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

Vincenzo Botta
DIEM
University of Salerno

Daniele Friolo
Sapienza
University of Rome

Daniele Venturi
Sapienza
University of Rome

Ivan Visconti
DIEM
University of Salerno

Abstract use of a Blockchain

- There is a decentralized list of blocks that can be extended
- The past is immutable
- Eventually, all transactions are added to a block

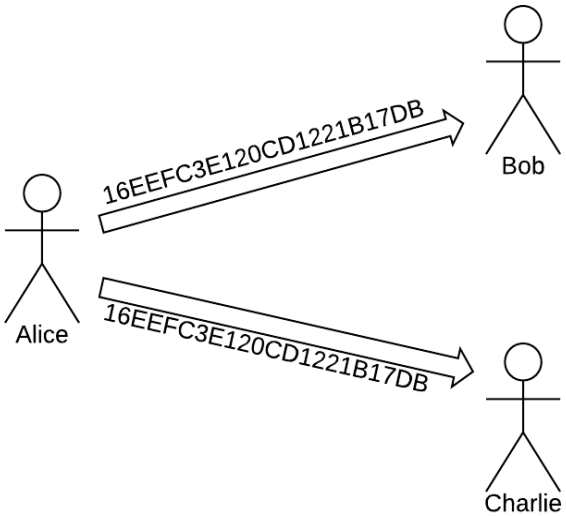
Many scientific results follow this abstraction and many implementations assume that the specific Blockchain used behaves like that

However, reality can be different…

Double Spending Attack

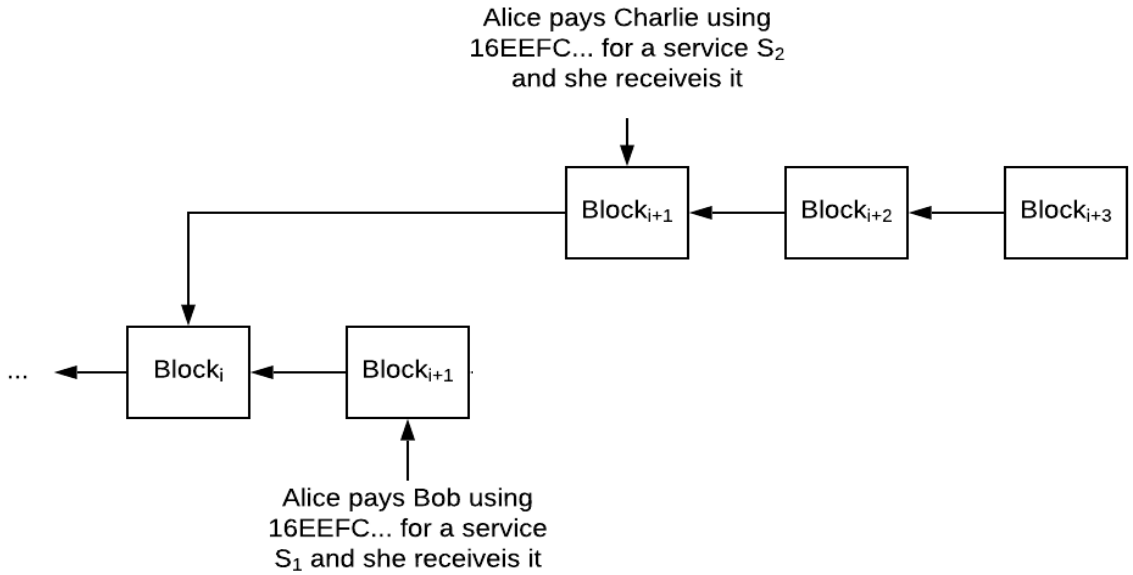
Double spending attack:

