Etherless Ethereum Tokens: Simulating Native Tokens in Ethereum

John Andrews[1], Michele Ciampi[2(✉)], and Vassilis Zikas[3]

[1] Sunday Group, Las Vegas, USA
jandrews@sundaygroupinc.com
[2] The University of Edinburgh, Edinburgh, UK
michele.ciampi@ed.ac.uk
[3] Purdue University, West Lafayette, USA
vzikas@cs.purdue.edu

Abstract. Standardized Ethereum tokens, e.g., ERC-20 tokens, have become the norm in fundraising (through ICOs) and kicking off blockchain-based DeFi applications. However, they require the user's wallet to hold both tokens and ether to pay the gas fee for making a transaction. This makes for a cumbersome user experience, and complicates, from the user perspective, the process of transitioning to a different smart-contract enabled blockchain, or to a newly launched blockchain. We formalize, instantiate, and analyze in a composable manner a system that we call *Etherless Ethereum Tokens* (in short, EETs), which allows the token users to transact in a closed-economy manner, i.e., having only tokens on their wallet and paying any transaction fees in tokens rather than Ether/Gas. In the process, we devise a methodology for capturing Ethereum token-contracts in the Universal Composability (UC) framework, which can be of independent interest. Our system can be seen as a targeted instance of the more general paradigm put forth—without a formal study—by the Ethereum Gas Station Network (GSN). We have implemented and benchmarked our system and compared it to GSN. In addition to being the first system with a rigorous security analysis, we demonstrate that EETs are far easier to deploy and less gas intensive than the GSN.

1 Introduction

As applications of smart contracts, e.g., Decentralized Finance (DeFi) and Non-Fungible Tokens (NFTs), become mainstream, there is a need to make them as independent from the Ethereum chain as possible. This is particularly relevant for Ethereum tokens (e.g., ERC-20 tokens [20]). Indeed, for a token-holder to exchange or transfer such tokens, they need to also hold Ether for fuelling the Ethereum transaction. This is counter-intuitive and counter-productive: on the one hand, token creators need to provide a wallet which supports both their token and Ethereum, making it more challenging to transition to their own blockchain or switch token platforms while offering a smooth user experience. On the other hand, users need to make sure that they hold not only the token but also Ether,

S. Dolev et al. (Eds.): CSCML 2022, LNCS 13301, pp. 388–407, 2022.
https://doi.org/10.1007/978-3-031-07689-3_29

which makes it more challenging to expand this technology to less tech-savvy audiences, thereby hindering wider societal adoption.

The easiest way to conceptualize the relevant bottleneck is through considering the life cycle of an ETH-based initial coin offering (ICO): in a first stage, the token creator solicits investment (typically in different cryptocurrencies), under the promise of a certain (prearranged) amount of tokens once the token launches.[1] In a second phase, the token creator initializes the promised new token by launching a token smart contract (e.g. an ERC 20 token) on the Ethereum chain. The token creator then would then have the investors create and provide an Ethereum address where the promised tokens can be transferred. This can be done by means of a wallet that offers generic support for Ethereum tokens.

Often, however, ICO-funded applications launch tokens which have the ultimate goal of eventually being disconnected from the main Ethereum blockchain, and/or which aim to create an ecosystem independent of Ethereum. In such cases, the token creator would typically also offer its users a token-specific wallet application. However, in order for anyone to use this application to transfer his tokens, the token-specific wallet needs to also support Ether as a currency. This leads to confusion for less tech-savvy investors, and makes the user experience of migrating the token to a different smart contract platform—e.g. a different smart-contract-enabled blockchain or a blockchain developed by the token creator—less intuitive. We note that such migration is becoming more relevant as more smart-contract-enabled blockchains are released, and as the gas price for Ethereum smart contracts rises to a point where its use makes the corresponding tokens less attractive.

In this work, we start by proposing a design methodology and formal treatment of Ethereum tokens which allow their creator to provide the option to its users of making transfers without the need to hold Ether in their wallet, a mechanism which we term *Etherless Ethereum Tokens (in short, EETs)*. The high-level idea is simple: allow the token creator to take on the cost (i.e., gas) for the token transaction, and have the token contract perform an on-the-fly exchange of token-to-ether at a pre-agreed rate, giving the user the experience of a native token. As one might expect, properly specifying, implementing, and proving such a protocol secure is a challenging task; in particular, it requires a model for token-enabled ledgers, which we provide and believe it will be helpful for all the future systems that might rely on token contracts in a composable manner. We remark that, as a concept, etherless transactions have been frequently discussed within the Ethereum community for several years, often under the term *meta transactions* [1,2,12,18]. However, to our knowledge, our work is the first to provide a formal treatment and security analysis of the concept.

The need for arguing the security of blockchain systems formally and in a composable manner is motivated by the fact that a blockchain does not live

[1] There are a number of legal issues regarding ICO's—in particular, how to hold the token creator to his promise and how to avoid scamming attacks—and there are technological advances that allow us to circumvent them; these topics are outside the scope of this paper.

in isolation, and that many applications might run on top of it. Hence, it is fundamental to argue the security of new blockchain applications in a setting where multiple protocols are running in concurrency. Indeed, many recent works have focused specifically on this task, proposing formal security models and proving that existing protocols (like Bitcoin [5,10]) satisfy some important and well-formalized security properties. On the same spirit, many other works have used the same rigorous approach to argue and define the security of other blockchain systems and applications. Just a few other examples are proof-of-stake blockchains, private blockchains, private smart contracts and the lighting network. [4,13–15].

