revealing (x_i, ω_i) there will be a simulatable-extractable NIZK proof proving that these values are consistent with the commitments played in the first round of π_{wealth}. Obviously, the opening of a commitment has already a verification procedure, but we keep such NIZK proof as the instructions of the compiler prescribe it.[22]

A.2 Security Analysis

We give now a sketch of the security proof of the above instantiated protocol. Indeed, since π_{wealth} is not a secure protocol under the simulation paradigm, we can not directly rely on the security guaranteed by the generic compiler. Let us start by recalling the ideal world functionality. In the ideal world, a trusted party receives the inputs of all players, and then sends all of them to everybody. Consider the following PPT (ideal-world adversary) simulator S^*:

1. Run an internal simulation of the real-world protocol by first sending two commitments to the all-zeroes string on behalf of all honest players, and receive all commitments of the parties controlled by the adversary.
2. Send the first round of π_{wealth} by committing again to the all-zeroes string on behalf of all honest players, and moreover send a simulated NIZK proof for each honest player and wait for all commitments and NIZK proofs of the corrupted players.
3. Extract the inputs of the corrupted players by using the extractor of the NIZK (notice that the entire extraction process is efficient since it requires to run one extraction procedure per corrupted player, and future rewinds do not affect the already extracted input).
4. Play in the ideal world the inputs of the adversary extracted during the previous step.
5. Get the output of the trusted functionality and identify the inputs of the honest players in the ideal world.
6. Rewind the simulation in order to play again the commitments of π_{wealth}, but this time commit to the same inputs used by the honest players in the ideal world; moreover,

[22]Since the compiler is black-box, when applying all the steps of the protocol some optimizations are of course possible.

compute the corresponding NIZK proofs using the NIZK simulator.

7. Send the opening of the commitments to the inputs of the honest ideal-world players along with the corresponding NIZK proofs computed using the NIZK simulator.
8. Output whatever the adversary outputs.

The indistinguishability of the ideal-world experiment from the real-world experiment can be proven through standard hybrid arguments. In particular, consider the experiments described below.

Hybrid $\mathbf{H}_0(\lambda)$: Identical to the real-world experiment.

Hybrid $\mathbf{H}_1(\lambda)$: We replace the second NIZK computed by honest players with a simulated one. Clearly, the indistinguishability comes from the zero-knowledge property guaranteed by the simulator of the NIZK proof.

Hybrid $\mathbf{H}_2(\lambda)$: We replace also the first NIZK computed by honest players with a simulated one. Indistinguishability from the previous hybrid follows again by zero knowledge.

Hybrid $\mathbf{H}_3(\lambda)$: We run the extractor in order to obtain the inputs of the corrupted players (as in step 3 of the simulator). Indistinguishability from the previous hybrid follows by the proof of knowledge property of the NIZK proof obtained via the Fiat-Shamir transform.

Hybrid $\mathbf{H}_4(\lambda)$: After extracting the inputs of the corrupted players, the commitment phase of π_{wealth} for all honest players is repeated until the adversary completes again the commitments and NIZK proofs of the corrupted players. Indistinguishability from the previous hybrid comes from the fact that the adversary completes an experiment with non-negligible probability and thus after sufficiently many repetitions she will play commitments and NIZK proofs again.

Hybrid $\mathbf{H}_5(\lambda)$: When the commitments of the honest players are played for the first time in π_{wealth}, we replace them with commitments to the all-zeroes string. Indistinguishability from the previous hybrid follows from the hiding property of the commitment scheme.

Hybrid $\mathbf{H}_6(\lambda)$: We replace the first commitments of the protocol (i.e., the ones introduced by the compiler) computed by honest players with commitments to the all-zeroes string. Indistinguishability from the previous hybrid comes again from the hiding property of the commitment scheme.

Hybrid $\mathbf{H}_7(\lambda)$: This experiment starts with honest players having as input the all-zeroes string, but before repeating the commitment phase of π_{wealth} the extracted inputs of the corrupted players are played in the ideal world obtaining the inputs of the honest ideal-world players. These values are then used when the commitments of π_{wealth} are computed during repetitions. Indistinguishability from the previous hybrid comes from the fact that in both the ideal and the real world experiments the honest players run with the same inputs.

The proof now follows by observing that $\mathbf{H}_7(\lambda)$ corresponds to the ideal-world experiment.

```
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;

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

  //common input of the VUF
  uint VUFmessage;

  function beginCoinTossing()  {
    sid = ...; //generate a session id
  }
  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
    }
  function claim(uint rand, uint proof) public {
    require (claimPhase && now < time2 && players[msg.sender].isPlaying && !players[msg.sender].
        hasClaimed && VUFCheck(VUFmessage,rand,proof,players[msg.sender].pk));
    Player p = players[msg.sender];
    p.c = rand;
    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;
    VUFMessage = sha3(players]playersAddr[0]].pk||...||players[playersAddr[n-1]].pk||sid);
    claimPhase = true;
    }
  function checkClaimed() public { //if the second timestamp has passed and some player didn't redeem,
      penalize the players
    require(claimPhase && now >= time2);
    for (uint i = 0; i < playersAddr.length; i++) {
      address pAddr = playersAddr[i];
      if (players[pAddr].c == 0)
        penalize(pAddr);
      }
    }

  //local functions
  function penalize(address penalized) private {  //penalize dishonest players by sending the
      compensation
    for (uint j = 0; j < playersAddr.length; i++) {
      address pAddr = playersAddr[i];
      uint n = playersAddr.length;
      if (pAddr != penalized) pAddr.transfer(minDep/n);
      }
    }
}
```

Figure 3: Pseudocode implementation of our smart contract for realizing parallel coin tossing. For simplicity, we omit an explicit definition of the `VRFCheck` function and of the concatenation function in the computation of `VRFMessage`.

```
pragma solidity >=0.4.21 <0.6.0;

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

constructor(address[] _addresses) 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 = addresses[i];
    p.isPlaying = true;
    p.hasCommitted = false;
    p.index = i;
    }
  n = addresses.length;
  }
  function commit(bytes32 _com) public payable { //sha3 value commit
    require(msg.value >= (addresses.length)*(addresses.length-1)) &&   players[msg.sender].isPlaying && !players[msg.sender].hasCommitted);
    Player p = players[msg.sender];
    p.com = _com;
    p.hasCommitted = true;
    p.balance = msg.value;
    allBalance += msg.value;
    init = block.number;
  }
  function openCom(int openVar) public  { //opening of the commitment
    require(players[msg.sender].hasCommitted && allBalance >= n*(n-1) && block.number < (init+28) && !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 && block.number >= (init+36));
    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 putMoney() public payable { //function for betting
    require (msg.value == 1 && players[msg.sender].hasCommitted);
    players[msg.sender].isBetting = true;
    allBalance += msg.value;
  }
  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
}
//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 4: Pseudocode implementation of the lottery protocol by Andrychowicz *et al.* [ADMM16], when using smart contracts.