FairSwap: How to Fairly Exchange Digital Goods

Stefan Dziembowski
Institute of Informatics
University of Warsaw, Poland
stefan.dziembowski@crypto.edu.pl

Lisa Eckey
Department of Computer Science
TU Darmstadt, Germany
lisa.eckey@cs.tu-darmstadt.de

Sebastian Faust
Department of Computer Science
TU Darmstadt, Germany
sebastian.faust@cs.tu-darmstadt.de

ABSTRACT

We introduce FairSwap – an efficient protocol for fair exchange of digital goods using smart contracts. A fair exchange protocol allows a sender $\mathcal{S}$ to sell a digital commodity x for a fixed price p to a receiver $\mathcal{R}$. The protocol is said to be secure if $\mathcal{R}$ only pays if he receives the correct x. Our solution guarantees fairness by relying on smart contracts executed over decentralized cryptocurrencies, where the contract takes the role of an external judge that completes the exchange in case of disagreement. While in the past there have been several proposals for building fair exchange protocols over cryptocurrencies, our solution has two distinctive features that makes it particular attractive when users deal with large commodities. These advantages are: (1) minimizing the cost for running the smart contract on the blockchain, and (2) avoiding expensive cryptographic tools such as zero-knowledge proofs. In addition to our new protocols, we provide formal security definitions for smart contract based fair exchange, and prove security of our construction. Finally, we illustrate several applications of our basic protocol and evaluate practicality of our approach via a prototype implementation for fairly selling large files over the cryptocurrency Ethereum.

ACM Reference Format:
Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. 2018. FairSwap: How to Fairly Exchange Digital Goods. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243857

1 INTRODUCTION

Consider a setting where a receiver $\mathcal{R}$ wishes to buy a digital commodity x from a sender $\mathcal{S}$. The receiver is willing to pay price p for x if it satisfies some predicate ϕ, i.e., if $\phi(x) = 1$. For instance, x may be a file (e.g., some movie) and ϕ outputs 1 if hashing x results into some fixed value h (i.e., $H(x) = h$). Suppose that the parties wish to execute the exchange over the Internet, where $\mathcal{R}$ and $\mathcal{S}$ do not trust each other. A fundamental problem that arises in this setting is how to guarantee that the exchange is executed in a fair way. That is, how can we make sure that $\mathcal{S}$ receives the payment when he delivers x to $\mathcal{R}$ such that $\phi(x) = 1$; and similarly, how can we guarantee that $\mathcal{R}$ only needs to pay the money if x is indeed correct. Unfortunately, it has been shown that without further assumptions it is impossible to design protocols that guarantee such strong fairness properties [40]. A simple way to circumvent this impossibility is to introduce a trusted middleman – in practice often referred to as an escrow service [33] – which waits to receive the money from $\mathcal{R}$ and the commodity x from $\mathcal{S}$, and only executes the exchange if $\phi(x) = 1$ is satisfied. Unfortunately such fully trusted middleman are often not available (e.g., there are countless examples of so-called "bogus" escrow services), or if available, this serive is very costly.

A promising alternative for implementing escrow services over the Internet is offered by decentralized cryptographic currencies. Cryptocurrencies replace the trusted middleman by a distributed network, where so-called miners maintain a shared data structure – the blockchain – storing the transactions of the system. In addition to simple payments many cryptocurrencies offer more complex transactions, often referred to as *smart contracts*, that allow users to carry out payments depending on the execution of a program. Using smart contracts one can easily design a straightforward solution for the problem of securely selling digital goods over the Internet. In an initial step the two parties $\mathcal{R}$ and $\mathcal{S}$ set up a contract, where $\mathcal{R}$ blocks p coins in the underlying cryptocurrency which guarantees the payment if $\mathcal{S}$ (within some time frame) sends a witness x to the contract such that $\phi(x) = 1$. Once the contract is deployed on the network, either $\mathcal{S}$ can trigger the payment of p coins by publishing x to the contract, or $\mathcal{R}$ can get his coins back after the timeout has passed. The underlying consensus mechanism of the cryptocurrency guarantees that the money transfer is only executed if the conditions are met, i.e., if x satisfies the predicate ϕ, or the refund is valid.

While the above smart contract example achieves fairness from the point of view of $\mathcal{S}$ and $\mathcal{R}$, it has an important drawback if x is large. Since in cryptocurrencies users pay fees to the miners for every step of executing a smart contract, storing and computing complex instructions results into high costs due to fees. For instance, in the cryptocurrency Ethereum, which offers rich support for smart contracts, the amount of *gas* (the currency unit used in Ethereum to pay fees) paid for executing the smart contract strongly depends on two factors: (a) the complexity of the program ϕ and (b) the size of x. Concretely, for storing a value x of size 1 MB in Ethereum the parties would need to pay more than USD 500 in transaction fees[1].

An appealing solution to the above problem called zero knowledge contingent payments (ZKCP) has been proposed in [46]. ZKCP protocols use zero knowledge proofs [24], which guarantee that a prover can convince a verifier of the correctness of a statement, e.g., $\phi(x) = 1$, without revealing the witness x to the verifier. More precisely a ZKCP protocol between $\mathcal{S}$ and $\mathcal{R}$ works as follows: First,

[1] Considering an exchange rate of 500 USD/Eth and a gas price of 3 Gwei.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243857

$\mathcal{S}$ encrypts x with key k. Moreover, it computes a commitment $c = H(k)$ (where H is a hash function) and a zero knowledge proof showing that computation of the ciphertext and the commitment was indeed done with a witness x which satisfies $\phi(x) = 1$. Next, $\mathcal{S}$ sends the ciphertext, the hash and the zero knowledge proof to $\mathcal{R}$, who verifies the correctness of the zero knowledge proof, and deploys a smart contract that pays p coins to $\mathcal{S}$ when the key is published. All the contract needs to do is then to verify that $h = H(k)$. It can be shown that if the underlying cryptographic primitives are secure, then the ZKCP smart contract scheme realizes a fair exchange protocol.

From a financial point of view the ZKCP protocol is very cheap as it only requires the contract to evaluate a hash function on a short input (the key). However, using the ZKCP unfortunately puts significant computational burden on the players $\mathcal{R}$ and $\mathcal{S}$. Indeed, despite impressive progress on developing efficient zero knowledge proof system over the last few years, current state-of-the-art schemes [9, 19, 23, 41] are still rather inefficient if either ϕ gets complex, or the witness x becomes large. Hence, an important question is whether we can design smart contract based protocols for fair exchange, which combine the benefits of both approaches. That is, the main goal of this work is:

(1) Design simple smart contracts that can be executed with low fees.
(2) Put low computational burdens on the players by avoiding the use of heavy zero-knowledge proof systems.

In this work we develop a novel solution that achieves both goals simultaneously and enables efficient general fair exchange of digital commodities at low costs.

1.1 Our contribution

Efficient fair exchange. Our main contribution is a novel protocol for carrying out fair exchange by relying on smart contracts while avoiding costly zero-knowledge proofs. Our protocol works for arbitrary predicate functions ϕ and witnesses x of large size. Concretely, we model ϕ as a circuit with m gates taking as input a witness $x = (x_1, \ldots, x_n)$, where each x_i is represented as a bit string of length λ. We require that the gates of the circuit represent operations from some set of allowed operations Γ. The main distinctive feature of our construction is its efficiency. Concretely, for a circuit of size m our smart contract has asymptotic complexity of $O(\log(m))$, where the hidden constants in the asymptotic notation are small. In addition to small costs, the protocols that $\mathcal{R}$ and $\mathcal{S}$ execute are very efficient and avoid the use of expensive zero knowledge proofs. More precisely, in addition to evaluating ϕ the parties only have to compute $O(m)$ hash values.

Proof of misbehavior. Our construction is based on the following observation. While for large circuits ϕ and witnesses x proving that $\mathcal{S}$ behaves correctly is very costly, it is much cheaper to instead prove that $\mathcal{S}$ behaved incorrectly. As we show such a proof of misbehavior can be short and its verification involves only a small number of cryptographic operations. The proof can efficiently be verified by the underlying smart contract, which upon receiving such a proof penalizes the sender for cheating. At a technical level, we rely on ideas originally proposed in the context of multiserver delegation of computation [18], but several technical challenges need to be addressed to apply this idea in our setting. In particular, our construction is non-interactive and involves only two parties (the sender and the receiver). Moreover, we have to provide privacy that guarantees that the witness stays hidden until the receiver has committed coins into the contract for paying the sender. We give a detailed description of our construction in Section 4.

Definitions and security analysis. A second contribution of our work is to provide a formalization of contract-based or coin aided fair exchange protocols. To this end, we follow the universal composability framework of Canetti [16] and develop a new ideal functionality that formally captures the security properties one would expect from a fair exchange protocol. Our model deviates in two ways from the standard UC modeling. First, we take into account that protocol messages, which relate to blockchain transactions, may take up to time Δ to be processed due to the mining process. To integrate these delays in our model, we let the adversary decide the exact duration of a round, which can last at most time Δ. The second difference that is special to our modeling is that we need to deal with coins. We deviate from earlier works by Bentov and Kumaresan [10], who also model coins in a UC-like model, by introducing a global ideal functionality called ledger $\mathcal{L}$. The ledger functionality ensures that coins cannot be double spent and users cannot create new money. In addition to providing a formal model, we also carry out a full security analysis of our construction in the global random oracle model [17].

Applications. An appealing use case for our fair exchange protocol is selling files over the Internet. Such file sharing protocols allow exchanging, e.g., software (like Linux, LaTeX, Microsoft Windows 10 updates [36]), archived Internet data [4], public governmental databases [45], scientific data [34, 35], and movies [39], and are widely used in the Internet. In Section 5 we describe a protocol, which allows a receiver to buy a large file x that matches with a particular hash value h. Notice that in this application x may be many gigabytes large, but using our construction and the proofs of misbehavior, we can reduce the data that has to be processed by the smart contract to a few 100 bytes, while still guaranteeing fairness of the file exchange.

While our original motivation is to design efficient protocols for fair exchange, we emphasize that our work has also other interesting applications in the context of cryptocurrencies. In particular, we observe that our protocol offers an efficient and low-cost construction for realizing the "claim-or-refund" functionality of [10]. Claim-or-refund is used to design fair protocols for multiparty computation and works as follows. In an initial preparation phase a receiver can deposit some coins p and a function ϕ into the contract. This preparation phase is followed by two stages. First, in the claim phase, a party can claim the reward p by publishing an x such that $\phi(x) = 1$. Finally, in the refund phase the receiver can refund its p coins if nobody has claimed the reward yet. It is easy to see that the above describes the fair exchange setting, where the reward corresponds to the price paid for receiving x. Bentov and Kumaresan argue that claim-or-refund can be realized with smart contracts, however, a naive implementation will result into large costs from fees when ϕ is complex or x is large. Using our protocol claim-or-refund can be realized at significantly lower costs.

Extensions. A first extension that we consider is to integrate penalties into our protocol to mitigate the risk of denial of service attacks by the sender. This is realized by also letting the sender $\mathcal{S}$ lock q coins into the contract, which will go to the receiver $\mathcal{R}$ if $\mathcal{S}$ is caught cheating. Such financial penalties allow us to deal with the costs and fees for $\mathcal{R}$, which naturally occur in smart contract based protocols. Concretely, we want that when $\mathcal{S}$ is caught cheating, e.g., by sending a wrong file, $\mathcal{R}$ gets compensated for its costs of interacting with the contract (e.g., for the initial contract deployment) but also for the time where he blocked the p coins (collateral costs). This addition enforces honest behavior of rational senders.

To further reduce the costs of our construction, we discuss how we can run it inside off-chain state channels (see, e.g., [2, 37]), when sender and receiver wish to execute multiple recurring fair exchanges. To illustrate this setting consider a sender that wishes to sell t commodities to a receiver. In our original protocol from above this use case results into $O(t)$ interactions with the contract running on the blockchain. In contrast using state channels the parties can use the contract multiple times without requiring interaction with the blockchain, thereby significantly reducing costs of our construction. Once the state channel is open, the users can execute multiple fair exchanges between each other, and in the optimistic case only need to interact with the blockchain during opening and closing. This allows us to amortize the on-chain costs over multiple fair exchange executions.

Implementation. Besides our conceptual contributions, we also provide a proof-of-concept implementation of our contract (c.f. github.com/lEthDev/FairSwap). In our implementation we focus on the file sale application mentioned above and discuss the advantages of contracts specialized for this application in comparison to the general construction. Additionally, we benchmark the costs for deploying this contract and running our protocol over Ethereum. For more information on the details of the implementation we refer the reader to Section 5.

1.2 Related work

As mentioned above, the ZKCP protocols were introduced in [46]. Their first implementation (for selling solutions of Sudoku puzzles) was presented in [12], and was subsequently broken by Campanelli et al. [15]. The weakness discovered in [15] concerns all the ZKCP protocols that use NIZKs protocols [11] where the verifier generates the common reference string. The authors of [15] present a fix to this problem using a tool called *Subversion-NIZK* [8]. They also extend the concept of ZKCP to protocols for paying for *service* (i.e.: not only for static data). ZKCP protocols for cryptocurrencies that do not support contracts or scripts in the transactions were constructed in [7].

Fair exchange is a well studied research problem. It has been shown that without further assumptions fair exchange cannot be achieved without a Trusted Third Party (TTP) [25, 40, 47]. To circumvent this impossibility, research has studied weaker security models – most notably, the optimistic model in, which a TTP is consulted only in case one party deviates from the expected behavior [5, 13]. One may view smart-contract based solutions as a variant of optimistic protocols, where the smart contract takes the role of the TTP. In particular [33] considers a similar use case (file sharing), but the security guarantees it achieves are very different from our work: (a) the arbiter uses a cut-and-choose approach and hence for a corrupted file the probability of not detecting a cheater is non-negligible (and in fact quite high for some cases, see citation [13] in [33]); (b) due to the cut-and-choose the workload of the arbiter is large, resulting into high fees in a smart contract setting. In contrast our solution only has a negligible error rate, and the financial costs are small. We also stress that the cost model of an arbiter and a smart contract is very different.

Very recently, there has been a large body of work on using cryptocurrencies such as Bitcoin to achieve fairness in cryptographic protocols (see, e.g., [3, 10, 30–32] and many more). As already discussed we emphasize that our protocol can be used to further reduce on-chain complexity of these protocols by providing an efficient realization of the claim-or-refund functionality that is heavily used in the protocols constructed in these works.