At a less technical level, we believe that in addition to offering a more intuitive, closed-economy user experience, EET also provides assurance to the original ICO investors that the token creator indeed expects value on the token, as he is willing to make marginal exchanges. Indeed, in the system we design anyone (in particular the token creator) could pay the fee (in Ether) on the behalf on another party. Throughout we will generically refer to such an entity as *intermediary*. We note in passing that despite being explicitly implemented on the Ethereum blockchain, our design is generic and can be ported to any smart-contract-enabled blockchain platform, and thus can enable transferring the tokens from one blockchain to another.

We have implemented our EET design, and we demonstrate how it outperforms existing generic systems that enable etherless transactions, such as the Gas Station Network (GSN) [1], both in terms of simplicity of deployment and in terms of gas usage. We also compare such a deployment with how a native token could perform on Ethereum and demonstrate that the overhead makes the flexibility offered by black-box usage of smart-contract-based tokens a reasonable compromise for the moderate increase in the required gas it incurs over what a native token would require.

2 Our Contributions and Related Work

Our contribution is threefold: (1) A universally composable (UC) [6] treatment of ledgers supporting a broad class of smart contracts, which includes token contracts (e.g. ERC 20). (2) A design and UC security analysis of EETs. (3) An implementation of our EET, benchmarks, and comparison with alternative approaches. In the following, we expand on the key components of the above contributions, and put our results in perspective with existing literature and systems.

2.1 Smart-Contract-Enabled Transaction Ledgers

The first analyses of blockchain protocols showed that they satisfy certain desirable properties, such as common-prefix (also referred to as safety or consistency), chain-growth (also referred to as liveness), chain quality, etc. [4,5,9–11,16,17]. Badertscher *et al.* [5] put forth the first universally composable treatment of the

Bitcoin backbone (i.e. consensus layer) by introducing a UC functionality, called $\mathcal{F}_{\text{LEDGER}}$, which captures the interface that Bitcoin offers to external applications, rather than the way in which this interface is implemented. At a very high level, $\mathcal{F}_{\text{LEDGER}}$ takes as input transactions which are validated by means of a validation predicate Validate. All valid transactions are then stored into a data structure denoted as *state*. The adversary has full control over the order in which transactions appear in `state`, and can define (in a limited way) the portion of the state that each party can access. However, once something is added to the state, it cannot be removed (not even by the adversary). We note that the advantage of proving security in UC is that it enables use of the ledger as an ideal primitive, and ensures that replacing this ideal ledger primitive by its implementation—the corresponding blockchain—does not compromise the security of primitives that make ideal calls to the ledger; nor does it affect the security of systems and protocols that run alongside the ledger. This property is often referred to as universal composability, and it allows for a constructive approach to cryptographic/security protocols, analogous to how programming uses libraries with fixed APIs without worrying about their implementation. Following that work, a number of papers on the design and analysis of blockchains have adopted UC as the model to prove their security and have devised systems implementing variants of the above ledger [4,14]. UC [5] has also been leveraged to describe how $\mathcal{F}_{\text{LEDGER}}$ may be used together with a digital signature scheme to derive a *transaction ledger*, abstracting the cryptocurrency aspects of Bitcoin in addition to its backbone guarantees.[2]

This was done by relying on digital signatures where, to ensure composability, the ideal adversary is allowed to choose the signing and verification keys.

The Transaction Ledger. In this paper we consider a simpler, more UC-friendly approach that abstracts away the public-key infrastructure (PKI), analogous to how the UC signatures functionality [7] would. In a nutshell, instead of having Validate rely on a specific signature scheme, we define a new transaction ledger $\mathcal{F}_{\text{T-LEDGER}}$ that internally runs $\mathcal{F}_{\text{LEDGER}}$ and also emulates existentially unforgeable signatures, similar to [7]. $\mathcal{F}_{\text{T-LEDGER}}$ accepts transactions with the format $\mathtt{tx} := (v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{fee})$ where v represents the number of coins involved in the transaction, fee is the fee that the issuer of the transaction is willing to pay, and addr_i and addr_j represent the wallet addresses of the sender and the receiver respectively. Upon receiving a transaction, $\mathcal{F}_{\text{T-LEDGER}}$ checks the state of $\mathcal{F}_{\text{LEDGER}}$ to ensure that the wallet address addr_i has at least $v + \mathsf{fee}$ coins and that the fee is sufficient, i.e. that $\mathsf{fee} \geq f(\mathtt{tx})$, where f is function specified in the description of $\mathcal{F}_{\text{T-LEDGER}}$ that determines the fee that needs to be payed for the input transaction. We note that it is straightforward to adapt the analysis of the transaction ledger [5]—using a specific existentially-unforgeable signatures scheme—to prove security of our ledger for a standard Bitcoin-style blockchain protocol, such as

[2] Unlike transaction ledgers, the bare $\mathcal{F}_{\text{LEDGER}}$ captures the consensus layer, and does not interpret its contents as transactions which need to be verified with respect to whether or not they are spending some already spent coin.