same money used in different transactions

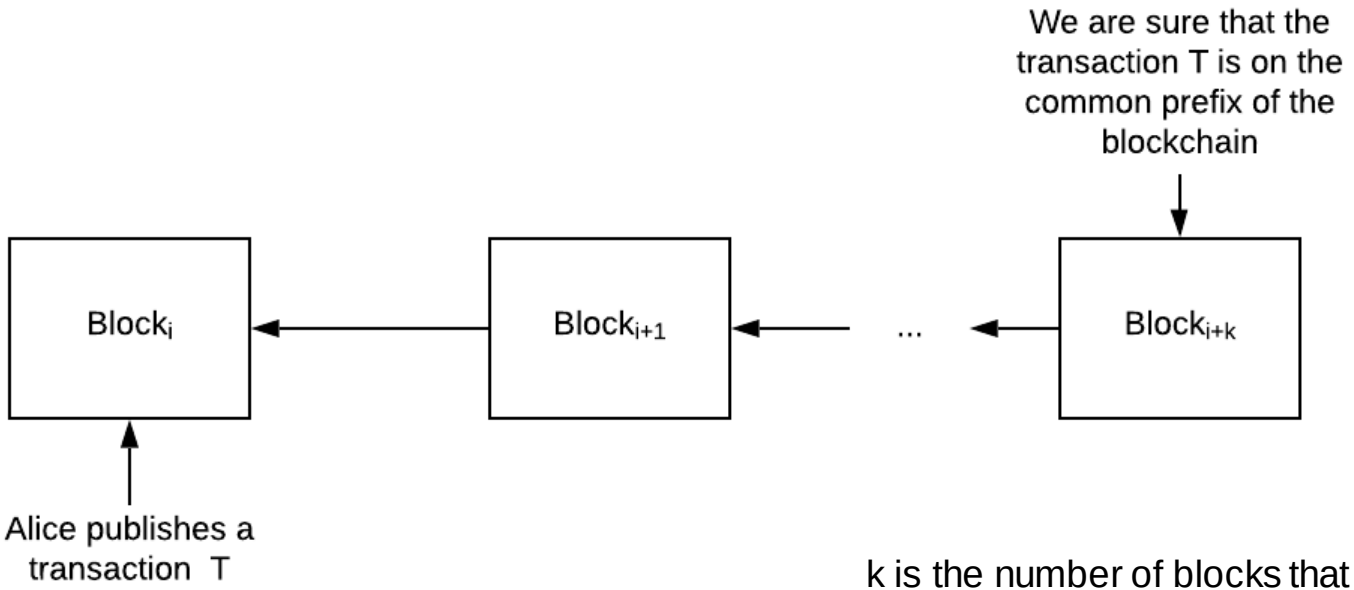


Forks and Double Spending Attack

Double spending attack: the attacker pays for an off-chain service using a transaction that later on will be discarded by the blockchain



Forks and Double Spending Attack



k is the number of blocks that we have to wait to be sure that the transaction is confirmed on the blockchain

Forks and Double Spending Attack

The classic solution to the double spending attack:

*the receiver of the payment has to **wait** until the associated transaction is confirmed on the blockchain and after that she can provide the off-chain service*

This solution introduces the problem of huge delays to confirm transactions in forking blockchains

Forks and Correlated On-Chain Transactions

A different scenario:

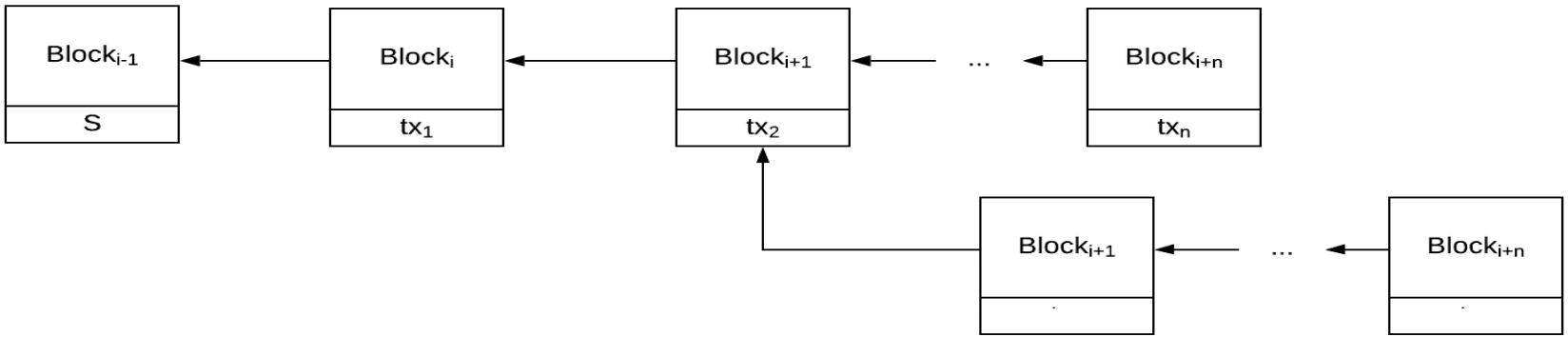
On-chain transactions can be correlated by having a common state and a sequential logic; invalidating one of them invalidates all the subsequent correlated transactions (e.g., consider transactions that monitor a good in a supply chain)

Smart contracts can be used to updated the state

Forks and Correlated On-Chain Transactions

S is a smart contract

S goes from state s_i to state s_{i+1} when the transaction tx_i is published on the ledger



Do we need to wait for confirmations in such scenarios?

Rushing Players and Smart Contracts

The confirmation time is too long => maybe players can rush!

The damage of double spending attacks is not replicated by such smart contracts: therefore one can think of running such smart contracts more efficiently, without waiting until transactions are confirmed

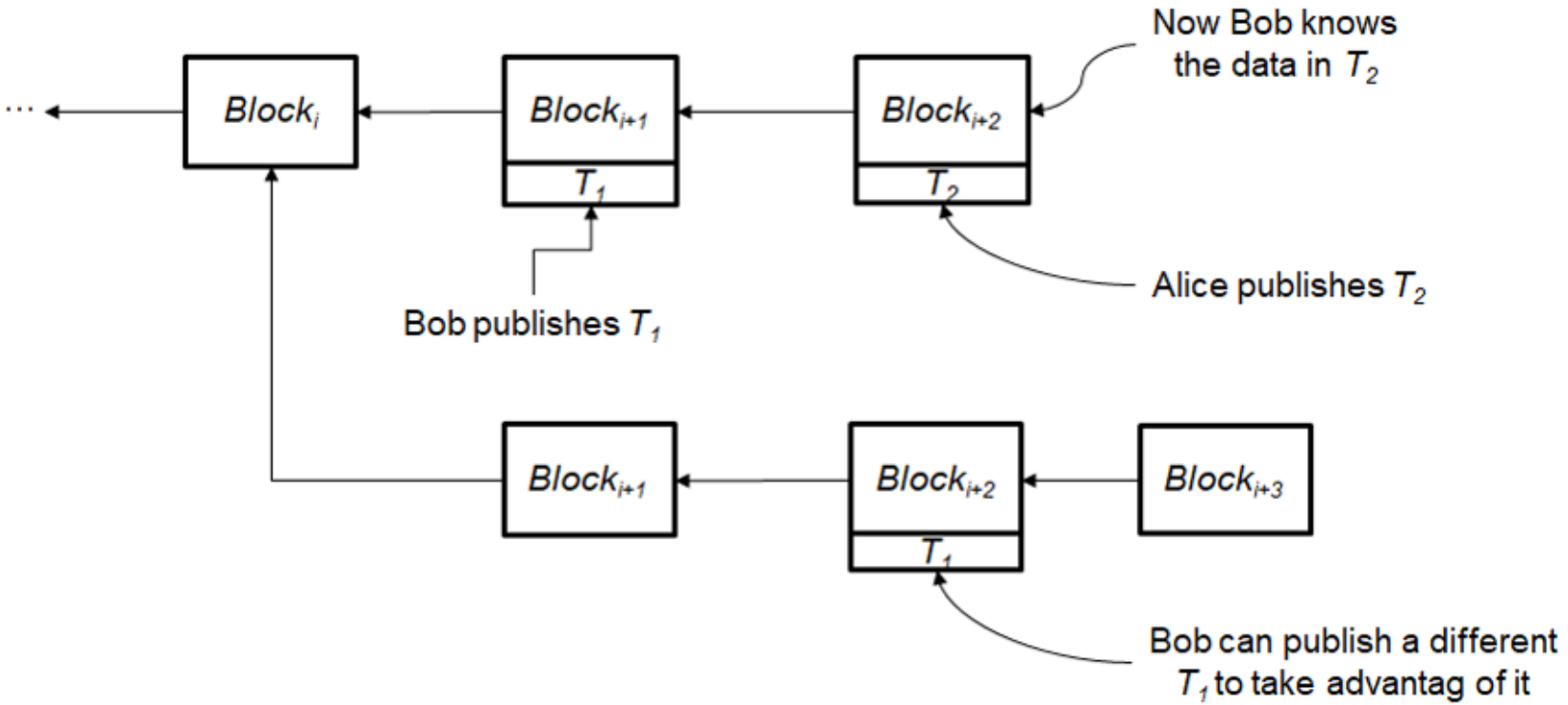
In other words, apparently players **could safely rush**