Finally, we point out that the concept of "proofs of misbehavior" used in our construction are a frequently applied technique in practical smart contract based protocols. One notable example is the scalability solution called TrueBit, developed by Teutsch and Reitwießner [44]. The idea is to outsource the potentially resource intensive process of finding solutions for complicated computational puzzles. The system consists of *provers, verifiers* and *judges* where the provers are paid for solving computationally hard tasks. Since the provers have the ability to lie about their results and still claim the money, they are punished whenever a misbehavior is detected. This detection is done by the verifiers, which are rewarded whenever they find bugs in the solution of the provers. Again, the verifiers have the ability to lie about their results to get money, which is why there exist the judges. A judge is a computationally bounded, trusted entity backed by the blockchain security and can be implemented as a smart contract. This setting is similar to ours since we also rely on the cryptocurrency and its smart contracts to judge a dispute between two parties by verifying only a single operation instead of running complex computation off-chain. The main difference is that their protocol requires all provers to publish their solutions and interact with the verifier and judge in case they are challenged. In our case we need the verification of misbehavior to be non-interactive and the result of the computation should stay secret to outside observers. These restrictions add additional overhead to our protocol in comparison to a simple protocol which only helps to resolve disputes.

Finally, with respect to TrueBit we point out that its current whitepaper [44] provides only very little details on how such proofs are created at a technical level and no formal security analysis is provided. Hence, one may view our construction of the subroutines *Encode*, *Extract* and *Judge* described in Section 4 as a building block for TrueBit, and our formal security analysis as a first step in formally analyzing the TrueBit system.

2 PRELIMINARIES

Notation. In this section we present the notation, cryptographic building blocks, and the formal abstraction of circuits, which we will use in our schemes. For further details the reader may also consult Appendix B. By writing $[n]$ we refer to the set of natural numbers

$\{1, \ldots, n\}$. We say that two distribution ensembles $\mathsf{X} = \{X(n)\}$ and $\mathsf{Y} = \{Y(n)\}$ with $n \in \{0,1\}^*$ are *computationally indistinguishable* (denoted as $\mathsf{X} \approx_c \mathsf{Y}$) if every probabilistic poly-time (ppt) algorithm $\mathcal{Z}$ cannot distinguish X and Y except with negligible probability. In the following all parties are modeled as probabilistic poly-time (ppt) Turing machines and we will often omit to explicitly mention it.

Cryptographic building blocks. The main cryptographic primitive used by our scheme are cryptographic hash functions. A *hash function* $H : \{0,1\}^* \to \{0,1\}^\mu$ maps stings of arbitrary length to binary strings of a fixed length μ. We require that the hash function H satisfies standard security guarantees such as collision resistance for sufficiently large μ. While in practice collision resistant hash functions can be instantiated with constructions such as SHA3, for our security analysis, we will assume that H is modeled as a global random oracle $\mathcal{H}$ [14]. We refer the reader to Appendix A for further details on the global random oracle model.

A *commitment scheme* for input values $x \in \{0,1\}^*$ is a pair of algorithms (Commit, Open), where the (probabilistic) algorithm $\mathsf{Commit}(x)$ outputs a commitment c and an opening value d, and the algorithm $\mathsf{Open}(c, d, x) = \{0, 1\}$ outputs 1 for a valid commitment $(c, d) \leftarrow \mathsf{Commit}(x)$. Cryptographically secure commitment schemes have to satisfy the *hiding* and *binding* properties. Hiding guarantees that for any two messages x, x' and $(c, d) = \mathsf{Commit}(x)$ and $(c', d') = \mathsf{Commit}(x')$, we have that $c \approx_c c'$. The binding property requires that it is computationally hard to find a triple (c, d, d') such that $\mathsf{Open}(c, d, x) = 1$ and $\mathsf{Open}(c, d', x') = 1$ with $x \neq x'$.

The symmetric encryption scheme used in this paper needs to satisfy *indistinguishability under chosen plaintext attacks* (IND-CPA security). This means that for any ppt adversary that chooses two messages x_0, x_1 and learns $c = \mathsf{Enc}(k, x_b)$ for a randomly chosen b, it must be impossible to guess b correctly except with negligible advantage.

We will also use a standard notion of Merkle trees. A *Merkle tree* of elements $x_1, \ldots, x_n \in \{0,1\}^*$ (where for simplicity n is an integer power of 2) is a labeled binary tree $M = \mathit{Mtree}(x_1, \ldots, x_n)$ with the ith leaf labeled by x_i. Moreover, a label v_j of every non-leaf node V_j is the hash of the labels v_j^l and v_j^r of its two child nodes V_j^l and V_j^r respectively (i.e. $v_j := H(v_j^l, v_j^r)$). We call V_j the *parent* of v_j^l and v_j^r. We say v_j^l is a *sibling* of v_j^r and vice versa. A Merkle tree of n elements $x_1, \ldots, x_n$ is created with the *Mtree* algorithm (c.f. Algorithm 1).

Algorithm 1 Merkle tree hash *Mtree*

Input: $(x_1, \ldots, x_n)$
  set V ▷ V will be the root node
  if n = 1 **then** ▷ If input is single value, V is a leaf
    $\mathit{label}(V) = x_1$ ▷ assign label of leaf V as x_1
  else ▷ otherwise recursively call *Mtree* algorithm again
    $v_0^l = \mathit{Mtree}(x_1, \ldots, x_{\lceil n/2 \rceil})$
    $v_0^r = \mathit{Mtree}(x_{\lceil n/2 \rceil + 1}, \ldots, x_n)$
    $\mathit{label}(V) = H(\mathit{root}(v_0^l) || \mathit{root}(v_0^r))$ ▷ label = hash of subtree
Output: Merkle tree M with root V

The label at the root of a Merkle tree M is denoted by $\mathit{root}(M)$. For efficiently proving that an element x_i is included in the Merkle tree (given by its root hash h), we use a *Merkle proof* ρ, which is a vector (of length $\log_2(n)$) consisting of labels on all the siblings of elements on a path from the ith leaf to the root of the Merkle tree. We denote the algorithm for generating a Merkle proof by *Mproof*, which on input a Merkle tree M and an index i outputs a Merkle proof ρ that x_i is the ith leaf of M (c.f. Algorithm 2).

Algorithm 2 Merkle tree proof *Mproof*

Input: Merkle tree M, index i
  $V = M[i]$ ▷ let V be the i-th leaf node of M
  for each $j \in [\log_2(n)]$ **do**
    set $l_j = \mathit{label}(\mathit{sibling}\ \text{of}\ v)$
    set $v = \mathit{parent}\ \text{of}\ v$
Output: Merkle Proof $\rho = (l_1, \ldots, l_d)$

Finally, the algorithm *Mvrfy* takes as input an element x_i, a Merkle proof ρ and a root of a Merkle tree $\mathit{root}(M)$. The algorithm *Mvrfy* verifies if the i-th leaf element x corresponds to a Merkle tree with root r using proof ρ (generated with Algorithm 3). If the verification holds, the algorithm outputs 1 if the verification fails, the algorithm outputs 0. If the root r of the tree is known

Algorithm 3 Merkle tree proof verification *Mvrfy*

Input: $i \in [n]$, $x \in \{0,1\}^\lambda$, $\rho = (l_1, \ldots, l_d)$, $h \in \{0,1\}^\mu$
  for each $l_j \in \rho$ **do**
    if $i/2^j = 0 \mod 2$ **then**
      $x = H(l_k || x)$
      if $H(\mathsf{isPrgrmd}(l_k || x)$ **then**
        Terminate and Output $\perp$ ▷ reject hash is programmed
    else
      $x = H(x || l_j)$
      if $H(\mathsf{isPrgrmd}(x || l_j))$ **then**
        Terminate and Output $\perp$ ▷ reject hash is programmed
  if $x = h$ **then**
    Output 1
  else
    Output 0

beforehand, this algorithm can be used to verify that x is the i-th element of a Merkle Tree with root r.

Circuits. In this work we will use circuits to model arbitrary program code over an admissible instruction set Γ. A circuit ϕ is represented by a directed acyclic graph, where the edges carry values from some set X and the nodes represent gates. We assume that gates evaluate some instruction $op : X^\ell \to X$, where $op \in \Gamma$. A gate is evaluated by taking as input up to ℓ values from X, carrying out the instruction op and sending the result on its outgoing wire. We limit fan-in of gates to ℓ and model arbitrary fan-out by letting the output of a gate be an input to any number of other gates. A special type of gate that we consider are input gates, which have no incoming edges (i.e., in-degree 0) and model the initial input of the circuit. We will often use the notation $\phi(x)$ to represent the output of evaluating a circuit ϕ on some input x, where the evaluation is done layer-by-layer starting with the input gates.

Our construction requires a concise way to fully describe the topology and the operations of a circuit ϕ. To this end, we assign to

each gate g of ϕ a label represented by a tuple $\phi_i := (i, op, I_i)$. Each such tuple consists of an instruction $op : X^\ell \to X$, which denotes the instruction carried out by this gate and a unique identifier $i \in \mathbb{N}$. The identifiers are chosen in the following way: All gates in the jth layer of the circuit have identifiers that are larger than the identifiers used by gates in layer $j-1$. This means that the identifier of g is larger than the identifiers of all input gates to g. Finally, the last element I_i is a set of identifiers, where $I_i = \emptyset$ if g is an input gate; otherwise I_i is defined to be the set of identifiers of the input gates to g. In the following, we will often abuse notation and sometimes use ϕ to present the circuit (e.g., when writing $\phi(x)$ for the evaluation of ϕ on input x), or to represent the tuple of labels, i.e., $\phi = (\phi_1, \ldots, \phi_m)$.

It is well known that any deterministic program can be represented by a Boolean circuit. In this case, we have $X = \{0, 1\}$ and $\Gamma = \{\mathsf{AND}, \mathsf{NOT}\}$ are the standard binary operations, where each gate has an in-degree of at most $\ell = 2$. For the purpose of this paper Γ will typically contain more powerful operations that compute on larger bit strings $\{0, 1\}^\lambda$. Examples of such higher-level instructions are hash function evaluations or modulo multiplication. This models more closely the capabilities that are offered by higher-level programming languages such as Solidity offered by Ethereum.

Smart Contracts. Besides normal payments between users many cryptocurrencies support the execution of smart contracts. Smart contracts bind money transfer to program code, and thereby allow to execute transactions based on complex contractual agreements enforced by the miners of the cryptocurrency. The most prominent system that supports expressive smart contracts is the cryptocurrency Ethereum. In Ethereum smart contracts can be written in a script language (e.g., Solidity), which is then complied down to low-level Ethereum Virtual Machine (EVM) bytecode. Once a contract is deployed its execution can be triggered via transactions, which are processed by the miners. Miners are incentivized to process transactions and execute smart contracts through transaction fees. In Ethereum these fees are paid in *gas* – an internal Ethereum currency – and the value of these fees depends on the complexity of the program code, which is executed by processing the transaction. Each EVM instruction has a fixed amount of gas assigned to it, but the exchange rate between Ether – Ethereum's currency – and gas may change depending on market demand. For instance, the evaluation of a standard hash function is fixed to cost 27265 gas, which with the current exchange rate translates to 0.06 USD. Of course, different currencies like Ethereum Classic may have much cheaper costs for executing smart contracts depending on their current exchange rate.

3 OUR SECURITY MODEL

In this section we give an introduction to the adversarial model and an overview of our system. We start with a high-level introduction to the *Universally Composable* (UC) framework [16] and show how to model a global ledger functionality $\mathcal{L}$. Next, we present the ideal functionality $\mathcal{F}_{\mathsf{cfe}}^{\mathcal{L}}$ for coin-aided fair exchange and discuss its security guarantees.

3.1 UC model for blockchain-based protocols

One of the most widely used methods to describe and analyze complex cryptographic protocols is the universal composability (UC) framework of Canetti [16]. In the UC framework the security of a cryptographic protocol Π is analyzed by comparing its real world execution with an idealized protocol running in an ideal world. In the *real world* Π is executed amongst a set of parties modeled as interactive poly-time Turing machines. A protocol is attacked by an adversary $\mathcal{A}$, who can corrupt some of these parties (for simplicity we consider static corruption), which means that these parties – including their internal state and all their actions – are fully controlled by $\mathcal{A}$. To analyze the security of Π in the real world, we "compare" its execution with an ideal world. In the *ideal world*, parties interact with an ideal functionality, which specifies the protocol's interface and can be viewed as an abstract specification of what security properties Π shall achieve. In the ideal world, the ideal functionality can be "attacked" through its interface by an ideal world adversary – often called the simulator *Sim*. In both the real and ideal world there is an additional special party called the environment $\mathcal{Z}$, which orchestrates both worlds by providing the inputs for all parties, and receiving their outputs. Informally speaking, a protocol Π is said to be *UC-secure* if the environment $\mathcal{Z}$ cannot distinguish whether it is interacting with the ideal or real world. This implies that Π is at least as secure as the ideal functionality.

Hybrid world. A common method that is used in UC to modularize the design of a protocol is to rely on hybrid ideal functionalities. To this end, we define a hybrid world where the protocol has access to some set of idealized functionalities $\mathcal{G}_1, \ldots, \mathcal{G}_m$. In our case, we will construct our fair exchange protocol in a hybrid world where an idealized functionality realizing a judge smart contract is available. We will explain this contract in more detail in Section 4.

The ledger functionality. In addition to the traditional UC model described above, our setting requires us to handle coins that can be transferred between parties. To this end we use the model of [21], which introduces a simple ledger functionality $\mathcal{L}$ to model the basic properties of a cryptocurrency. Concretely, we allow parties to transfer coins between each other and support contracts that lock coins. Since the ledger functionality is available both in the real and ideal world, and moreover can be used over multiple protocol executions, we model $\mathcal{L}$ as a global ideal functionality [14, 17].