Bitcoin or the proof-of-work-based version of Ethereum. Nonetheless, as we shall see, this makes it more intuitive to add cryptocurrency-relevant features to the ledger–such as etherless tokens.

Adding Smart Contracts. The functionality $\mathcal{F}_{\text{T-Ledger}}$ is sufficient to capture the base functionality of cryptocurrencies, but it does not support smart contracts. To achieve that, in this work we define an augmented functionality, which we denote $\mathcal{F}_{\text{TSC-Ledger}}$. This represents our first contribution. $\mathcal{F}_{\text{TSC-Ledger}}$ internally manages $\mathcal{F}_{\text{T-Ledger}}$ and a functionality $\mathcal{F}_{\text{SC}}$ that abstracts a smart contract: $\mathcal{F}_{\text{SC}}$ maintains its own state `cstate`—corresponding to the state of a (virtual) machine VM[3]—and is parametrized by a function f_{CFee}, that takes as input the query to the contract (which contains also the fee that the caller is willing to pay to run the contract), and checks whether or not the fee is enough for the VM to process the input and update its state.

The construction of $\mathcal{F}_{\text{TSC-Ledger}}$ from its components is illustrated in Fig. 1. $\mathcal{F}_{\text{TSC-Ledger}}$ accepts either standard transactions in the native currency `E` (that are forwarded to $\mathcal{F}_{\text{T-Ledger}}$) or inputs/transactions that are intended as queries to the contract $\mathcal{F}_{\text{SC}}$. Upon receiving such a query for the smart contract, $\mathcal{F}_{\text{TSC-Ledger}}$ forwards the query to $\mathcal{F}_{\text{SC}}$, which checks if the fee specified in the query is sufficient to update its state, and if so it updates `cstate` by running the VM on input the given transaction and the state of $\mathcal{F}_{\text{T-Ledger}}$ (which is handed to $\mathcal{F}_{\text{SC}}$ by $\mathcal{F}_{\text{TSC-Ledger}}$)[4], and returns the updated state (including the received input) to $\mathcal{F}_{\text{TSC-Ledger}}$. $\mathcal{F}_{\text{TSC-Ledger}}$ then pushes the query and the updated state `cstate` to the state of $\mathcal{F}_{\text{T-Ledger}}$ (by submitting it as a transaction). Consistently with the Ethereum smart contract mechanism, $\mathcal{F}_{\text{SC}}$ charges the contract caller only for the fee that is required to update its state, even if the contract's caller specified a higher fee. Moreover, if a contract caller did not specify a fee high enough to conclude an update on the contract's state, the fee will be deducted from the caller account, and the input used to query the contract will appear in the state of $\mathcal{F}_{\text{T-Ledger}}$, though no change to the contract's state will be committed.

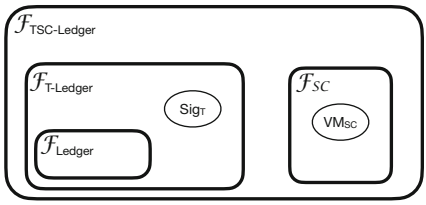

Fig. 1. The smart-contract-enabled transaction ledger functionality $\mathcal{F}_{\text{TSC-Ledger}}$

Tokens as Smart Contracts. Given the above smart-contract-enabled ledger, it is straightforward to capture a smart contract for creating a standard (e.g. ERC 20 [20]) Ethereum token by instantiating $\mathcal{F}_{\text{TSC-Ledger}}$ with contract functionality that stores and updates the state (balances for different addresses)

[3] We do not specify a model of computation for describing the VM; one can use any such model, e.g. Turing machines, RAMs, etc.

[4] Note that $\mathcal{F}_{\text{TSC-Ledger}}$ also keeps track of the history of the state of $\mathcal{F}_{\text{T-Ledger}}$.

of such a token. Note that this results in a token-enabled transaction ledger $\mathcal{F}_{\text{LEDGER}}^{\mathsf{Token}}$ which allows parties both to issue transactions in the native coin $\mathtt{E}$, and to exchange tokens $\mathtt{T}$.

In more detail, $\mathcal{F}_{\text{LEDGER}}^{\mathsf{Token}}$ instantiates $\mathcal{F}_{\text{TSC-LEDGER}}$ with a token-contract $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ which works as follows: $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ collects all token transactions, and upon receiving a read-request returns only the *valid* token transactions. Similarly to the way the ledger $\mathcal{F}_{\text{T-LEDGER}}$ deals with native transactions, a token transaction consists of the components $(v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}})$, where v is the number of tokens involved in the transaction, and addr_i and addr_j represent the token wallet addresses of the sender and the receiver respectively. Furthermore, $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ internally emulates an existentially-unforgeable signature scheme related to the token which is independent of the one that is used in $\mathcal{F}_{\text{T-LEDGER}}$.[5]

We observe that there is no fee appearing in the description of the token transaction. The reason is that the fee will be part of the query to the contract, and it is expressed in the native currency $\mathtt{E}$. Indeed, the issuer of the token transaction, in order to query the contract $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$, needs to possess coins of type $\mathtt{E}$.