Rushing Players and Possible Attacks

Let’s look at it carefully: as soon as a transaction T1 from Bob appears on the blockchain, Alice answers with a transaction T2

If Bob sees a rushed T2 in a branch, then he can adaptively choose another value for T1 to play in another branch leveraging on prior knowledge of T2

Rushing Players and Possible Attacks



Concrete Example of an Attack

Coin tossing example: Alice and Bob want to establish a random string using a smart contract; they want to make sure that unhappy players don’t leave the game

Preliminaries (Commitment and NIZK)

A commitment scheme allows a sender to encode a message m so that it can be revealed later
The encoding hides m, and there is no way to decode to something different than m

A non-interactive zero-knowledge (NIZK) proof is string proving that an assertion is correct, without revealing why

Preliminaries (Hash Functions and VUFs)

H denotes a cryptographic hash function heuristically behaving as a random oracle

A verifiable unpredictable function (VUF) is a public-key cryptographic tool

- One can generate a key pair (pk,sk)
- One can compute Eval(sk,m) obtaining in output strings (r, φ)
- r could not have been predicted without knowing sk
- φ is a proof that r is the only correct (unique) output according to pk

Andrychowicz, Dziembowski, Malinowski, Mazurek: (IEEE S&P 14)

There are n players $P_1, \ldots, P_n$

Each P_i samples a random string (of random length) s_i

They evaluate $w = |s_1| + \ldots + |s_n| \bmod n$

Solution from ADMM14

Main ideas:

All P_i commit to their s_i and publish the commitments in the blockchain using commit transactions

All parties put money in a compute transaction that given all the s_i evaluates w