Let us briefly describe the functionality $\mathcal{L}$ (cf. Figure 1), whose internal state is public and consists of the balances $p_i \in \mathbb{N}$ of parties $\mathcal{P}_i$ and a list of contracts. For the latter, we define a partial function $L : \{0, 1\}^* \to \mathbb{N}$ that maps a contract identifier id to an amount of coins that is locked for the execution of contract id. The ledger functionality offers the following interface to the parties. The environment $\mathcal{Z}$ can update the account balance of the users via sending an *update* message to $\mathcal{L}$. The parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ cannot directly interact with $\mathcal{L}$, but their balance can be updated via *freeze*/*unfreeze* messages sent by other ideal functionalities, in which case we will write $\mathcal{G}^{\mathcal{L}}$. More precisely, *freeze* transfers money from the balance of a party to a contract, while *unfreeze* sends this money back to the user's account. To simplify exposition,

Functionality $\mathcal{L}$, running with a set of parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ stores the balance $p_i \in \mathbb{N}$ for every party $\mathcal{P}_i$, $i \in [n]$ and a partial function L for frozen cash. It accepts queries of the following types:

Update Funds Upon receiving message $(update, \mathcal{P}_i, p)$ with $p \geq 0$ from $\mathcal{Z}$ set $p_i = p$ and send $(updated, \mathcal{P}_i, p)$ to every entity.

Freeze Funds Upon receiving message $(freeze, id, \mathcal{P}_i, p)$ from an ideal functionality of session id check if $p_i > p$. If this is not the case, reply with $(nofunds, \mathcal{P}_i, p)$. Otherwise set $p_i = p_i - p$, store (id, p) in L and send $(frozen, id, \mathcal{P}_i, p)$ to every entity.

Unfreeze Funds Upon receiving message $(unfreeze, id, \mathcal{P}_j, p)$ from an ideal functionality of session id, check if $(id, p') \in L$ with $p' \geq p$. If this check holds update (id, p') to $(id, p' - p)$, set $p_j = p_j + p$ and send $(unfrozen, id, \mathcal{P}_j, p)$ to every entity.

Figure 1: Global ledger functionality $\mathcal{L}$

for a malicious party P_i we let the simulator *Sim* decide how many coins are sent back to P_i's account by an *unfreeze* message.[2]

Global random oracles. In addition to the global ledger functionality $\mathcal{L}$, we will also use a global random oracle $\mathcal{H}$. Concretely, we will follow the recent formalism of [14] and model hash functions as programmable and observable random oracles. Upon querying the oracle on some value q it returns a random response $r \in \{0,1\}^{\mu}$. If the same value is queried twice then for both queries $\mathcal{H}$ returns the same response. In addition, we require that $\mathcal{H}$ has an interface for observing the all input/output tuples, for which queries have been made to $\mathcal{H}$, and an interface for programming the $\mathcal{H}$. For further details on the global random oracle model and the formalism of [14] we refer the reader to Appendix A.

Communication model. We assume a synchronous communication model, where the protocol is executed in rounds and all parties are always aware of the current round. Formally, this can be modeled by a global clock functionality [6, 28, 29], but we omit the details here. We make the following assumptions for the time it takes for parties to communicate with each other. If a party (including the adversary) sends a message to another party in round i, then it is received by that party at the beginning of round $i + 1$. For communication with the ideal functionalities, we will explicitly specify when they are expecting inputs from the parties. The communication itself however is instantaneous. Notice that in general when there is interaction with the ledger $\mathcal{L}$ or with the smart contract, then the functionality will be prepared for this by expecting a message within a certain round. We emphasize that this is only an abstraction and round times can be very large, since in reality they correspond to communication with the blockchain which takes significantly more time than interaction between parties.

[2]Looking ahead this is needed to simulate the case when a malicious party in the real world decides to request a refund and thus lock its coins in the contract. To simplify the functionalities, coins are unlocked automatically in the ideal world.

The ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ (in session id) interacts with a receiver $\mathcal{R}$, a sender $\mathcal{S}$, the ideal adversary *Sim* and the global ledger $\mathcal{L}$.

Initialize

(Round 1) Upon receiving $(sell, id, \phi, p, \boldsymbol{x})$ with $p \in \mathbb{N}$ from $\mathcal{S}$, leak $(sell, id, \phi, p, \mathcal{S})$ to *Sim*, store witness $\boldsymbol{x}$, circuit ϕ and price p.

(Round 2) Upon receiving (buy, id, ϕ, p) from receiver $\mathcal{R}$ in the next round, leak $(buy, id, \mathcal{R})$ to *Sim* and send $(freeze, id, \mathcal{R}, p)$ to $\mathcal{L}$. If $\mathcal{L}$ responds with $(frozen, id, \mathcal{R}, p)$ go to Reveal phase.

Reveal

(Round 3) Upon receiving $(abort, id)$ from the corrupted sender $\mathcal{S}^*$ in round 3, send $(unfreeze, id, p, \mathcal{R})$ to $\mathcal{L}$ in the next round and terminate. Otherwise if you do not receive such message in round 3, then send $(bought, id, \boldsymbol{x})$ to $\mathcal{R}$ and go to Payout phase.

Payout

(Round 4) Upon receiving $(abort, id)$ from the corrupted receiver $\mathcal{R}^*$, wait until round 5 to send $(sold, id)$ to $\mathcal{S}$, $(unfreeze, id, p, \mathcal{S})$ to $\mathcal{L}$ and terminate. Otherwise, if no such message was received:

- If $\phi(\boldsymbol{x}) = 1$, send messages $(unfreeze, id, p, \mathcal{S})$ to $\mathcal{L}$ and $(sold, id)$ to $\mathcal{S}$,
- If $\phi(\boldsymbol{x}) \neq 1$, send messages $(unfreeze, id, p, \mathcal{R})$ to $\mathcal{L}$ and $(not\ sold, id)$ to $\mathcal{S}$.

Figure 2: Ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ for coin aided fair exchange

Simplifications in comparison to full UC. To simplify our presentation, we omit session identifiers and the sub-session identifiers (typically denoted with *sid* and *ssid*) and use instead the contract identifier *id* to uniquely distinguish sessions. In practice, the contract identifier may correspond to the contract address.

3.2 Ideal functionality for fair exchange

Our ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ (cf. Figure 2) describes a setting where $\mathcal{S}$ sells a witness $\boldsymbol{x}$ to a receiver $\mathcal{R}$ and obtains p coins if this witness was correct. Correctness of the witness is defined through a predicate function ϕ, which for a valid input $\boldsymbol{x}$ outputs 1, and 0 otherwise. Internally, $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ will interact with the global ledger functionality $\mathcal{L}$ to maintain the balance of the parties during the fair exchange (for instance, when a witness was successfully sold then p coins are unfrozen in $\mathcal{S}$'s favor).

The functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ has three phases, which we first describe for the case when the parties are honest. During the *initialization phase* the ideal functionality receives inputs from both $\mathcal{S}$ and $\mathcal{R}$. $\mathcal{S}$ sends the input $\boldsymbol{x}$ and a description of the predicate circuit ϕ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. If $\mathcal{R}$ confirms this request, the functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ instructs $\mathcal{L}$

to freeze p coins from $\mathcal{R}$. If this is not possible due to insufficient funds, the functionality ends the fair exchange protocol. During the *reveal phase*, the receiver will learn $\boldsymbol{x}$ after which the *payout phase* is started. In the payout phase we consider two cases. If $\phi(\boldsymbol{x}) = 1$, then the sender $\mathcal{S}$ receives the coins as a payment; otherwise if $\phi(\boldsymbol{x}) \neq 1$, the functionality instructs $\mathcal{L}$ to send the coins back to $\mathcal{R}$.

In addition to the above, malicious parties can abort the execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in both the *reveal phase* and the *payout phase*. Concretely, during the *reveal phase* a malicious sender $\mathcal{S}^*$ may abort, which results into sending the funds back to $\mathcal{R}$. On the other hand a malicious receiver $\mathcal{R}^*$ may abort the exchange during the *payout phase*, which results into $\mathcal{S}$ receiving the coins. Both these aborts model that in the protocol a malicious party may abort its execution by not sending a required message.[3]

Security properties. Let us now discuss what security properties are guaranteed by our ideal functionality. Since our protocol realizes the ideal functionality these security properties are also achieved by our protocol in the real world.

Termination. If at least one party is honest, the fair exchange protocol terminates within at most 5 rounds and unlock all coins from the contract.

Sender Fairness. An honest sender $\mathcal{S}$ is guaranteed that the receiver $\mathcal{R}$ only learns the witness iff he pays p coins.

Receiver Fairness. An honest receiver $\mathcal{R}$ is ensured that he only pays p coins iff the sender delivers the correct witness in exchange.

Consider first the case of a malicious receiver $\mathcal{R}^*$. The ideal functionality Π only proceeds to the *reveal* phase if the receiver has locked p coins into the contract during initialization. Then, in the *payout* phase these coins are only given to $\mathcal{R}^*$ iff $\phi(\boldsymbol{x}) \neq 1$. In all other cases (i.e., if $\phi(\boldsymbol{x}) = 1$ or a malicious $\mathcal{R}^*$ aborts), $\mathcal{S}$ receives p coins as required by sender fairness. Now assume instead a malicious sender $\mathcal{S}^*$, who only receives a payment during the payout phase if either $\phi(\boldsymbol{x}) = 1$ (i.e., the witness was valid), or the receiver aborts in Step (4^*), which an honest receiver never would do. This implies receiver fairness. Finally, it is easy to see that the ideal functionality will terminate after at most 5 rounds, which may happen during payout when a malicious receiver $\mathcal{R}$ aborts.[4].

UC Definition of security. Consider a protocol Π with access to the judge contract functionality $\mathcal{G}_{\text{jc}}$, the global random oracle $\mathcal{H}$ and the global ledger functionality $\mathcal{L}$. The output of an environment $\mathcal{Z}$ interacting with a protocol Π and an adversary $\mathcal{A}$ on input 1^κ and auxiliary input $x \in \{0,1\}^*$ is denoted as

$$REAL_{\Pi,\mathcal{A},\mathcal{Z}}^{\mathcal{G}_{\text{jc}},\mathcal{L},\mathcal{H}}(\kappa, x).$$

In the ideal world the parties do not interact with each other but only forward their inputs to an ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. In this setting we will call the adversary a *simulator Sim* and denote the

[3]Looking ahead, in the protocol a malicious sender $\mathcal{S}^*$ may not reveal the key to the contract resulting into $\mathcal{R}$ receiving back his locked funds. On the other hand a malicious receiver $\mathcal{R}^*$ may not complain during the payout phase, even though he received a witness $\boldsymbol{x}$ with $\phi(\boldsymbol{x}) \neq 1$. In this case the funds must go to $\mathcal{S}$ because from the contract's point of view the case when a malicious receiver did not complain even though $\phi(X) \neq 1$ is indistinguishable from the case when $\phi(X) = 1$.

[4]Note, the ideal functionality does not provide any fairness or termination guarantees for two corrupted parties.

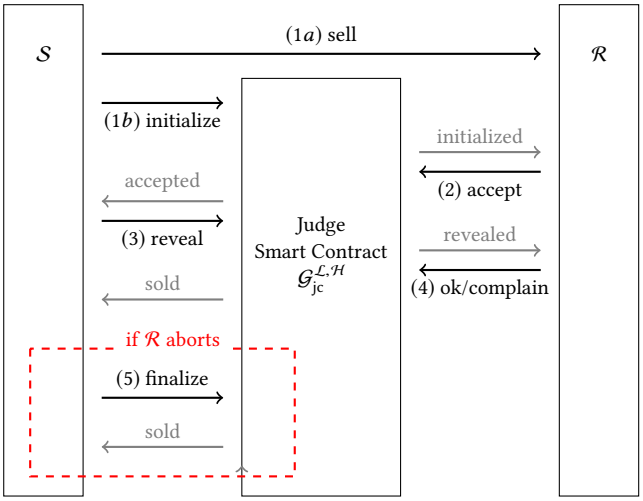


Figure 3: Outline of fair exchange with judge contract

above output as

$$IDEAL_{Sim,\mathcal{Z}}^{\mathcal{F}_{\text{cfe}},\mathcal{L},\mathcal{H}}(\kappa, x).$$

Given these two random variables, we can now define the security of our protocol Π as follows.

Definition 1 (GUC security of Π). *Let $\kappa \in \mathbb{N}$ be a security parameter, Π be a protocol in the $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$-hybrid world. Π is said to* GUC realize *$\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$-hybrid world if for every ppt adversary $\mathcal{A}$ attacking Π there exists a ppt algorithm Sim, such that the following holds for all ppt environments $\mathcal{Z}$ and for all $x \in \{0,1\}^*$:*

$$IDEAL_{Sim,\mathcal{Z}}^{\mathcal{F}_{\text{cfe}},\mathcal{L},\mathcal{H}}(\kappa, x) \approx_c REAL_{\Pi,\mathcal{A},\mathcal{Z}}^{\mathcal{G}_{\text{jc}},\mathcal{L},\mathcal{H}}(\kappa, x)$$

4 OUR MAIN CONSTRUCTION

As highlighted in the introduction, we will solve the disagreement where the sender $\mathcal{S}$ claims that he sent a witness $\boldsymbol{x}$ such that $\phi(\boldsymbol{x}) = 1$ to the receiver $\mathcal{R}$, while $\mathcal{R}$ claims the contrary. To resolve this conflict, we will use a smart contract to act as a *judge* and can decide, which of both cases occurred. In order to minimize costs for the execution of this contract we do not want the judge contract to learn ϕ, $\boldsymbol{x}$ nor require it to run $\phi(\boldsymbol{x})$. Instead, we outsource the heavy work of evaluating the circuit to $\mathcal{S}$ and $\mathcal{R}$, respectively. The judge contract will only need to verify a *concise proof of misbehavior*, which $\mathcal{R}$ generates if he wants to complain about the fact that $\phi(\boldsymbol{x}) \neq 1$. We will show in this section how to generate such a proof, whose size is logarithmic in the circuit size representing ϕ. This is an important property, since we allow the witness $\boldsymbol{x}$ and therefore also ϕ to be large, i.e., $\boldsymbol{x}$ may consist of n elements, $\boldsymbol{x} = (x_1 \ldots, x_n)$, with $x_i \in \{0,1\}^\lambda$. The circuit ϕ takes as input $\boldsymbol{x}$ and has $m > n$ gates, which are evaluated according to the topology of the circuit where gate g_m is the output gate of the overall circuit. If the operation of gate g_m outputs 1, the circuit ϕ accepts the witness $\boldsymbol{x}$, otherwise the witness is rejected.