The EET Functionality. As discussed in the introduction, the above contract implementation of tokens—which has become a standard for Ethereum—has the undesireable property that a party who wants to send tokens requires coins of type $\mathtt{E}$ to do so, coins which they might not have. In this work, we introduce EETs to allow the token creator to offer, as a service, to take on the cost of the token transaction, in exchange for tokens at a pre-agreed $\mathtt{E}$-to-$\mathtt{T}$ rate. This is captured by tweaking the token-enabled ledger $\mathcal{F}_{\text{LEDGER}}^{\mathsf{Token}}$ toward an EET-enabled ledger, denoted as $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, which supports an additional input called SUBMIT-DELEGATION. Upon receiveing SUBMIT-DELEGATION, $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ allows the user to issue a token transaction which pays a fee, in $\mathtt{T}$, to a special party, called *intermediary* (that we denote with M), in exchange for the intermediary submitting the token transaction to $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ and paying the $\mathtt{E}$ needed for the token contract to process the transaction. In our system anyone can be an intermediary. More precisely, there might be multiple intermediaries that are willing to pay the Ether fee for a transaction, but each of them will do that at a potentially different exchange rate. This means that any user that wants to delegate the payment of the fee can look at what rates are available and decide accordingly with what intermediary to interact with. The agreement on the rate is made completely off-chain, and for sake of simplicity in the paper we assume that there is only one intermediary and that the rate has been already pre-agreed between the parties.

2.2 EET Construction and Analysis

To realize $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ we rely only on $\mathcal{F}_{\text{T-LEDGER}}$ and signatures. In particular, any party that wants to issue a token transaction and has enough coins of type $\mathtt{E}$ to

[5] Note that we cannot generically use the same signature emulator procedure of $\mathcal{F}_{\text{T-LEDGER}}$, as a token address is typically overloaded to also be an Ethereum address.

cover for the fee can issue a transaction $\mathtt{tx} = (0, \mathsf{addr}_i, 0^\lambda, (\mathsf{aux}, \sigma), \mathsf{fee})$, where $\mathsf{aux} = (v, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}})$ and σ is a signature of aux that verifies under $\mathsf{addr}_i^{\mathsf{T}}$.[6]

In a nutshell, $\mathtt{tx}$ is a standard transaction for $\mathcal{F}_{\text{T-Ledger}}$ that contains in its payload the information related to the token transaction properly signed by the sender. By definition, if the fee fee is high enough, then $\mathtt{tx}$ will become part of $\mathcal{F}_{\text{T-Ledger}}$'s state. Let $\mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}$ be the token wallet address of M. To delegate a transaction, the sender P_i creates a special token transaction $\mathsf{aux} = ([v, \mathsf{del\text{-}fee}], \mathsf{addr}_i^{\mathsf{T}}, [\mathsf{addr}_j^{\mathsf{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}])$ (where $\mathsf{del\text{-}fee}$ is a fee expressed in T that parametrizes $\mathcal{F}_{\text{Ledger}}^{\mathsf{EET}}$) and signs it to obtain σ. aux is the atomic representation of two token transactions: the first moves v tokens from $\mathsf{addr}_i^{\mathsf{T}}$ to $\mathsf{addr}_j^{\mathsf{T}}$, and the second moves $\mathsf{del\text{-}fee}$ from $\mathsf{addr}_i^{\mathsf{T}}$ to $\mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}$. M, upon receiving (aux, σ) submits a transaction to $\mathcal{F}_{\text{T-Ledger}}$ that contains (aux, σ) in its payload. If a party wants to obtain only the valid token transaction, they need to filter out the payload of the transactions stored in $\mathcal{F}_{\text{T-Ledger}}$'s state, and output only the valid transactions. Similarly to what we have described above, a token transaction $(v, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}})$ is valid if the sum of tokens with receiver address $\mathsf{addr}_i^{\mathsf{T}}$ minus the sum of tokens in the state with sender address addr_i (including the fees) is greater than or equal to v.

2.3 Implementation, Benchmarks, and Comparisons

The Gas Station Network (GSN) is a relatively recent development in the Ethereum community that shares some of our goals, but a broader scope. In particular, the GSN aims to create a decentralized, trustless network of *relay servers* which can pick up the transaction fees for any GSN-enabled contract. The GSN is built around a $\mathsf{RelayHub}$ smart contract that:

1. Records available relay servers and their service fees,
2. Keeps ether deposits from GSN-enabled contracts for repayment of relay servers,
3. Facilitates the interaction between relays and GSN-enabled contracts, and punishes any detected bad actors.

This is in contrast to our mechanism, in which there is no separate smart contract to manage the delegation of transactions. Additionally, each GSN-enabled contract must interact with a separate *paymaster* contract, which is responsible for performing any action needed to extract or verify payment from users. Paymaster contracts may be written generically and shared between multiple contracts, or purpose-written for particular contracts.

The outward functionality of the GSN is similar to our mechanism: a gasless user submits a transaction to an intermediary relay server instead of directly to the blockchain, and the relay submits the transaction on the user's behalf, receiving an ether repayment from the target contract. The target contract, in turn, is allowed to extract any payment it wishes from the user, e.g. tokens.

[6] In the protocol, the addresses become verification keys for a signature scheme.