All parties open the commitments using an opening transaction

Solution from ADMM14

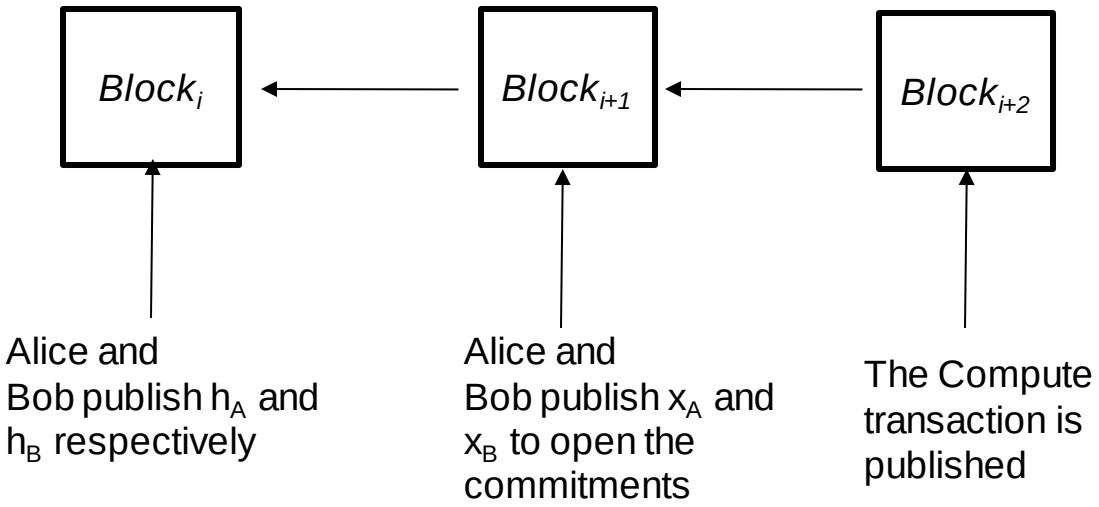
Commit transaction:
The value to commit is x_i, the commitment is generated computing $h_i = H(x_i)$

Each player P_i prepares n-1 commit transactions that include h_i and correspond to deposits of coins, and adds other n - 1 transactions that give the right to each other player P_j to redeem one of the deposits of P_i after some time t

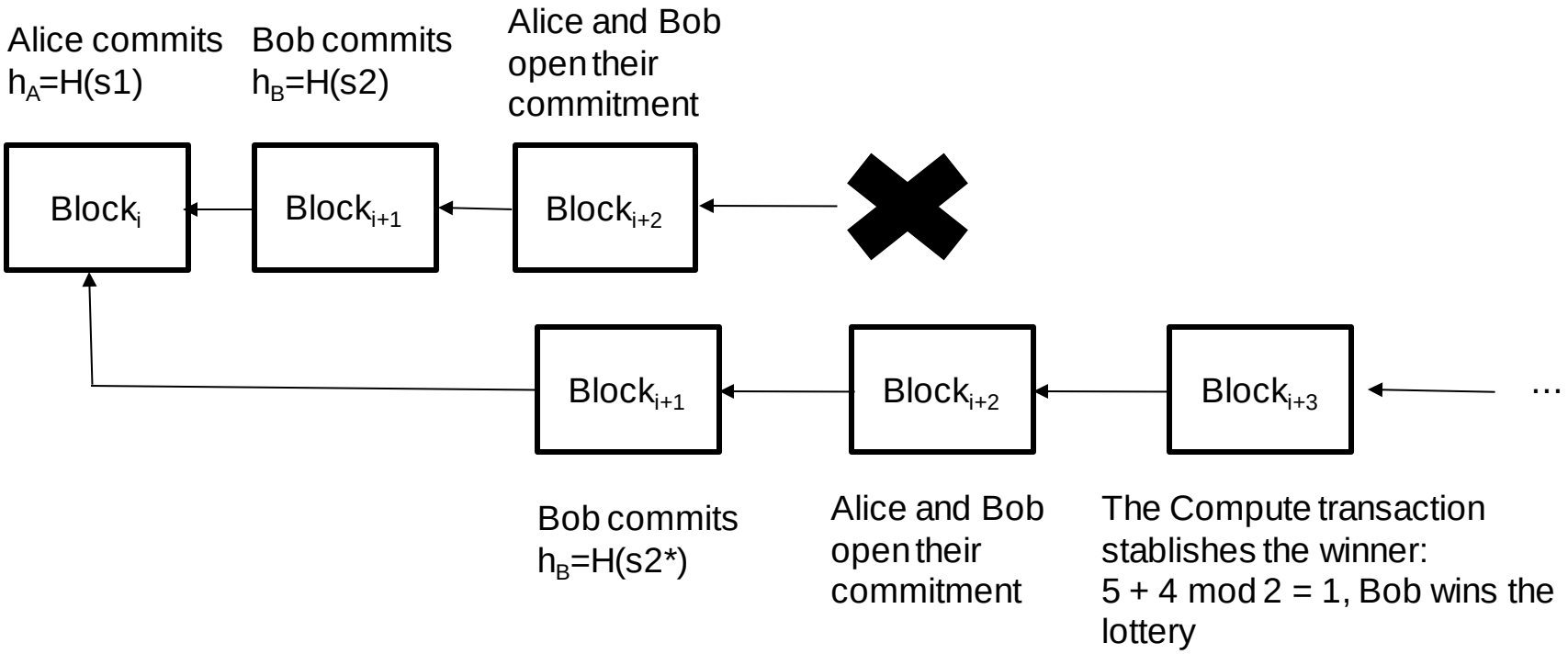
The commit transactions are redeemed by P_i if she shows x_i such that $H(x_i) = h_i$ before time t, otherwise they are redeemed by other players