We propose a new scheme that at a high-level works as follows (cf. also Figure 3). In the first round the sender $\mathcal{S}$ encrypts $\boldsymbol{x}$ and auxiliary information about the computation of $\phi(\boldsymbol{x})$ and sends

these ciphertexts to the receiver $\mathcal{R}$ (Step 1a). In the same round it sends a commitment of the key k used for the encryption to the judge contact (Step 1b). The receiver does some preliminary consistency checks in the next round and (if he accepts) sends p coins to the judge contract (Step 2). In the third round, the sender is supposed to reveal the secret key k to the judge contract (Step 3). This enables $\mathcal{R}$ to decrypt $\boldsymbol{x}$ and verify the computation of $\phi(\boldsymbol{x})$. If $\boldsymbol{x}$ was not correct, i.e., $\phi(\boldsymbol{x}) \neq 1$, then $\mathcal{R}$ has the chance to complain about the invalid $\boldsymbol{x}$ in the fourth round via a concise proof of misbehavior (Step 4). In this case the p coins locked by $\mathcal{R}$ in the contract get refunded. Finally, in case $\mathcal{R}$ was malicious and did no react in round 4, $\mathcal{S}$ can finalize the smart contract in round 5 (Step 5).

4.1 Concise proofs of misbehavior

Before we describe our main protocol in detail, we start by taking a closer look on how $\mathcal{R}$ can generate the complaint for the judge smart contract. The key idea is that checking if some part of the claimed computation was carried out incorrectly is much easier than verifying the correctness of the entire computation. In our construction we let the judge validate only the operation and the result of a single incorrectly computed gate of ϕ. This is done via a *concise proof of misbehavior*. Such a proof includes the inputs and output of the gate to allow verification of this particular gate. In addition, in order to prevent $\mathcal{R}$ from sending wrong inputs and outputs, the judge contract has to be ensured that the values used for the proof of misbehavior were originally sent by $\mathcal{S}$. This is guaranteed in an efficient way using Merkle proofs.

We present our protocol in a modular way using the subroutines *Encode*, *Extract* and *Judge* shown in Algorithms 1-3. In our protocol $\mathcal{S}$ uses the algorithm *Extract* to encrypt $\boldsymbol{x}$ and the intermediate values that are produced during the evaluation of $\phi(\boldsymbol{x})$ (c.f., Algorithm 1). The output $\boldsymbol{z}$ of this encoding procedure is sent to the receiver $\mathcal{R}$. Moreover, as described above $\mathcal{S}$ sends a commitment of the key k to the smart contract.

Algorithm 4 *Encode*$(\phi, \boldsymbol{x}, k)$

for each $i \in [n]$ **do**
  $out_i = x_i$ ▷ Assign witness to input wires
  $z_i = \mathsf{Enc}(k, out_i)$ ▷ Encrypt input values
for each $i \in \{n+1, \ldots, m\}$ **do**
  parse $\phi_i = (i, op_i, I_i)$
  $out_i = op_i(out_{I_i[1]}, \ldots, out_{I_i[\ell]})$ ▷ Compute the i-th operation
  $z_i = \mathsf{Enc}(k, out_i)$ ▷ Encode output values
Output: $\boldsymbol{z} = (z_1, \ldots, z_m)$

Once $\mathcal{S}$ reveals the encryption key k, $\mathcal{R}$ can run the *extraction subroutine Extract* (cf. Algorithm 2) and recover $\boldsymbol{x}$. The algorithm gets as input the encryption $\boldsymbol{z}$, the circuit ϕ, the key k and outputs a tuple, where the first element is the decoding of the witness $\boldsymbol{x}$ and the second is either $\perp$ (if $\phi(\boldsymbol{x}) = 1$) or a concise proof of misbehavior π (if $\phi(\boldsymbol{x}) \neq 1$). The proof π is used later to convince the judge/contract that some step of the computation of $\phi(\boldsymbol{x})$ is incorrect.

On input the decoding key k, the root elements r_z and r_ϕ, and the proof π the algorithm *Judge* outputs 1 if the complaint succeeds

Algorithm 5 *Extract*$(\phi, \boldsymbol{z}, k)$

parse $\phi = \phi_1, \ldots, \phi_m$
for each $i \in [n]$ **do**
  $out_i = \mathrm{Dec}(k, z_i)$ ▷ Decrypt first n outputs
  $x_i = out_i$ ▷ Extract witness
$M_z = Mtree(\boldsymbol{z})$ ▷ Compute Merkle tree over $\boldsymbol{z}$
$M_\phi = Mtree(\phi)$ ▷ Compute Merkle tree over ϕ
for each $i \in \{n+1, \ldots, m\}$ **do**
  parse $\phi_i = (i, op_i, I_i)$
  $out_i = op_i(out_{I_i[1]}, \ldots, out_{I_i[\ell]})$ ▷ Compute output of i-th gate
  if $\mathrm{Dec}(k, z_i) \neq out_i$ or $(i = m$ and $out_i \neq 1)$ **then**
    $\pi_\phi = Mproof(i, M_\phi)$ ▷ Proof that $\phi_i \in \phi$
    $\pi_{out} = Mproof(i, M_z)$ ▷ Proof that $z_i \in \boldsymbol{z}$
    for each $k \in [\ell]$ **do**
      set $j = I_i[k]$ ▷ j is the k-th index in set I_i
      $\pi^k_{in} = Mproof(j, M_z)$ ▷ Proof that $z_j \in \boldsymbol{z}$
    set $\pi = (\pi_\phi, \pi_{out}, \pi^1_{in}, \ldots, \pi^\ell_{in})$
    Output: $((x_1, \ldots, x_n), \pi)$
Output: $((x_1, \ldots, x_n), \perp)$

or 0 otherwise (cf. Algorithm 3). In order to verify the $i-$th step of $\phi(\boldsymbol{x})$, the judge needs to know the label $\phi_i = (op_i, i, I_i)$, all values $out_{I_i[1]}, \ldots, out_{I_i[\ell]}$ on its input wires and the value of its output wire out_i. Using this information the algorithm computes the output of the i-th gate and compares it with the value out_i. If both values are the same, then the computation was carried out correctly, and the algorithm outputs 0 (i.e., it rejects the complain). Otherwise, it outputs 1 and we say that the judge algorithm *accepts* the complain.

To guarantee that $\mathcal{R}$ can only complain about values that he has indeed received from $\mathcal{S}$ and that violate the predicate function ϕ on which both $\mathcal{S}$ and $\mathcal{R}$ have agreed on, we require that the Merkle roots $r_z = root(Mtree(\boldsymbol{z}))$ and $r_\phi = root(Mtree(\phi))$ are stored in the judge contract. Concretely, $\mathcal{S}$ sends r_z and r_ϕ to the contract in the first round, and $\mathcal{R}$ will only deposit p coins into the contract if these values are consistent with $\boldsymbol{z}$. When later *Judge* receives a concise proof of misbehavior *Judge* checks if the containing Merkle proofs are consistent with r_z and r_ϕ. Only if this is the case a complaint is accepted by the contract.

Algorithm 6 *Judge*(k, r_z, r_ϕ, π)

parse $\pi = (\pi_\phi, \pi_{out}, \pi^1_{in}, \ldots, \pi^\ell_{in})$
parse $\pi_\phi = (\phi_i, \rho_\phi)$
parse $\phi_i = (i, op_i, I_i)$ ▷ Reject if ϕ_i not i-th step of $\phi(\boldsymbol{x})$
if $Mvrfy(\phi_i, \rho_\phi, r_\phi) \neq 1$ **output:** 0
parse $\pi_{out} = (z_i, \rho_{out})$ ▷ Reject if z_i not i-th element of $\boldsymbol{z}$
if $Mvrfy(z_i, \rho_{out}, r_z) \neq 1$ **output:** 0
$out_i = \mathrm{Dec}(k, z_i)$
if $i = m$ **and** $out_i \neq 1$ **output:** 1 ▷ Accept if $\phi(\boldsymbol{x}) \neq 1$
for each $j \in [\ell]$ **do** ▷ j is the k-th index in set I
  parse $\pi^j_{in} = (z_j, \rho_j)$ ▷ Reject if z_j not $\boldsymbol{z}[j]$
  if $Mvrfy(z_j, \rho_j, r_z) \neq 1$ **output:** 0
  $out_{I_i[j]} = \mathrm{Dec}(k, z_j)$
if $op_i(out_{I_i[1]}, \ldots, out_{I_i[\ell]}) \neq out_i$ **output:** 1 ▷ Accept
Else Output: 0 ▷ Reject complaint if evaluation correct

This concise proof of misbehavior π consists in total of $\ell + 2$ Merkle proofs, and hence the complexity of the Judge is $O(\ell \log(m))$.

The first element $\pi_\phi \in \pi$ includes the Merkle proof that shows that label ϕ_i is indeed the label corresponding to the i-th gate in ϕ. The second element π_{out} includes a Merkle proof ρ_{out}, which is required to verify that z_i is indeed the i-th element in $\boldsymbol{z}$. Finally, π contains Merkle proof $\pi_{in}^1, \ldots, \pi_{in}^\ell$ for the ℓ encrypted input values of the gate with label ϕ_i. Given these Merkle proofs the judge algorithm verifies their correctness, decrypts z_i of the i-th operation ϕ_i into the output value out_i. Then, it checks whether op_i evaluated on the ℓ inputs yields into out_i. If all these checks pass it outputs 1; otherwise it outputs 0.

The ideal functionality $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ acts as a judge smart contract for session id id and interacts with the global $\mathcal{L}$ functionality and the parties $\mathcal{S}$ and $\mathcal{R}$. It locally stores addresses $pk_{\mathcal{S}}$ and $pk_{\mathcal{R}}$, price p, commitment c, decryption key k, Merkle tree root hashes r_z, r_ϕ and state s.

Initialize

(Round 1) Upon receiving $(init, id, p, c, r_\phi, r_z)$ from $\mathcal{S}$ with $p \in \mathbb{N}$, store r_ϕ, r_z, p, c, output $(initialized, id, p, r_\phi, r_z, c)$, set $s = initialized$ and proceed to the *reveal* phase.

(Round 2) Upon receiving $(accept, id)$ from $\mathcal{R}$ when $s = initialized$, send $(freeze, id, \mathcal{R}, p)$ to $\mathcal{L}$. If it responds with $(frozen, id, \mathcal{R}, p)$, set $s = active$ and output (accepted, id).

Reveal

(Round 3) Upon receiving $(reveal, id, d, k)$ from sender $\mathcal{S}$ when $s = active$ and $\text{Open}(c, d, k) = 1$, send $(revealed, id, d, k)$ to all parties and set $s = revealed$. Then proceed to the *payout* phase.

Otherwise if no such message from $\mathcal{S}$ was received, send message $(unfreeze, id, p, \mathcal{R})$ to $\mathcal{L}$ and abort.

Payout

(Round 4) Upon receiving a message m from the receiver $\mathcal{R}$ when $s = revealed$ set $s = finalized$ and do the following:

- If $m = (complain, id, \pi)$ s.t. $\mathit{Judge}(k, r_z, r_\phi, \pi) = 1$ send $(unfreeze, id, p, \mathcal{R})$ to $\mathcal{L}$, $(not\ sold, id)$ to $\mathcal{S}$ and terminate.
- Otherwise, send $(unfreeze, id, p, \mathcal{S})$ to $\mathcal{L}$, $(sold, id)$ to $\mathcal{S}$ and terminate.

(Round 5) Upon receiving message $(finalize, id)$ from the sender $\mathcal{S}$, when $s = revealed$, send message $(unfreeze, id, p, \mathcal{S})$ to $\mathcal{L}$. Then output $(sold, id)$ to $\mathcal{S}$ and terminate.

Figure 4: Ideal functionality $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ for the judge contract

4.2 The witness selling protocol

Now we can formally construct our protocol Π by using the three algorithms *Encode*, *Extract* and Judge. The protocol consists of the judge contract and the specification of the behavior of the two honest parties $\mathcal{S}$ and $\mathcal{R}$. In order to formally define the functions provided by the judge smart contract $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, we model it as an ideal functionality $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. The full description of $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ can be found in Figure 4 and the specification of the protocol is given in Figure 5.

Our protocol proceeds in three phases thereby closely following the structure of the smart contract $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. In the first round in the initialization phase, $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ takes as input from $\mathcal{S}$ the root elements r_z and r_ϕ as well as the commitment c. $\mathcal{R}$ receives $\boldsymbol{z}$ directly from $\mathcal{S}$, and r_z, r_ϕ from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. If these roots are computed correctly, then $\mathcal{R}$ accepts the contract. Additionally, if both parties agree and $\mathcal{R}$ has sufficient funds, p coins are blocked for this execution of the fair exchange protocol. Only after this phase is successfully executed, the judge contract is considered active. If during this phase, some party decides to abort the execution, this is not considered malicious.

In the reveal key phase, the contract expects $\mathcal{S}$ to reveal the key k, which allows to verify the commitment c. If $\mathcal{S}$ fails to send the *reveal* message, he is considered malicious and $\mathcal{R}$ can get his money back. Otherwise, if $\mathcal{S}$ revealed the key, $\mathcal{R}$ can decode the witness $\boldsymbol{x}$ by running $(\boldsymbol{x}, \pi) = \mathit{Extract}(\boldsymbol{z}, \phi, k)$.

In the next phase the payout of the coins can be triggered. If the witness is valid (i.e, $\pi = 0$) $\mathcal{R}$ sends message $(finalize, id)$ to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, which will trigger the smart contract to unfreeze the coins in $\mathcal{S}$'s favor. If instead *Extract* output a valid complaint, $\mathcal{R}$ sends a message $(complain, id, \pi)$ to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. If the extracted $\phi(\boldsymbol{x}) \neq 1$, the verification algorithm $\mathit{Judge}(k, r_z, r_\phi, \pi)$ will output 1 and thus accept the complaint and all coins are payed to $\mathcal{R}$. If $\mathcal{R}$ sends neither message, $\mathcal{S}$ can call the judge contract in round 5, to trigger the payout of coins.

4.3 Security

It remains to analyze the security of our protocol and provide an intuition why either party cannot break the fairness property for the other party as defined in Section 3.2.

Termination. The protocol terminates either after four rounds, in the payout phase, after $\mathcal{R}$ sends the *finalize*; or *complain* message or in the fifth round after $\mathcal{S}$ sent *finalize*. We distinguish the following termination cases for the protocol with an active judge contract and at least one honest party:

No abort: This case occurs when both parties act honestly. In this case, the protocol terminates in the *payout* phase, after $\mathcal{R}$ sends the *finalize* or *complain* message to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$.