The primary difference is in the complexity of implementation and development; where the GSN aims to be fully generic and decentralized, and admits a great deal of complexity in service of that aim, we have endeavored to keep our efforts very self-contained in order to ease implementation, simplify formal analysis, and keep operational costs manageable.

As is common in designs that aim for maximally generic functionality, the GSN pays for its genericity with increased complexity. This complexity manifests both in development effort—anecdotally, we found setting up a testing environment for a GSN-enabled contract to be significantly more cumbersome than for other contracts—and in gas consumption. Our experiments indicate a 4-5x overhead in gas consumption when using the GSN as opposed to using our EET contract. (Note that gas is pretty much the only relevant measurable unit of comparison. Other metrics—e.g. running time, settlement time, etc.—are either very difficult to test in a controlled way, are irrelevant for a contract which aims only to facilitate token exchange, or are negligible compared to other confounding factors.) We note in passing that, to our knowledge, there is no formal security analysis of the GSN, making our work the first rigorous treatment of the etherless token paradigm.

Contract-Based vs Native Tokens. Recently, the blockchain/cryptocurrency community has been entertaining the idea of making tokens native to the cryptocurrency chain. In parallel and independent work [8] the authors propose a solution that allows users posting a token transaction along with a token-to-native exchange rate he is willing to pay; at the same time anyone could issue transactions aimed at covering the fee of such token transactions in exchange of tokens coins. Then any miner/minter that can match such transactions (if any valid match exists) can create a block that contains both transactions, in which the fee for the token transaction has been payed by a third party. A similar solution has been proposed in [19]. Such approach yields an advantage in terms of fees needed for the transaction, but it does come at a cost: (1) The block miner are in an advantageous position and can always front-run other users proposing their own transactions to cover for the fees of token transaction; (2) The token functionality is limited to what is hardwired on the token chain, and is therefore far less flexible than a smart-contract-based solution. For example, it is unclear if or how such a solution would allow the use of amortization/batching to save on bulk transactions. (3) If one adopts the natural "pay-per-use" principle for fees—i.e. you pay more for a more complex transaction—as Ethereum does, then adding this functionality would increase the cost of all transactions, including those that only involve the native cryptocurrency. Although this increase is expected to be minimal, it is unclear how the implicit auction for the submitted token transaction created by such a mechanism would affect fees.

In the full version [3], we have included an attempt to estimate the overhead this might incur in a hypothetical implementation on Ethereum, and compare it with using a smart contract. We note that in the absence of a (platform or blockchain supporting an) actual implementation of native tokens, the relevant

experiments are somewhat artificial and speculative. Thus, we do not consider these experiments an important part of our contributions (and we defer them to the appendix). Nonetheless, we do believe they give an interesting perspective to the discussion on native tokens, and a pointer for experiments once such a functionality is implemented on a mainstream blockchain. Finally, we stress that our solution works on all blockchains that support token contracts (i.e., no need for turing completeness) like Cardano, Dfinity and Ethereum, whereas the solution proposed in [8] would require to fork an existing blockchain to accommodate for a new validation rule.

3 Preliminaries and Model

We denote a randomized assignment is denoted with $a \xleftarrow{\$} A$, where A is a randomized algorithm. We use existentially unforgeable and non-repudiable signatures [7]. A signature scheme is a triple of PPT algorithms $\Sigma = (\mathtt{Gen}, \mathtt{Sign}, \mathtt{Ver})$ where $(s, v) \xleftarrow{\$} \mathtt{Kgen}(1^\lambda)$ generates a secret-key/public key pair, $\sigma \xleftarrow{\$} \mathtt{Sign}(s, m)$ generates a signature and $\mathtt{Ver}(v, m, \sigma)$ verifies that σ is a valid signature. We refer to the full version for the formal definition. We provide our protocols and security proofs in Canetti's universal composition (UC) framework [6]. We assume that the reader is familiar with simulation-based security and has basic knowledge of the UC framework. For more detail, we refer to the full version [3]. We now elaborate on the main hybrid functionality used in our paper.

The functionality $\mathcal{F}_{\mathbf{ledger}}$. The main functionality (in fact, a global setup) we rely on is a cryptographic distributed transaction ledger. We use the (backbone) ledgers proposed in the recent literature [4,5] in order to describe a transaction ledger and its properties. As proved in [4,5], such a ledger is implemented by known permissionless blockchains based on either proof-of-work (PoW), e.g. Bitcoin, or poof-of-stake (PoS), e.g. Ouroboros Genesis. The ledger stores an immutable sequence of blocks called *state*—each block containing several messages typically referred to as *transactions* and denoted by `tx`—which is accessible from the parties under some restrictions discussed below. It enforces the following basic properties that are inspired by [10,16]:

- *Ledger growth.* The size of the ledger's state should grow—new blocks should be added—as the rounds advance.
- *Chain quality.* It is guaranteed that a percentage of honest blocks are created in a sufficiently long sequence of blocks.
- *Transaction liveness.* Old enough (valid) transactions are included in the next block added to the ledger state.

We next give a brief overview of the ledger functionality $\mathcal{F}_{\text{LEDGER}}$ proposed in [4,5], focusing on the properties of $\mathcal{F}_{\text{LEDGER}}$ that are relevant for the understanding our results. Along the way we also introduce some useful notation and terminology. We refer the reader interested in the low-level details of the ledger

functionality and its UC implementation to the full version [3] and [4,5]. We note that with minor differences related to the nature of the resource used to implement the ledger, PoW vs PoS, the ledgers proposed in these works are identical.