Solution from ADMM14

An example with two players, Alice and Bob



ADMM14: Our Attack with Rushing Players



ADMM14: Our Analysis with Rushing Players

We show that ADMM14 can’t have both efficiency and security at the same time

Is this intrinsic?

Our Solution

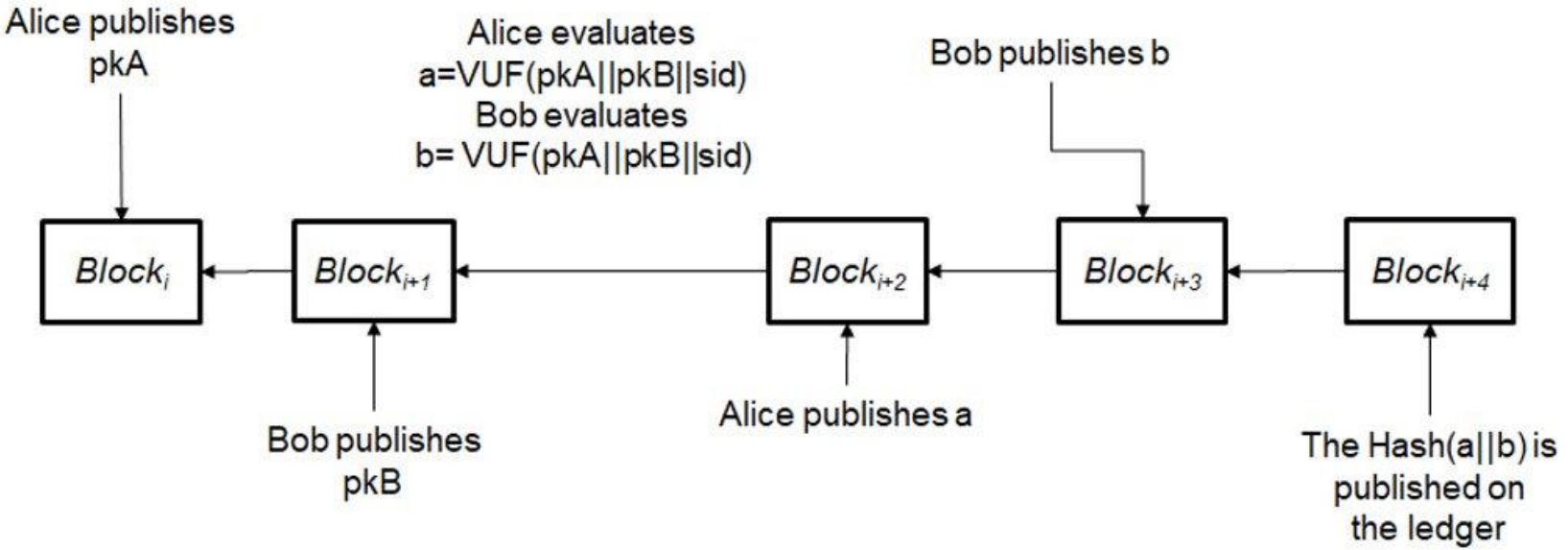
Alice owns (sk_A, pk_A), Bob owns (sk_B, pk_B) of a VUF