$\mathcal{S}$ **aborts:** In case $\mathcal{S}$ does not reveal the key k, $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ will terminate in the reveal phase and make sure that $\mathcal{L}$ assigns all coins to $\mathcal{R}$.

$\mathcal{R}$ **aborts:** This case occurs when $\mathcal{R}$ does not react anymore after the key was revealed. In the fifth round $\mathcal{S}$ will then send $(finalize, id)$ to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ and the coins will be sent to $\mathcal{S}$.

Sender Fairness. Sender fairness means that $\mathcal{R}$ shall not be able to learn the witness $\boldsymbol{x}$ unless the honest sender $\mathcal{S}$ is guaranteed to be payed. From the secrecy property of our encoding scheme and the hiding property of the commitment, it follows directly that $\mathcal{R}$ cannot read the content of the encrypted witness before $\mathcal{S}$ publishes the decryption key k. At the point, when k is revealed, the

The protocol consists of descriptions of the behavior of the honest sender $\mathcal{S}$ and receiver $\mathcal{R}$.

Initialize

$\mathcal{S}$: Upon receiving input $(sell, id, \phi, p, \boldsymbol{x})$ in round 1, $\mathcal{S}$ samples $k \leftarrow \text{Gen}(1^\kappa)$, computes $(c, d) \leftarrow \text{Commit}(k)$ and $\boldsymbol{z} = Encode(\phi, \boldsymbol{x}, k)$. Then he sends $(sell, id, \boldsymbol{z}, \phi, c)$ to $\mathcal{R}$ and $(init, id, p, c, r_\phi, r_z)$ to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, where $r_\phi = root(Mtree(\phi))$ and $r_z = root(Mtree(\boldsymbol{z}))$. Then he continues to the *reveal* phase.

$\mathcal{R}$: Upon receiving input (buy, id, ϕ), $\mathcal{R}$ checks if he received message $(sell, id, \boldsymbol{z}, c)$ from $\mathcal{S}$ in round 1 and computes $r_z = root(Mtree(\boldsymbol{z}))$ and $r_\phi = root(Mtree(\phi))$. Upon receiving $(init, id, p, c, r_\phi, r_z)$ from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, $\mathcal{R}$ responds with $(accept, id)$ and proceeds to the *reveal* phase.

Reveal

$\mathcal{S}$: Upon receiving $(active, id)$ from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, $\mathcal{S}$ responds with $(reveal, id, d, k)$ and proceeds to the *payout* phase. If no $(active, id)$ message was received from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ in the third round, he instead terminates the protocol.

$\mathcal{R}$: Upon receiving $(revealed, id, d, k)$ from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, $\mathcal{R}$ proceeds to the *payout* phase. Otherwise, if no $(revealed, id, d, k)$ message was received from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ in round 4, $\mathcal{R}$ terminates the protocol.

Payout

$\mathcal{R}$: The receiver runs $(\boldsymbol{x}, \pi) = Extract(\phi, \boldsymbol{z}, k)$. If $\pi = \perp$, he sends message $(finalize, id)$ to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, otherwise he sends $(complain, id, \pi)$ instead. Then he outputs $(bought, id, \boldsymbol{x})$ and terminates the protocol execution.

$\mathcal{S}$: Upon receiving $(sold, id)$ or $(not\ sold, id)$ from $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$, $\mathcal{S}$ outputs this message and terminates the protocol. If no message has been received in round 4, he sends $(finalize, id)$ to $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$.

Figure 5: Formal protocol description for honest $\mathcal{S}$ and $\mathcal{R}$

coins have been successfully frozen for the execution of the smart contract $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. Now, that the exchange of the witness is initiated, an honest $\mathcal{S}$ is guaranteed to receive the payment, even if $\mathcal{R}$ aborts. Lastly it remains to show that a malicious $\mathcal{R}$ cannot forge a proof π, which is accepted by the judge contract, although $\mathcal{S}$ behaved honestly and $\phi(\boldsymbol{x}) = 1$. Forging such a proof would require $\mathcal{R}$ to forge a Merkle proof over a false element of $\boldsymbol{z}$. Informally speaking, this is not possible unless he finds a collision in the hash function H.

Receiver Fairness. If $\mathcal{S}$ sends the encoding $\boldsymbol{z}$, $\mathcal{R}$ continues with the protocol until the coins are frozen for the execution of the smart contract $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. To prove fairness for an honest receiver $\mathcal{R}$, we have to show that a malicious sender $\mathcal{S}$ cannot send a wrong witness $\boldsymbol{x}' \notin L$ such that $\mathcal{R}$ is not able to generate a correct proof of misbehavior, which is accepted by the $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ contract. In order to successfully execute such an attack, $\mathcal{S}$ must be able to find an encoding $\boldsymbol{z}$ such that $Extract(\boldsymbol{z}, k, \phi) = (\boldsymbol{x}', \pi')$ but the judge on input of π does not accept the complaint. The probability of $\mathcal{S}$ finding such values is negligible, since this would require him to break collision resistance of the underlying hash function. Therefore, $\mathcal{R}$ is guaranteed, that as soon as $\mathcal{S}$ publishes k he will either receive the witness $\boldsymbol{x}$ with $\phi(\boldsymbol{x}) = 1$, or he has the guarantee that by executing $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ on a valid proof of misbehavior he will get $p + q$ coins. Therefore Π satisfies receiver fairness.

Formal security definition. In order to formally state the security of our protocol Π we use the previously introduced GUC-style security notion.

Theorem 1. *There exists an efficient two party protocol* Π*, which GUC-realizes the ideal fair exchange functionality* $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ *in the judge smart contract* $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$*-hybrid world, where* $\mathcal{H}$ *is modeled as a global programmable random oracle.*

In the Appendix we formally prove the Theorem 1 in the global UC model using the judge smart contract $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ and a global programmable random oracle $\mathcal{H}$. We start by giving some background on the global programmable random oracle model (c.f. Appendix A) and provide definitions of the cryptographic building blocks and explain how to construct them in the programmable random oracle (c.f. Appendix B). In Appendix C follows a brief overview about simulation in the UC Model and a detailed description of the four simulators that are needed to prove the above theorem.

5 APPLICATION AND PERFORMANCE

The efficiency of the protocol Π can vary for different circuits ϕ, where circuits with small instruction alphabets Γ and fan-in ℓ are the most promising candidates. For such circuits the overhead of the encoding is small and the judge contract can run at low costs. In this section we show how to use our general protocol for digital file sale and highlight its use for distributed peer-to-peer file sharing. Additionally, we will give performance indicators for our protocol and costs of our implementation for the judge contract in Ethereum.

Efficient and fair file transfer. Our protocol can be used whenever two parties want to exchange data that is identified via its Merkle hash. In this case the witness that the receiver wants to buy would be the file $\boldsymbol{x} = (x_1, \ldots, x_n)$, which is split up into n parts, where each x_i is of some short length and the circuit ϕ checks if the Merkle Hash root of this file equals some given value h (i.e., $\phi(\boldsymbol{x}) = 1, if\ f\ Mtree(\boldsymbol{x}) = h$). The instruction alphabet of ϕ consists of the operations $H(x, y)$ and $eq(x, h)$, where H is the Hash function used for the Merkle tree and $eq(x, y)$ is a function that outputs 1 if $x = y$. Figure 6 shows such a circuit for a file with $n = 8$ elements of size λ. Let us next provide some further details on the application of our protocol for fair file exchange with coins.

Instantiated with the above "file hashing" circuit our protocol provides an elegant solution for the so-called *free-riders problem*, which is a major drawback in distributed file sharing systems. The free-riding problem states that a system for digital file exchange suffers if enough peers only benefit without providing content. Surveys like [42, 43] show that free-riding is a common problem

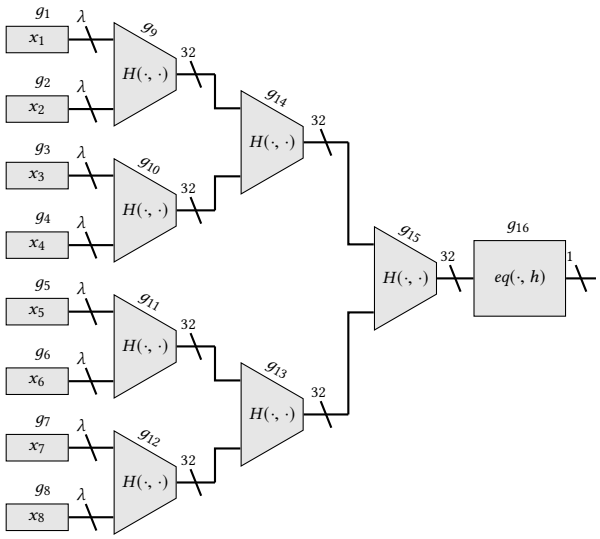


Figure 6: Exemplary circuit ϕ for exchange of $\boldsymbol{x} = (x_1, \ldots, x_8)$ with hash $h = root(Mtree(\boldsymbol{x}))$

in decentralized systems whenever creating identities is cheap and users can dynamically join and leave the system. In [1] researchers found that at one point 75% of users of the popular platform gnutella were free-riders. Some incentive mechanisms have been proposed to make free-riding less attractive [27], e.g., by introducing payments and let users pay small fees to the senders of files. However, such approaches do not address the case when a malicious user offers content that is incorrect (i.e., it does not belong to the file that the receiver intends to download).

A natural solution to the free-rider problem is to use cryptocurrencies that support smart contracts since they provide a decentralized trust platform which handles payments. One possible solution already discussed in the introduction is to use ZKCP, but this only works well for small inputs as otherwise the users would suffer from huge efficiency penalties. For instance, in [26] the authors show that proving in zero knowledge the correctness of a single evaluation of a hash function (SHA256) on a witness of 64 bytes requires 3 MB of additional data transfer between the parties. On the other hand our protocols also solves the fairness problems of digital file exchange, but results only in small overheads for the users in terms of computation and data transfer.

5.1 Implementation

To benchmark the runtime and execution costs of our protocol, we implemented the protocol for the file sale application[5] using the file sale circuit (cf. Figure 6). A nice property of this circuit is the small size of the instruction alphabet ($|\Gamma| = 2$), and the small fan-in of operations ($\ell = 2$). This allows us to provide a highly efficient smart contract implementation for this particular use case. The advantage of the small instruction alphabet is that the contract can derive the operation of gate g_i from the index i (indeed there is only one operation in the entire circuit ϕ except for the very last instruction). This allows us to implement the verification without committing, sending and verifying ϕ_i. Additionally, for the special case of a Merkle tree circuit we have that the input to all gates (i.e., hash function evaluations) are natural siblings in the encoding in $\boldsymbol{z}$. This means that in the concise proof of misbehavior to verify the correct evaluation of one hash functions on two inputs, we only need one (slightly modified) Merkle proof verification, which verifies both input values in one step. Thus, the proof π only includes two input values of at most length λ, one output hash of length μ and two Merkle proofs – one for the two input elements and one for the output of the gate.

In our implementation the users have the ability to change the parameters of the protocol, namely the number of file chunks n, which directly relates to two other parameters in our application: the length of each file chunk $|x_i| = \lambda$ and the depth of the Merkle Tree δ (again we assume a full tree for simplicity). We can observe the following relation of the parameters:

$$\lambda = \frac{|\boldsymbol{x}|}{n} = \frac{|\boldsymbol{x}|}{2^{\delta}}$$

The hash function optimized in the Ethereum virtual machine language is `keccak256`, which outputs hashes of size $\mu = 32$ bytes. Since the instruction set of Solidity is currently limited, but provides a relatively cheap (in gas costs) and easy hashing, we use this hash function to implement our encryption scheme. Since the judge contract needs the possibility to decrypt each element $z_i \in \boldsymbol{z}$ without knowledge of the whole vector $\boldsymbol{z}$ we use a variant of the plain counter mode for symmetric encryption, for which `keccak256` is evaluated on input of a key k and index i, and the ciphertext is the bitwise XOR of the plaintext with this hash output taking as input the key and the current counter. From the construction of the encryption scheme, it follows that the file chunk length λ should be a multiple of 32 bytes to allow efficient encryption and decryption.

The judge contract implementations offers four different options for $\mathcal{R}$ to call during the *payout* phase. The function *nocomplain* allows $\mathcal{R}$ to accept the file transfer and directly send p coins to $\mathcal{S}$, the *copmlainAboutRoot* function is used whenever $\mathcal{R}$ complains about a false output of the circuit, namely that $z_m \neq h$. The functions *complainAboutLeaf* and *complainAboutNode* allow $\mathcal{R}$ to complain about the computation of two input gates g_i, g_{i+1}, $i \in 1 \ldots, n$ or the computation of some other gates g_i, g_{j+1} where $n < j \leq m$ respectively. The reason for these different complain functions is that each of them requires a differently sized input.

5.2 Benchmarks

The costs for running the protocol consist of the fees, which need to be payed to the miners of the cryptocurrency – in case of our implementation this will be Ethereum. In the first round the sender deploys the *FairSwap* contract including the Ethereum addresses of $\mathcal{S}$ and $\mathcal{R}$, the price value p, the commitment c and the roots r_ϕ and $r_{\boldsymbol{z}}$. The main gas costs result from this deployment, which costs roughly 1050000 gas, which for a gas price of 3 GWei translates to 0.00315 Ether or 1.57 USD for an exchange rate of 500 USD/Ether. The price for the execution of the functions *deploy*, *accept*, *reveal*, *refund* and *no complain* stays almost constant for different parameters, but the

[5] The sourcecode of the solidity contract can be found at github.com/lEthDev/FairSwap

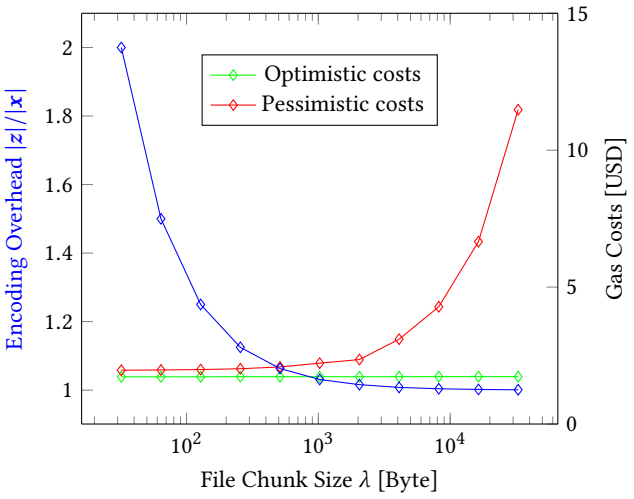


Figure 7: Costs and encoding size for different values of λ

cost of the complain function varies highly, depending on the kind of complaint, the choice of the file chunk size λ and the Merkle depth δ. The data, which needs to be sent to the blockchain (as part of π) increases with the size of λ. Figure 7 shows the costs for optimistic (green) and pessimistic (red) execution costs for different file chunk lengths for a file of one GByte size. Optimistic means, that the protocol continues until the *payout* phase, where $\mathcal{R}$ accepts the encoding without complaint, whereas pessimistic means, that $\mathcal{R}$ complains to the judge contract about the wrong computation of some input. The costs for the complain originate from the length of the concise proof of misbehavior π, which needs to be sent to the contract and evaluated on-chain:

$$\begin{aligned}|\pi| &= |z_{in1}| + |z_{in1}| + |z_{out}| + |\rho_{in}| + |\rho_{out}| \\ &\leq 2\mu \times \lambda + \mu + 2\delta \times \mu\end{aligned}$$

Figure 7 illustrates that even for very large file chunk sizes, the costs for optimistic execution is close to constant around 1.73 USD, where the cost for pessimistic execution increases linearly in the length of the file chunks. We highlight that using different cryptocurrencies can decrease the price for execution even further. In Ethereum classic[6], a well known fork of the Ethereum blockchain, the cost for optimistic execution is only fractions of cents.