The functionality $\mathcal{F}_{\text{LEDGER}}$ is parametrized by three main functions Validate, ExtendPolicy and Blockify. At a high level, anyone (honest miner or the adversary) may submit a transaction to $\mathcal{F}_{\text{LEDGER}}$. The transaction is validated by means of a filtering predicate Validate, and if it is found to be valid it is added to a *buffer* that we denote `buffer`. Taking a peak at the actual implementation of the ledger, this buffer contains transactions that, although validated, are either not yet inserted into a valid block, or are in a block which is not yet deep enough in the blockchain to be considered immutable for an adversary. The adversary $\mathcal{A}$ is informed that the transaction was received and is given its contents. Periodically, $\mathcal{F}_{\text{LEDGER}}$ does the following: (1) fetches some of the transactions in the buffer under the influence of the adversary (more on this will follow), (2) modifies them by means of a procedure Blockify, (3) creates a block including the output of Blockify, and (4) adds this block to its permanent state, denoted as `state`. `state` is a data structure that includes the sequences of blocks that the adversary can no longer change. (In [10,16] this corresponds to the *common prefix*.) Any miner or the adversary is allowed to request a read of the contents of the state and, every honest miner will eventually receive `state` as its output.[7] To enforce transaction liveness and chain-quality, $\mathcal{F}_{\text{LEDGER}}$ relies on the function ExtendPolicy. At a high level, ExtendPolicy makes sure that the adversary cannot create too many blocks with arbitrary (but valid) contents (chain quality) and that if a transaction is old enough, and still valid with respect to the actual state, then it is included into the state. In more detail, ExtendPolicy takes the current contents of the buffer, along with the adversary's recommendation `NxtBC`, and the block-insertion times vector $\tau_{\texttt{state}}$. The latter is a vector listing the times when each block was inserted into the state. The output of ExtendPolicy is a vector including the blocks to be appended to the state during the next state-extend time-slot. Each of these blocks is then given as input to Blockify. We conclude the discussion by providing a high-level description of the main input command of $\mathcal{F}_{\text{LEDGER}}$ used in our protocols/definitions, and refer to Sect. 3 for a formal description of the functionality.

– The input $(\text{READ}, \text{sid})$ is used to request the content of the ledger's `state`. Concretely, upon receiving $(\text{READ}, \text{sid})$ from some party (or the adversary on behalf of a corrupted party), the ledger returns (a prefix of) `state` to the caller.

[7] As observed in [5], it is not possible to guarantee with existing constructions that at any given point in time all honest parties see exactly the same state (blockchain) length, so each party may have a different view of the state which is defined by the adversary. However, the adversary can restrict the view of the honest parties only by a bounded number of blocks. The parameter that defines such a bound is called `windowSize`.

– The input (SUBMIT, sid, tx) is used to request that a transaction tx be added to the buffer. That is, upon receiving a (SUBMIT, sid, tx) message from any party (or the adversary), the ledger adds the transaction tx to the buffer buffer. If the validation predicate Validate, on input state, buffer, tx outputs 1, then tx will be included in state.[8] The time required for the transaction to be part of state and visible to all honest parties who query $\mathcal{F}_{\text{LEDGER}}$ depends on the transaction liveness parameter defined in ExtendPolicy.

4 The Cryptocurrency-Ledger Functionality $\mathcal{F}_{\text{T-LEDGER}}$

The ledger $\mathcal{F}_{\text{LEDGER}}$ does not itself realize a cryptocurrency (unless if couple with a signature scheme as described in [5]). To this direction we define and instantiate a cryptocurrency (transaction) ledger $\mathcal{F}_{\text{T-LEDGER}}$ hosting a coin denoted by E. As discussed in the introduction, in contrast to the transaction ledger from [5] our construction does not assume an external signature functionality. This makes it more useful for defining smart contracts (see Sect. 5).

The validation predicate of $\mathcal{F}_{\text{LEDGER}}$, in this case, is defined to always output 1, and it is $\mathcal{F}_{\text{T-LEDGER}}$'s responsibility to make sure that only valid transactions are submitted to $\mathcal{F}_{\text{LEDGER}}$. $\mathcal{F}_{\text{T-LEDGER}}$ also generates and manages the *wallets* of the parties. A transaction supported by $\mathcal{F}_{\text{T-LEDGER}}$ consists of five main components $(v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{aux}, \mathsf{fee})$, where v represents the amount of coins of type E, addr_i is the sender's wallet address, addr_j is the receiver's wallet address, aux is a payload, and fee represents the fee. At a high level, a transaction is valid if the fee fee is high enough and if the amount of coins stored in the wallet with address addr_i is at least $v + \mathsf{fee}$. How high the fee should be in order for the transaction to be considered is specified by a function f that is part of the description of $\mathcal{F}_{\text{T-LEDGER}}$. f takes as input the transaction tx and computes the required fee. In the case where the output of f is greater than fee, the transaction is immediately discarded. Otherwise, $\mathcal{F}_{\text{T-LEDGER}}$ replaces fee with the output of the function and submits it. This captures the fact that $\mathcal{F}_{\text{T-LEDGER}}$ charges the issuer of the transaction only for the cost of processing the transaction, even if the transaction specifies a higher fee. In more detail, each party has an associated wallet address, and different parties have different wallet addresses. $\mathcal{F}_{\text{T-LEDGER}}$ manages a table $\mathcal{T}$ that, for each party P_i, stores P_i's wallet address addr_i. We initialize $\mathcal{F}_{\text{T-LEDGER}}$ with a party P_0 which initially holds all the coins (e.g., V coins) of type E[9]. To do so, $\mathcal{F}_{\text{T-LEDGER}}$ generates an address addr_0 and sends (SUBMIT, sid, tx) to the wrapped $\mathcal{F}_{\text{LEDGER}}$ with $\mathtt{tx} := (V, 0^\lambda, \mathsf{addr}_0, \perp, 0)$, where V is the initial amount of coins held by P_i and 0^λ is a special address used only for the initialization. Upon receiving a registration request from a party P_i, $\mathcal{F}_{\text{T-LEDGER}}$ creates a new wallet address addr_i and adds (addr_i, P_i) to the table