They want to evaluate $\text{Hash}(\text{VUF}_{skA}(pk_A||pk_B)||\text{VUF}_{skB}(pk_A||pk_B))$

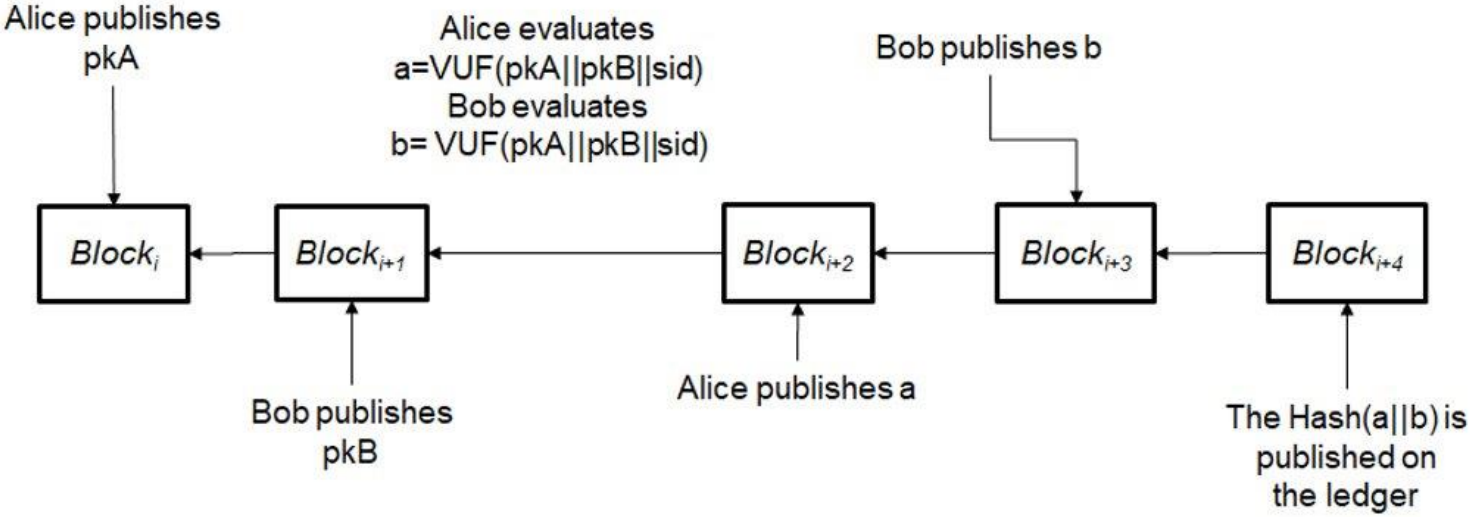
Our Solution

- One player (any) creates a smart contract with a session id
- Each participant with a public key deposits d coins in the smart contract
- Each participant computes $x = \text{Hash}(pk_1 || \ldots || pk_n || sid)$
- Using a VUF the i-th player computes $(y_i , \varphi_i) = \text{VUF}(sk_i, x)$ where y_i is unpredictable and φ_i is the proof that y_i is evaluated using x and sk_i
- Each participant publishes (y_i, φ_i) on the smart contract
- The smart contract outputs $\text{Hash}(y_1 || \ldots || y_n)$