The heavy computation of the protocol is executed in round 1 and 3, by both the sender and the receiver in the two algorithms *Extract* and *Encode*. We will only take a closer look at the performance of the sender, since the receiver will perform almost identical computations only in reverse order. To encode file $\boldsymbol{x} = (x_1, \ldots, x_n)$, $\mathcal{S}$ needs to first generate $M = Mtree(\boldsymbol{x})$ and store all intermediate hashes. The result is $n - 1$ elements of size μ. Next, he encrypts each file chunk (which requires $n \times \lambda$ hashes in total) and each hash from the Merkle tree M ($n - 1$ hashes). It remains to compute the Merkle root of the combined encoding $r_{\boldsymbol{z}} = Mtree(z_1, \ldots, z_m)$, for which he needs to hash $2m$ Elements. Therefore we get the following estimates:

$$\begin{aligned}|\boldsymbol{z}| &= |\boldsymbol{x}| + (n-1)32 \text{ Bytes} = n \times \lambda \times 32 \text{ Bytes} \\ \text{Runtime of } Extract &= n \times \lambda \times O(H)\end{aligned}$$

The size of $\boldsymbol{z}$ therefore can be used as an indicator for the performance of the algorithm *Extract* and additionally affects the communication complexity, since it needs to be transfered in the first protocol message which is the longest message in the protocol. We did not optimize the implementation of the algorithms *Encode*, *Extract* for runtime but it shall be noted that we reach an encoding throughput of approximately 2 MB per second in a straightforward node.js implementation running on single core of a 2.67 GHz Intel©Core i7 CPU with 8 GB of RAM. Clearly, these estimations indicate that the performance of the protocol is optimal for small λ. Figure 7 illustrates his trade-off between the costs of the protocol and its performance, measured in the overhead of encoding size in comparison to file length $\frac{|\boldsymbol{z}|}{|\boldsymbol{x}|}$.

The protocol can be executed in 4 rounds, where each round requires sending a message to the blockchain. The round ends, when the message is accepted by the miners and included in a block. Cryptocurrencies ensure, that a correct message (with sufficient fees) is eventually included in the blockchain, but this process might take some time. We denote this maximal round duration with Δ, therefore the judge contract will have timeouts Δ to measure whether some message has been sent or not. The exact value of this parameters is chosen by the parties and depend on the congestion of the blockchain, the amount of fees, they are willing to pay, their availability and the number of blocks they require to succeed a transaction in order to consider it valid. We note that the minimum duration of the protocol is 4 rounds, which in Ethereum can be executed in only a few minutes, as long both parties agree.

5.3 Repeated fair exchange

For reasonably small values of λ the main cost for running fair file sale consists of the deployment costs. Whenever two parties want to repeatedly run protocol Π, they can save costs by slightly modifying the contract to decouple deployment of code and the initialization function. This allows them to re-use the same contract for repeated executions of file sale. Deploying such a modified contract costs 0.20 USD more than the same step in the standalone file sale, but every following repetition of the protocol re-using same contract only costs 1.60 USD to execute. But even 1.60 USD is a high price to pay for fees in e.g. the distributed file sharing case, where these fees would have to be payed for every file transfer. Another problem is that the execution of the protocol requires four slow blockchain interactions. For distributed file sharing this means the execution of the protocol will probably last at least a few minutes.

To minimize gas costs and confirmation time for repeated execution, we propose to run the judge contract described above off-chain in a so called state-channel. State channels are an extension of payment channels and allow users to execute arbitrary smart contracts off-chain without requiring interaction with the blockchain. Constructions for state channels have been proposed in earlier works, e.g., in [21, 37]. Hence, we only describe here how they can be used for our application, and provide a more technical specification in Appendix D. In our basic system, the seller and the buyer open a

[6] https://ethereumclassic.github.io/

state channel by blocking money in a contract such that the money can only be payed out (before some timeout) if both parties agree or by forcing the execution of the judge contract. Since the parties want to execute multiple file sales, they freeze enough money in the channel.

Now the users can run multiple fair exchange executions without costly and time consuming interactions with the blockchain. If however at some point one of the parties starts to behave maliciously (e.g., a sender does not provide the secret key for the i-th repetition of the protocol) the parties can always execute the contract for this repetition on-chain and settle their disagreement in a fair way. That is, the contract is executed on the blockchain and the funds are distributed to the parties within some predefined time.

The above approach suffers from the drawback that when $\mathcal{S}$ and $\mathcal{R}$ want to run a fair exchange, they first need to open a state channel between each other on the blockchain. When a system has many senders and receivers that moreover may take different roles during the lifetime of the system, this would result into large overheads because opening a state channel requires a costly interaction with the blockchain. To minimize these costs, we may integrate our system into a *state channel network*. A state channel network allows to compose multiple state channels into new longer state channels without the need to interact with the blockchain. For illustration, suppose that Alice has a state channel opened with Bob, and Bob has a state channel opened wit Carol. In a state channel network, Alice then also has an "implicit" state channel with Carol, which is executed via the intermediate Bob. Notice that using the state channel between Alice and Carol does not require any on-chain transactions. Using state channel networks, a sender Alice may thus execute a fair exchange with Carol instantaneously via the intermediate Bob without the need to explicitly open a state channel with Carol on the blockchain. We notice however that state channel networks are currently only under development and no fully functioning system has been deployed yet. We leave it as an interesting direction for future research to integrate fair file sale within a state channel network.

5.4 Fault Attribution and Denial of Service Attacks

As soon as the judge contract is active, $\mathcal{S}$ can abort the protocol execution without being penalized. Note that in the simplified version of the protocol which we considered so far, this does not hold for sending a false witness. This allows the sender to force $\mathcal{R}$ to freeze coins. The sender $\mathcal{S}$ could initiate the protocol without knowing the actual file which is sold. This cannot be prevented because $\mathcal{R}$ is only allowed to learn the file at the end of the protocol, when the key is revealed. But he needs to freeze his money in the beginning. A solution for this problem is to penalize $\mathcal{S}$ when he misbehaves. This can be achieved by letting $\mathcal{S}$ also freeze some money for the execution of the contract. If he behaves correctly (i.e. follows the protocol description and provides a correct circuit) the money will be send back to him but if he does not, the money is instead sent to $\mathcal{R}$. This means the contract itself allows *fault attribution* and compensates attacked parties.

But even when penalties are added, we face the problem that $\mathcal{R}$ could run a *Denial of Service (DoS)* attack towards $\mathcal{S}$. This way DoS attacks cannot be prevented but it will make them more expensive and the attacked parties will be compensated. The receiver can request multiple files from senders without then accepting the execution of the contract, which forces the sender to (a) deploy and initialize the contract on the blockchain and (b) force him to compute $z = \mathit{Extract}(\phi(\boldsymbol{x}))$. The financial risk of the sender can be mitigated by changing the protocol to let $\mathcal{R}$ deploy the contract and thus pay the fees but this only shifts the risk to $\mathcal{R}$. In order to mitigate the risk of attack (b), we propose that the sender precomputes z and uses the same encoding/key combination for all protocol executions (as long as the key was not revealed). Therefore if $\mathcal{R}$ repeatedly aborts after the first round, $\mathcal{S}$ only has the overhead of computing z once.

6 CONCLUSION

In this paper we presented a protocol, which allows fair sale of a witness where a judge smart contract verifies concise proofs of misbehavior. These proofs are short statements which a receiver generates if the delivered witness does not satisfy a circuit ϕ. We present three algorithms, *Encode*, *Extract* and *Judge* where *Encode* is used by the sender $\mathcal{S}$ to generate an encoding of $\boldsymbol{x}$ and each step of the evaluation of the circuit $\phi(\boldsymbol{x})$. Using this encoding and the decryption key k, the receiver can run *Extract* to learn the witness $\boldsymbol{x}$. If this witness does not satisfy the circuit, namely $\phi(\boldsymbol{x}) \neq 1$, *Extract* outputs a concise proof of misbehavior, which can be sent to the judge contract. We show that the verification of this proof reveals if $\phi(\boldsymbol{x}) \neq 1$, and the contract uses this information to pay out the money correctly. We argue why our protocol satisfies sender and receiver fairness and terminates after at most 5 rounds. In the appendix we provide a formal proof of security in the GUC model and show that our construction securely realizes the ideal functionality for fair exchange $\mathcal{F}_{\mathsf{cfe}}^{\mathcal{L}}$. We provide an implementation of the judge contracts and show that our protocol works efficiently for large files and can be executed at low costs.

ACKNOWLEDGMENTS

This work has been supported by the Foundation for Polish Science grant TEAM/2016-1/4 founded within the UE 2014-2020 Smart Growth Operational Program, by the Polish National Science Center grant 2014/13/B/ST6/03540, and by the German Research Foundation (DFG) as part of project S7 within the CRC 1119 CROSSING and by the Emmy Noether Program FA 1320/1-1.

REFERENCES

[1] Eytan Adar and Bernardo A Huberman. 2000. Free riding on Gnutella. *First monday* 5, 10 (2000).

[2] Ian Allison. 2016. Ethereum's Vitalik Buterin explains how state channels address privacy and scalability. https://tinyurl.com/n6pgct.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, California, USA, 443–458. https://doi.org/10.1109/SP.2014.35

[4] The Internet Archive. 2012. Over 1,000,000 Torrents of Downloadable Books, Music, and Movies. https://blog.archive.org/2012/08/07/over-1000000-torrents-of-downloadable -books-music-and-movies/.

[5] N. Asokan, Victor Shoup, and Michael Waidner. 1998. Optimistic Fair Exchange of Digital Signatures (Extended Abstract). In *Advances in Cryptology – EUROCRYPT'98 (Lecture Notes in Computer Science)*, Kaisa Nyberg (Ed.), Vol. 1403. Springer, Heidelberg, Germany, Espoo, Finland, 591–606.

[6] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO 2017*. 324–356.

[7] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. 2016. Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. In *ESORICS 2016: 21st European Symposium on Research in Computer Security, Part II (Lecture Notes in Computer Science)*. Springer, Heidelberg, Germany, 261–280. https://doi.org/10.1007/978-3-319-45741-3_14

[8] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. 2016. NIZKs with an Untrusted CRS: Security in the Face of Parameter Subversion. In *Advances in Cryptology – ASIACRYPT 2016, Part II (Lecture Notes in Computer Science)*. Springer, Heidelberg, Germany, 777–804. https://doi.org/10.1007/978-3-662-53890-6_26

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *Advances in Cryptology – CRYPTO 2014, Part II (Lecture Notes in Computer Science)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8617. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 276–294. https://doi.org/10.1007/978-3-662-44381-1_16

[10] Iddo Bentov and Ranjit Kumaresan. 2014. How to Use Bitcoin to Design Fair Protocols. In *Advances in Cryptology – CRYPTO 2014, Part II (Lecture Notes in Computer Science)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8617. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 421–439. https://doi.org/10.1007/978-3-662-44381-1_24

[11] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract). In *20th Annual ACM Symposium on Theory of Computing*. ACM Press, Chicago, Illinois, USA, 103–112.

[12] Sean Bowe. 2016. Pay-to-sudoku. https://github.com/zcash/pay-to-sudoku

[13] Christian Cachin and Jan Camenisch. 2000. Optimistic Fair Secure Computation. In *Advances in Cryptology – CRYPTO 2000 (Lecture Notes in Computer Science)*, Mihir Bellare (Ed.), Vol. 1880. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 93–111.

[14] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. 2018. The Wonderful World of Global Random Oracles. Cryptology ePrint Archive, Report 2018/165. https://eprint.iacr.org/2018/165.

[15] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. 2017. Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services. In *ACM CCS 17: 24th Conference on Computer and Communications Security*. ACM Press, 229–243.

[16] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (2000), 143–202.

[17] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. 2014. Practical UC security with a Global Random Oracle. In *ACM CCS 14: 21st Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 597–608.

[18] Ran Canetti, Ben Riva, and Guy N. Rothblum. 2011. Practical delegation of computation using multiple servers. In *ACM Conference on Computer and Communications Security*. ACM, 445–454.

[19] Alessandro Chiesa, Eran Tromer, and Madars Virza. 2015. Cluster Computing in Zero Knowledge. In *Advances in Cryptology – EUROCRYPT 2015, Part II (Lecture Notes in Computer Science)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9057. Springer, Heidelberg, Germany, Sofia, Bulgaria, 371–403. https://doi.org/10.1007/978-3-662-46803-6_13

[20] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. 2018. FairSwap: How to fairly exchange digital goods. Cryptology ePrint Archive, Report 2018/734. https://eprint.iacr.org/2018/734. Extended version of this paper.

[21] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2017. Perun: Virtual Payment Hubs over Cryptocurrencies. Cryptology ePrint Archive, Report 2017/635. https://eprint.iacr.org/2017/635, accepted to IEEE S&P 2019.