[8] We have the guarantee that any transaction (either generated by a malicious or honest party) that manages to go in buffer will eventually be included in state.

[9] It is easy to intialize the functionality with an arbitrary number of parties that hold an initial amount of coin. To simplify the description on the functionality, we decided to use only one party in this phase.

$\mathcal{T}$. $\mathcal{F}_{\text{T-LEDGER}}$, upon receiving $(\text{SUBMIT}, \text{sid}, \texttt{tx})$ from a party P_i, performs the following steps.

- Parse $\texttt{tx}$ as $(v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{aux}, \mathsf{fee})$ and continue if and only if $(P_i, \mathsf{addr}_i) \in \mathcal{T}$ and $\mathsf{fee} \geq f(\texttt{tx})$.
- Get $\texttt{state}$ and $\texttt{buffer}$ of $\mathcal{F}_{\text{LEDGER}}$ and check that the balance of transactions to/from the wallet address addr_i is at least $v' \geq v + f(\texttt{tx})$ coins. That is, the sum of coins with receiver address addr_i minus the sum of coins in the state with sender address addr_i (including the fees) is greater than or equal to $v + f(\texttt{tx})$. If this is not the case, deem the transaction invalid; otherwise, submit $\texttt{tx}$ to $\mathcal{F}_{\text{LEDGER}}$ with the fee $f(\texttt{tx})$.

$\mathcal{F}_{\text{T-LEDGER}}$ is also parametrized with the identifier of an ideal functionality $\mathcal{F}_{\mathsf{trap}}$. Whenever $\mathcal{F}_{\text{T-LEDGER}}$ receives the command $(\text{SUBMIT-TRAPDOOR}, \text{sid}, \texttt{tx}, P_i)$ from $\mathcal{F}_{\mathsf{trap}}$, it forwards the transaction $\texttt{tx}$ on behalf of P_i to $\mathcal{F}_{\text{LEDGER}}$ without checking anything about $\texttt{tx}$ in terms of balances and fees. This simple mechanism allows $\mathcal{F}_{\text{T-LEDGER}}$ to interact with other ideal functionalities when required. This becomes particularly helpful when we want to enhance the behavior of $\mathcal{F}_{\text{T-LEDGER}}$ with smart contracts, and in the next section we show how to do that. For all the other input commands, $\mathcal{F}_{\text{T-LEDGER}}$ just acts as a proxy between $\mathcal{F}_{\text{LEDGER}}$ and its external interface. To conclude the description of $\mathcal{F}_{\text{T-LEDGER}}$, we need to specify how Blockify works. Blockify is a simple procedure that takes as input the next block to be added to the state, and outputs a concatenation of the transactions contained in the block. This means that the state of $\mathcal{F}_{\text{LEDGER}}$ (which will correspond also to the state of $\mathcal{F}_{\text{T-LEDGER}}$) is represented by just list of transactions. We do not specify how ExtendPolicy works, as any realization of ExtendPolicy can be used in our formalization. We provide a more detailed description of $\mathcal{F}_{\text{T-LEDGER}}$ in the full version [3]. We note that $\mathcal{F}_{\text{T-LEDGER}}$ does not specify who gets the fee, but this would not be difficult to do since $\mathcal{F}_{\text{LEDGER}}$ keeps track of the party that generated each block. Hence, it would be easy to modify $\mathcal{F}_{\text{T-LEDGER}}$ to keep track of which party gets the fees of the transactions that constitute a block. Another simplification we make is to consider fixed relation between the cost required to execute a transaction (or call a contract as we will see) and the complexity of the transaction (or the contract call). In system like Ethereum this is not the case, as the fee that a party pays depends on the complexity of the transaction (which determines the amount of gas) and on the gas price. This means that how fast and if a transaction will be executed depends on the product of gas price and amount of required gas. We could modify $\mathcal{F}_{\text{T-LEDGER}}$ (and the other functionalities we will consider) to accommodate for an additional mechanism that allows the adversary communicating to the functionality the average gas price, in such a say that we can use this gas cost to decide whether to accept or reject a transaction. However, since these aspects are not relevant for our results, to simplify the description of our already involved ideal functionalities, we have decided to not include such mechanisms in our model.