Our Solution



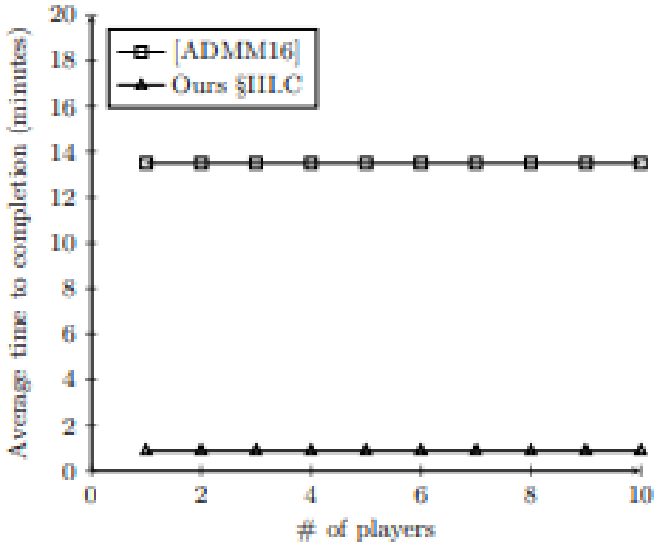
Our Solution



If an adversary changes something in the bottom branch (es., the public key of the VUF) then the honest player computes another unpredictable value, and the random oracle Hash will compute a random string.

Performance Analysis

ADMM14 vs our protocol



A Generic Compiler

We show a generic compiler that on input a classical secure MPC protocol (i.e., an MPC protocol without setup and players connected by point-to-point channels) outputs an MPC protocol that can be safely played on a blockchain allowing rushing, even when the blockchain forks.

The compiler builds on a new notion that we call *fork-resilient secure computation* and that is inspired by the notion of *resettably secure computation*.

This has an impact on smart contracts that using our techniques can be resilient to forks and efficient by design.

Toy Example: Wealth

There are n players P_1, …, P_n and they want to know who is the richerst player.

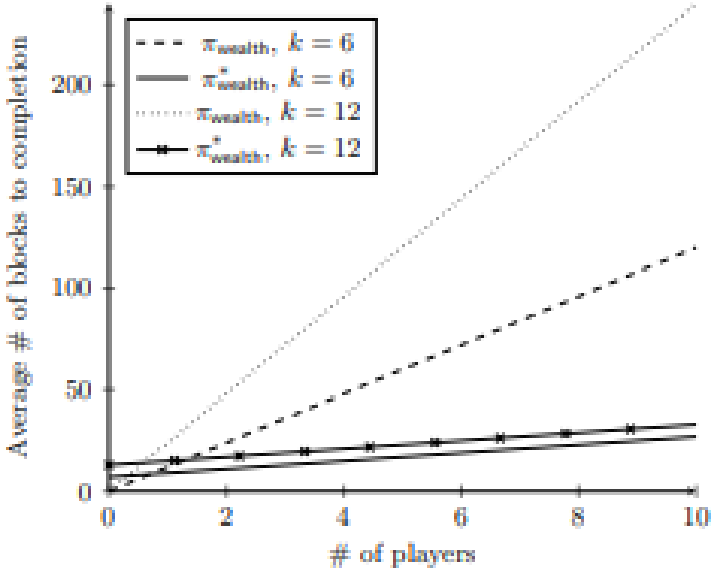
All players publish the commitment to their wealth on the blockchain.

They open the commitments in inverse order (to avoid computation of related commitments – this is a malleability problem, sorry for brevity!)

They have to wait that the previous transaction is confirmed on the blockchain before publishing another transaction

Efficiency of the compiled protocol wealth

We obtain the following results



π is the original protocol

π^* is our compiled version of π

k is the number of blocks to wait until the transaction is confirmed on the blockchain

Conclusion

- Designing fast smart contracts can be very non-trivial when some forms of security and privacy are desired
- Knowledge of technical details of the underlying blockchain is essential to avoid issues that often are hidden under the rug
- Knowledge of advanced cryptographic notions and constructions can highly improve the quality of applications of Blockchain Technology