[22] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. 2010. Random Oracles with(out) Programmability. In *Advances in Cryptology – ASIACRYPT 2010 (Lecture Notes in Computer Science)*, Masayuki Abe (Ed.), Vol. 6477. Springer, Heidelberg, Germany, Singapore, 303–320.

[23] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. 2016. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *USENIX Security Symposium*. USENIX Association, 1069–1083.

[24] Oded Goldreich. 2006. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA.

[25] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th Annual ACM Symposium on Theory of Computing*, Alfred Aho (Ed.). ACM Press, New York City,, New York, USA, 218–229.

[26] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 13: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 955–966.

[27] Murat Karakaya, İbrahim Körpeoğlu, and Özgür Ulusoy. 2008. Counteracting free riding in Peer-to-Peer networks. *Computer Networks* 52, 3 (2008), 675–694.

[28] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *TCC 2013: 10th Theory of Cryptography Conference (Lecture Notes in Computer Science)*, Amit Sahai (Ed.), Vol. 7785. Springer, Heidelberg, Germany, Tokyo, Japan, 477–498. https://doi.org/10.1007/978-3-642-36594-2_27

[29] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. 2016. Fair and Robust Multi-party Computation Using a Global Transaction Ledger. In *EUROCRYPT 2016*. 705–734.

[30] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. 2015. Fair and Robust Multi-Party Computation using a Global Transaction Ledger. Cryptology ePrint Archive, Report 2015/574. Accepted to EUROCRYPT'16, http://eprint.iacr.org/.

[31] Ranjit Kumaresan and Iddo Bentov. 2016. Amortizing Secure Computation with Penalties. In *ACM CCS 16: 23rd Conference on Computer and Communications Security*. ACM Press, 418–429.

[32] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. 2016. Improvements to Secure Computation with Penalties. In *ACM CCS 16: 23rd Conference on Computer and Communications Security*. ACM Press, 406–417.

[33] Alptekin Küpçü and Anna Lysyanskaya. 2012. Usable optimistic fair exchange. *Computer Networks* 56, 1 (2012), 50–63.

[34] Morgan G. I. Langille and Jonathan A. Eisen. 2010. BioTorrents: A File Sharing Service for Scientific Data. *PLoS ONE* 5, 4 (04 2010), 1–5. https://doi.org/10.1371/journal.pone.0010071

[35] Henry Z. Lo and Joseph Paul Cohen. 2016. Academic Torrents: Scalable Data Distribution. *CoRR* abs/1603.04395 (2016). http://arxiv.org/abs/1603.04395

[36] Lee Mathews. 2015. Windows 10 lets you torrent updates and apps. http://www.geek.com/microsoft/windows- 10-lets-you-torrent-updates-and-apps-1618036.

[37] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment Channels that Go Faster than Lightning. *CoRR* abs/1702.05812 (2017). http://arxiv.org/abs/1702.05812

[38] Jesper Buus Nielsen. 2002. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case. In *Advances in Cryptology – CRYPTO 2002 (Lecture Notes in Computer Science)*, Moti Yung (Ed.), Vol. 2442. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 111–126.

[39] Chris O'Falt. 2015. BitTorrent to Now Offer Legal Movie Downloads. http://www.hollywoodreporter.com/news/bittorrent-offer-legal-movie-downloads-769733.

[40] Henning Pagnia and Felix C Gärtner. 1999. *On the impossibility of fair exchange without a trusted third party*. Technical Report. Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany.

[41] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, California, USA, 238–252.

[42] Muntasir Raihan Rahman. 2009. A survey of incentive mechanisms in peer-to-peer systems. *Cheriton School of Computer Science, University of Waterloo, Tech. Rep. CS-2009-22* (2009).

[43] Lakshmish Ramaswamy and Ling Liu. 2003. Free riding: A new challenge to peer-to-peer file sharing systems. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. IEEE, 10–pp.

[44] Jason Teutsch and Christian Reitwiessner. 2018. TrueBit – a scalable verification solution for blockchains. https://truebit.io/.

[45] Torrentfreak. 2014. UK Government uses Bittorrent to Share Public Spending Data. https://torrentfreak.com/uk-government- uses-bittorrent- to-share-public-spending-data-100604/.

[46] Bitcoin Wiki. 2018. Zero Knowledge Contingent Payment. https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.

[47] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Toronto, Ontario, Canada, 162–167.

A PROGRAMMABLE GLOBAL RANDOM ORACLE

A common model for crpytographic proofs is the random oracle model. This model assumes that hash functions can be modeled to return perfectly random values, which can not be predicted. For this reason hash functions are instantiated with random oracles. For proving security in the UC-model, we use a random oracle ideal functionality $\mathcal{H}$ which (unless otherwise instructed) responds to all queries with uniformly random sampled values $r \leftarrow \{0,1\}^\mu$ and stores all query response pairs (q, r) in the set Q. If the query value q has been answered before, such that $(q, r) \in Q$ is stored, $\mathcal{H}$ responds with r.

The $\mathcal{H}$ functionality is the global random oracle with restricted programming and observability, which takes as input queries $q \in \{0,1\}^*$ and outputs values $r \in \{0,1\}^\mu$. Internally it stores initially empty sets Q, P and a set Q_{id} for all sessions id.

Query

Upon receiving message (query, id, q) from a party of session id' proceed as follows:

- If $(id, q, r) \in Q$ respond with (query, q, r).
- If $(id, q, r) \notin Q$ sample $r \in \{0,1\}^\mu$, store (id, q, r) in Q and respond with (query, q, r).
- If the query is made from a wrong session ($id \neq id'$), store (q, r) in Q_{id}.

Program

Upon receiving message $(\text{program}, id, q, r)$ by the adversary $\mathcal{A}$ check if (id, q, r') is defined in Q. If this is the case, abort. Otherwise, if $r \in \{0,1\}^\mu$ store (id, q, r) in Q and (id, q) in P.
Upon receiving message $(\text{isPrgrmd}, q)$ from a party of session id check if $(id, q) \in P$. If this is the case respond with $(\text{isPrgrmd}, 1)$.

Observe

Upon receiving message (observe) from the adversary of session id respond with $(\text{observe}, Q_{id})$.

Figure 8: The ideal restricted programmable and observable random oracle functionality $\mathcal{H}$ [14]

It is common for UC security proofs to work with a *programmable* random oracle. Programmability means that the ideal UC adversary *Sim* can control the random oracle and program its hashes to specific responses. Additionally, it can see all queries made by the environment $\mathcal{Z}$ to the random oracle. Traditionally, such a random oracle is modeled as a local functionality, which is in control of the Simulator. This again implies that every unique protocol execution has its own local disjunct hash function. Since we want to explicitly allow composition of multiple protocols, we follow the argument of [14, 17] that such local functionalities are not a good model for standard hash function like keccak. A global functionality would respond to all queries in all sessions with the same values, which cannot be done with local random oracle functionalities.

Our construction models as a global functionality $\mathcal{H}$ following the works of [14, 17]. Every party in this model has oracle access to a global functionality $\mathcal{H}$ which represents our idealized hash function. Since we require programmability in the GUC model, we follow the work of [14] and model $\mathcal{H}$ as a restricted programmable and observable random oracle as defined in Figure 8. This functionality allows parties to simulate and observe only queries made from their own session (denoted with session and contract identifier id).

The first feature of the random oracle is that parties can query it on some value q, by calling (query, id, q). For simplicity we will write $r \leftarrow \mathcal{H}(q)$. The random oracle will return a uniformly sampled value or an already existing value r from the set Q.

But next to this straightforward functionality, we require *programmability*, which is a special property needed for proving security in the UC model. Programmability means that the adversary is allowed to fix the response of the oracle for certain queries if they have not been queried before by sending $(\text{program}, q, r)$ (we say the adversary *programs* the random oracle).

Programmable random oracles are a useful and practical tool in many UC simulations, which has been studied intensively before in the non global setting [22, 38]. In the local UC model, the adversary is always the simulator, who requires these properties to simulate indistinguishable commitments. In the global UC model, there might be multiple executions that all interact with the same global random oracle. Note, that the influence of any global functionality is not only for the simulator of a single execution, but also applies for all adversaries from different executions[7].

Intuitively, this adversarial power seems to break security of schemes that are based on this functionality since any adversary is allowed to program collisions, but we will show that this is not true. As protection against this adversarial power parties have the ability to verify if some response of the random oracle has been programmed by calling $\mathcal{H}(\text{isPrgrmd}, r)$. If $\mathcal{H}$ responds with 1, the parties know that the values is programmed and reject the value.

Additionally to the restricted programmability the functionality $\mathcal{H}$ allows leakage of all illegitimate queries, which were made by the environment over the adversary, by sending $\mathcal{H}(\text{observe})$. The functionality $\mathcal{H}$ will respond with the set Q_{id} which contains all illegitimate queries made from that session. This includes all queries from adversaries that are not from the desired session. For more information about the construction and properties of this ideal functionality we refer the reader to [14].

B CRYPTOGRAPHIC BUILDING BLOCKS

In this section, we give a detailed explanation of the properties we need from the encryption and commitment functions used in our protocol. Additionally, we will construct these schemes in the programmable random oracle model and show why they provide the required equivocability and extractability properties.

Note, that it is not easily possible to use a UC-style commitment and encryption functionalities here, since our smart contract hybrid functionality needs to run the open/decrypt procedure. Since in the UC-model functionalities are permitted from interacting with other functionalities, this prevents us from using ideal functionalities here.

Commitment Scheme. Let κ be the security parameter and $(a||b)$ denote the concatenation of two values a and b. Then we construct a commitment scheme (Commit, Open) in the global programmable random oracle model as follows:

To show that this scheme is hiding, it needs to hold that any ppt algorithm $\mathcal{A}$ cannot distinguish two commitments. From the randomness of the outputs of $\mathcal{H}$ it follows that this construction is hiding because the output of $\mathcal{H}(x) \approx_c \mathcal{H}(y)$ is indistinguishable if the $\mathcal{A}$ does not know (or programmed) $\mathcal{H}(x)$ or $\mathcal{H}(y)$ (which by

[7] In our case the the environment could access the global random oracle through the adversary of another session or protocol execution and program collisions.

Algorithm 7 Algorithm Commit

Input: $x \in \{0,1\}^*$
$d \leftarrow \{0,1\}^\kappa$ s.t. $\mathcal{H}(\text{isPrgrmd}, x||d) \neq 1$ ▷ choose d uniformly at random
$c \leftarrow \mathcal{H}(x||d)$ ▷ query the oracle on $x||d$
Output: (c, d)

chance only happens with a negligible probability for computationally bounded distinguishers). If d is chosen uniformly at random from domain $\{0,1\}^\kappa$ and κ large enough, any $\mathcal{A}$ cannot distinguish $\text{Commit}(x)$ from $\text{Commit}(y)$ if he does not know the opening d.

Algorithm 8 Algorithm Open

Input: $c \in \{0,1\}^\mu$, $d \in \{0,1\}^\kappa$
$c' \leftarrow \mathcal{H}(x||d)$ ▷ query the oracle on $x||d$
if $c == c'$ **and** $\mathcal{H}(\text{isPrgrmd}, x||d) \neq 1$ **then**
$b = 1$ ▷ ensure that the commitment was not programmed
$b = 0$ ▷ otherwise reject the opening
Output: b

In order to break the binding property, an adversary $\mathcal{A}$ needs to find a collision $\mathcal{H}(x) = \mathcal{H}(y)$, without programming $\mathcal{H}$. Since the outputs of $\mathcal{H}$ are uniformly distributed, the best strategy for $\mathcal{A}$ is to guess values and query $\mathcal{H}$ on them. If μ is large, this is hard for computationally bounded adversaries, since they can only make a polynomial in κ number of queries to $\mathcal{H}$. Thus, the scheme is computationally binding.

Encryption Scheme. The second cryptographic building block, which we need to construct for our protocol is a symmetric encryption scheme, which satisfies the IND-CPA security property. We instantiate our encryption scheme as follows. To encrypt to a tuple $\boldsymbol{x} = (x_1, \ldots, x_m)$ randomly chose $k \leftarrow \{0,1\}^\kappa$. Then compute $\boldsymbol{z} = z_1, \ldots, z_n$ as follows:

Algorithm 9 Algorithm Enc

Input: $\boldsymbol{x} = (x_1, \ldots, x_m)$, **s.t.** $\forall i \in [m] : |x_i| = \lambda$
$k \leftarrow \{0,1\}^\kappa$ s.t. $\forall i \in [m] : \mathcal{H}(\text{isPrgrmd}(k||i)) \neq 1$ ▷ choose k uniformly at random
for each $x_i \in \boldsymbol{x}$ **do**
$k_i = \mathcal{H}(k||i)$ ▷ generate i-th key
$z_i = k_i \oplus x_i$ ▷ xor key and plaintext
Output: $\boldsymbol{z} = (z_1, \ldots, z_n)$

Algorithm 10 Algorithm Dec

Input: $\boldsymbol{z} = (z_1, \ldots, z_m)$, **s.t.** $|z_i| = \lambda$ **and** $k \in \{0,1\}^\kappa$
for each $z_i \in \boldsymbol{z}$ **do**
$k_i = \mathcal{H}(k||i)$ ▷ generate i-th key
if $\mathcal{H}(\text{isPrgrmd}(k||i))$ **then**
Terminate and Output $\perp$ ▷ reject if any key is programmed
else
$x_i = k_i \oplus z_i$ ▷ xor key and ciphertext
Output $\boldsymbol{x} = (x_1, \ldots, x_n)$

The decryption is only accepted by the receiver if none of the results of $\mathcal{H}$ were programmed. Therefore, the receiver queries $\mathcal{H}(\text{isPrgrmd}(k||i))$ and rejects the result if for any i the random oracle responds with *true*.

This scheme is correct, i.e. $\text{Dec}(k, \text{Enc}(k, \boldsymbol{x})) = \boldsymbol{x}$ since for all $x_i \in \boldsymbol{x}$ it holds that $\mathcal{H}(k||i) \oplus z_i = \mathcal{H}(k||i) \oplus \mathcal{H}(k||i) \oplus x_i = x_i$. As long as all outputs of the random oracle are uniformly distributed over $\{0,1\}^\mu$, the ciphertexts are indistinguishable, which means that the scheme satisfies the chosen plaintext indistinguishability (IND-CPA). This property holds for as long as no programmed values are queried, which does not happen when the encryption was done honestly.

In our protocol we consider the special case where we commit to the key for the encryption. The authors of [38] show how to construct a non committing encryption scheme in the programmable random oracle model, but also state the danger that a commitment to the key in this construction might lead to a leakage of the plaintext. In our implementation, we make sure this is not the case, by letting the commitment be $\text{Commit}(k) = (\mathcal{H}(k), k)$. Note, that even with knowledge of $\mathcal{H}(k)$ it is not possible to distinguish $x_i \oplus \mathcal{H}(k||i)$ from $x_i \oplus \mathcal{H}(k'||i)$ for computationally bounded adversaries.

B.1 Extending Merkle Trees to Commitments

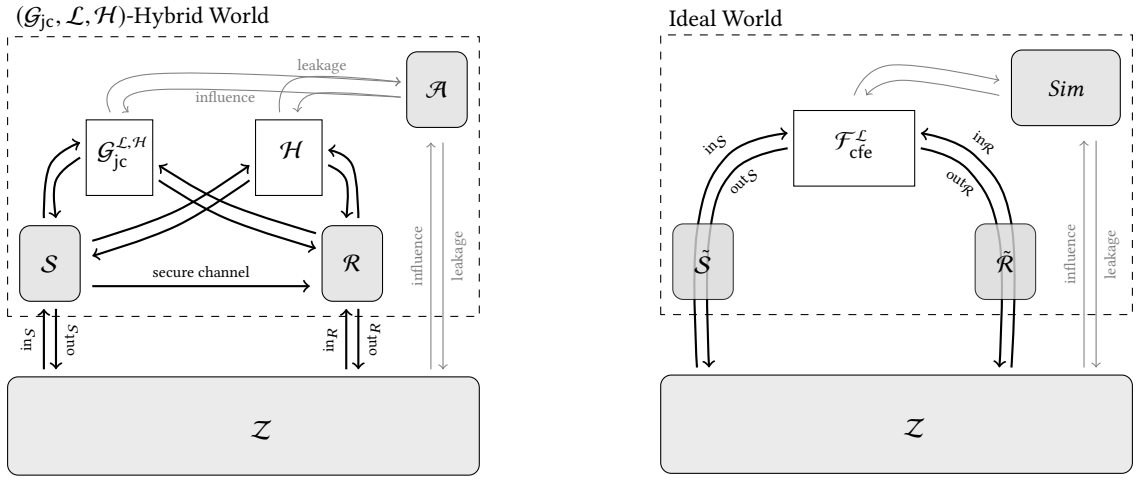
For the our protocol Π we need that both parties $\mathcal{S}$ and $\mathcal{R}$ jointly commit to the values $\boldsymbol{x}$ using a Merkle tree commitment towards the smart contract. The commitment on the values $\boldsymbol{x} = (x_1, \ldots, x_n)$ is generated using randomly sampled $d = (d_1, \ldots d_n)$, $d_i \in \{0,1\}^\kappa$ as follows:

$$\text{let } \boldsymbol{x'} = (x_1||d_1, \ldots, x_n||d_n))$$
$$\text{Commit}(\boldsymbol{x}) = (\mathit{root}(\mathit{Mtree}(\boldsymbol{x'})), d) = (c, d)$$
$$\text{Open}(c, \boldsymbol{x}, d) = \begin{cases} \boldsymbol{x}, \text{ if } \mathit{root}(\mathit{Mtree}(\boldsymbol{x'})) = c \\ 0, \text{ otherwise} \end{cases}$$

The scheme is hiding, as long as the randomness $r \in \{0,1\}^\kappa$ is chosen uniformly at random because then the commitments are indistinguishable for any ppt adversary. Note, that this commitment scheme does not satisfy the binding property if the random oracle is programmable, since the adversary has the power to program two values x, y to result in the same response $\mathcal{H}(x) = \mathcal{H}(y)$. In our protocol we do not need classical hiding when at least on of the parties $\mathcal{S}$ or $\mathcal{R}$ is honest (which is the only case in which our protocol satisfies fairness). In our case $\mathcal{S}$ generates the commitment (which he will do correctly if he is honest) and sends it together with the committed values to $\mathcal{R}$. The receiver $\mathcal{R}$ recomputes the commitment and additionally checks if any of the labels of the Merkle tree are programmed in $\mathcal{H}$. If he encounters any programmed value, an honest $\mathcal{R}$ will reject the root r and abort the protocol execution. This ensures that (as long as there is one honest party) the commitment is binding.

C GUC SECURITY PROOF

In order to prove Theorem 1 we need to show that the ideal world (the execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with dummy parties $\tilde{\mathcal{S}}$ and $\tilde{\mathcal{R}}$ and the ideal adversary *Sim*) is indistinguishable from the hybrid world. In our case the hybrid world is the execution of Π with parties $\mathcal{S}$, $\mathcal{R}$ and



(a) $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$**-hybrid world execution of** Π **with** $\mathcal{S}$, $\mathcal{R}$ **and** $\mathcal{A}$

(b) Execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ **with dummy parties** $\tilde{\mathcal{S}}$ **and** $\tilde{\mathcal{R}}$ **and** *Sim*

Figure 9: Setup of a Simulation with honest parties

an adversary $\mathcal{A}$ where each party interacts with the hybrid functionalities $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ and $\mathcal{H}$. Figure 9 depicts the setup of the security proof. The environment $\mathcal{Z}$ acts as a ppt algorithm, which distinguishes whether it interacts with the hybrid world execution of the protocol Π (c.f. Figure 9 (a)) or the with the ideal execution of the functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ (c.f. Figure 9 (b)). In the ideal world the parties $\tilde{\mathcal{R}}$ and $\tilde{\mathcal{S}}$ are so called dummy parties which only forward the in- and outputs of $\mathcal{Z}$ to the ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ whereas in the hybrid world the parties run the code of protocol Π. $\mathcal{Z}$ can use the leakage information from the adversary $\mathcal{A}$ (respectively the ideal adversary *Sim*) or actively influence the execution to distinguish the two worlds. Additionally, it selects the inputs for the two parties and learns its outputs. Lastly $\mathcal{Z}$ can corrupt any of the parties (using the adversary) to learn any internal values, and all messages send to and from the party. We will consider these cases in detail later.

To prevent $\mathcal{Z}$ from distinguishing the executions we need to construct a simulator, that outputs all messages such that it looks like the hybrid world execution to the Environment $\mathcal{Z}$. Specifically, *Sim* needs to ensure that the outputs of the parties are identical in the hybrid world and the simulation. Even with corrupted parties he needs to generate an indistinguishable transcript of the real world execution while ensuring that the global functionality $\mathcal{L}$ blocks or unblocks money in the same rounds. In order to formally prove Theorem 1 we need to consider four cases, the protocol execution with two honest parties, execution with a malicious sender $\mathcal{S}^*$, execution with a dishonest receiver $\mathcal{R}^*$ and the case where both parties are corrupt. All of the described termination cases (c.f. Section 4) provide seller and buyer fairness as defined in section 3.2. A thorough security proof can be found in the extended version of the paper [20]. It contains the construction of simulators for all possible corruption cases and for each a proof why the environment cannot distinguish the hybrid world execution from an ideal world execution. In the following we give a rough outline of the claims and proofs of the full version:

The Honest Case. Simulation of the protocol execution with an honest seller and honest receiver is special case, in which the dummy parties $\tilde{\mathcal{S}}$ and $\tilde{\mathcal{R}}$ will forward all messages from $\mathcal{Z}$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the ideal world. We proof the following claim:

Claim 1. *There exists a efficient algorithm* Sim^{honest} *such that for all ppt environments* $\mathcal{Z}$*, that* ***do not corrupt any party*** *it holds that the execution of* Π *in the* $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$*-hybrid world in presence of adversary* $\mathcal{A}$ *is computationally indistinguishable from the ideal world execution of* $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ *with the ideal adversary* Sim^{honest}*.*

The simulation in this case is straight forward since Sim^{honest} will be only required to generate a transcript of all messages of the execution of Π towards the adversary and thus $\mathcal{Z}$. This includes the simulation of the first protocol message, send from $\mathcal{S}$ to $\mathcal{R}$ and all following interactions with $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$ and $\mathcal{H}$. Note, that the communication between the honest $\tilde{\mathcal{S}}$ and $\mathcal{R}$ is private and $\mathcal{Z}$ cannot read the content of this message, but only see if a message was sent. We can show that this simulator can be build when the Merkle tree commitment is hiding.

Malicious Sender. Simulation with a corrupted sender is slightly more tricky than the simulation with two honest parties. The simulator in this case needs to simulate the transcript of Π and additionally all outputs of the corrupted sender towards $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and $\mathcal{Z}$.

Claim 2. *There exists a efficient algorithm* $Sim^{\mathcal{S}}$ *such that for all ppt environments* $\mathcal{Z}$*, that* ***only corrupt the sender*** *it holds that the execution of* Π *in the* $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$*-hybrid world in presence of adversary* $\mathcal{A}$ *is computationally indistinguishable from the ideal world execution of* $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ *with the ideal adversary* $Sim^{\mathcal{S}}$*.*

We proof this claim by showing that the hybrid and ideal world are indistinguishable to the environment $\mathcal{Z}$ if the commitment of the key is binding.

Malicious Receiver. Next, we will show security against malicious receivers (denoted as $\mathcal{R}^*$). The setup of the simulation is symmetrical to the one with malicious sender.

Claim 3. *There exists a efficient algorithm $Sim^{\mathcal{R}}$ such that for all ppt environments $\mathcal{Z}$, that* ***only corrupt the receiver*** *it holds that the execution of* Π *in the* $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$*-hybrid world in presence of adversary $\mathcal{A}$ is computationally indistinguishable from the ideal world execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with the ideal adversary $Sim^{\mathcal{R}}$.*

The main challenge in this proof is to provide the encoding z without knowledge of the witness x in the first round and to present key k in the third step such that the decryption of z yields x. Additionally, the key he provides during the reveal phase has to correctly open a commitment c, which $Sim^{\mathcal{R}}$ has to output in the first step. In order to construct this simulator, we will mainly utilize the programmability property of the global random oracle $\mathcal{H}$.

Two Malicious Parties. It remains to prove security for th last case, where $\mathcal{Z}$ corrupts both, the sender and the receiver. The case of malicious sender and receiver does not guarantee any fairness and might not even terminate. Recall, that we allow any malicious party to loose money if it does not follow to the protocol execution, aborts and declines the unfreezing request made by ledger (c.f. Section 3). A simple example of such a case is when $\mathcal{S}^*$ does not trigger the *finalize* message when interacting with $\mathcal{G}_{\text{jc}}^{\mathcal{L},\mathcal{H}}$. This will result in his money staying blocked forever. But even if the standalone case of two dishonest parties does not make much sense for the proposed applications, we still need to prove indistinguishability of the real and hybrid world, to guarantee security when composed with other protocols.

For more details on any of these proofs we refer the reader to the full version of the paper available on https://eprint.iacr.org/.

D FORMAL BACKGROUND OF STATE CHANNELS

Formally, a channel can always be described by its state γ, which is as an tuple with the following parameters:

$$\gamma = (id, \mathcal{R}, \mathcal{S}, p, q, nc, cash_{nc}, \tau)$$

If γ is a channel between the users $\mathcal{S}$ and $\mathcal{R}$ we call the users $\gamma.owners = \{\mathcal{S}, \mathcal{R}\}$. In this scenario, we use the identifier id to reference the channel instead of the contract. For every owner, the channel stores the values $\gamma.q$ and $\gamma.p$, which denote the amount of coins that are controlled by the channel. Some of these coins (denoted as $\gamma.cash_{nc}$) can be saved for a specific use, and will be handled by so called nanocontracts. The overall money that can be spend through the channel is $\gamma.cash = \gamma.q + \gamma.p + \gamma.cash_{nc}$. State Channels can contain nanocontracts, which are represented in the state as $\gamma.nc$. This value can either be empty $\gamma.nc = \perp$, which means that the channel currently does not contain a nanocontract, or $\gamma.nc = nc$, which means the contract nc is run as a nanocontract. A nanocontract is defined by its code and internal storage $state_{init}$, must have an initiating function $nc.init(state_{init})$ and outputs a distribution of coins $(out_{\mathcal{R}}, out_{\mathcal{S}})$. For simplicity we only consider one nanocontract nc per state channel but it could without loss of generality also contain multiple internal contracts. Before sending this message, both parties should verify whether it is possible to execute C without involvement of the other party within time τ and that it has the expected function. If this check is not performed, their coins reserved for the execution of id might get lost.

Whenever the parties want to update the money distribution or other internal values in a channel γ they agree on a new channel γ^* with the updated parameters. We say that γ^* is a successor of γ when the following checks hold:

- $\gamma^*.id = \gamma.id$,
- $\gamma^*.owners = \gamma.owners$,
- $\gamma^*.cash = \gamma.cash$ and
- $\gamma^*.nc \in \{nc, \perp\}$.

In short we write $\gamma \mapsto \gamma^*$.

To create the channel, both $\mathcal{S}$ and $\mathcal{R}$ have to agree on a channel with state γ. Then both send a message containing γ to the ideal channel contract functionality $\mathcal{F}_{Chan}^{\mathcal{L}}(C)$[8]. If some party aborts during this step or the funds are insufficient, the deposited money can be refunded after the timeout through $\mathcal{F}_{Chan}^{\mathcal{L}}(C)$. Otherwise, a channel is created with state γ. It can now be updated an arbitrary number of times as long as both parties agree. For this $\mathcal{S}$ and $\mathcal{R}$ update the state to γ^* with $\gamma \mapsto \gamma^*$.

If $\mathcal{S}$ and $\mathcal{R}$ want to run a nanocontract nc in the channel, they use its state parameters $\gamma.cash_{nc}, \gamma.nc, \gamma.\tau$. Both parties should verify beforehand whether it is possible to execute nc without involvement of the other party within time $\gamma.\tau$ and that it has the expected function. If this check is not performed, the coins $\gamma.cash_{nc}$ reserved for the execution of $\gamma.nc$ might get lost.

As long as both parties agree, they can update the channel whenever they want to run nanocontract nc. If they also agree on the outcome of the nanocontract execution, they update the channel state again, such that it contains the correct redistribution of coins and the nanocontract is deleted. Should they not agree on the outcome of the nanocontract, either party can enforce the execution of the internal nanocontract and terminate the channel. The execution of the nanocontract works analogue to the deployment and execution of a contract in the contract hybrid world.

[8] For more information how this ideal functionality is constructed refer to [21]