5 The Smart-Contract-Enabled Transaction Ledger

In this section we define the functionality $\mathcal{F}_{\text{TSC-Ledger}}$ that, in addition to $\mathcal{F}_{\text{T-Ledger}}$, captures a ledger that enables a large class of smart contracts. $\mathcal{F}_{\text{TSC-Ledger}}$ internally runs $\mathcal{F}_{\text{T-Ledger}}$ and a smart contract (formally defined by means of an additional ideal functionality). The contract has a state that can be updated by any party that can afford to pay a fee (that depends on the contract and on the input). After any valid update, the new contract state is pushed onto the $\mathcal{F}_{\text{T-Ledger}}$'s state. As we have alluded, in order for the contract to freely interact with $\mathcal{F}_{\text{T-Ledger}}$, the parameter $\mathcal{F}_{\mathsf{trap}}$ of $\mathcal{F}_{\text{T-Ledger}}$ is set to be equal to the identity of $\mathcal{F}_{\text{TSC-Ledger}}$, which will act as a bridge between the contract functionality and $\mathcal{F}_{\text{T-Ledger}}$. To simplify the description of the functionality, we describe the case where only one smart contract is running; however, it is easy to extend the functionality to the case where multiple smart contracts are running at the same time. A smart contract $\mathcal{F}_{\text{SC}}$ is a small functionality managed by $\mathcal{F}_{\text{TSC-Ledger}}$ that maintains its own state `cstate`. The behavior of $\mathcal{F}_{\text{SC}}$ is fully determined by three procedures: f_{CFee}, f_{filter} and f_{trans}.

- f_{CFee} (the contract fee function) takes as input the contract state `cstate`, the ledger state of $\mathcal{F}_{\text{T-Ledger}}$, a transaction, (which represents the input received by the contract's caller) and the fee specified in the input transaction. If the fee indicated is sufficient to update the contract state, then f_{CFee} returns the actual fee required to run the contract (which could be less than the fee indicated by the contract's caller). If the submitted fee is not sufficient, then the function returns $\perp$.
- f_{trans} (the state transition function) takes as input the payload of the input transaction, $\mathcal{F}_{\text{T-Ledger}}$'s state, and the contract state `cstate`, and returns a new contract state updated according to its inputs.
- f_{filter} (the filtering function) takes as input (1) the view that the contract's caller has of $\mathcal{F}_{\text{T-Ledger}}$'s state $\mathtt{state}_i$ and (2) the contract state, and returns an arbitrary sub-set of the information contained in $\mathtt{state}_i$.

The functionality $\mathcal{F}_{\text{TSC-Ledger}}$ is also parametrized by Fee, which represents the minimum fee that a party should pay in order to query a contract (to update the contract the fee might be higher). In more detail, $\mathcal{F}_{\text{TSC-Ledger}}$ accepts transactions with the following format: $\mathtt{tx} := (v, \mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_j^{\mathtt{E}}, \mathsf{aux}, \mathsf{fee}, \mathsf{type})$, where $\mathsf{type} \in \{\mathtt{E}, \mathsf{SC}\}$ denotes whether the transaction should be treated as a normal transaction or as a call to the contract. In particular, $\mathcal{F}_{\text{TSC-Ledger}}$ checks whether $\mathsf{type} = \mathtt{E}$ or $\mathsf{type} = \mathsf{SC}$. In the former case, $\mathcal{F}_{\text{TSC-Ledger}}$ removes the field type from the transaction and forwards it to $\mathcal{F}_{\text{T-Ledger}}$. In the latter, $\mathcal{F}_{\text{TSC-Ledger}}$ checks that $\mathsf{fee} \geq \mathsf{Fee}$ and that the issuer of the transaction has at least fee coins of type $\mathtt{E}$ in its wallet[10]. If this check is successful, then $\mathcal{F}_{\text{TSC-Ledger}}$ forwards the transaction and the current ledger state to $\mathcal{F}_{\text{SC}}$, which does the following: It uses f_{CFee} to check whether the fee specified in `tx` minus the fee required to query the contract (denoted with Fee) would be sufficient to update the contract state using

[10] $\mathcal{F}_{\text{TSC-Ledger}}$ can do this check since it has full access to $\mathcal{F}_{\text{T-Ledger}}$'s state and buffer.

the input aux. If f_{CFee} returns $\perp$, then the contract returns $(\mathtt{ko}, \mathtt{cstate}, \mathsf{fee})$. Else, if f_{CFee} returns $\mathsf{fee}^{\mathsf{SC}}$, $\mathcal{F}_{\mathrm{SC}}$ computes the updated contract state cstate by running f_{trans} on input the payload of tx (denoted with aux), the ledger state, and the contract state, and returns $(\mathtt{ok}, \mathtt{cstate}, \mathsf{fee}^{\mathsf{SC}} + \mathsf{Fee})$. $\mathcal{F}_{\text{TSC-Ledger}}$ upon receiving $(\mathsf{Flag}_{\mathsf{C}}, \mathtt{cstate}, \mathsf{actualfee})$ from $\mathcal{F}_{\mathrm{SC}}$, constructs and sends to $\mathcal{F}_{\text{T-Ledger}}$ the transaction $\mathtt{tx}^{\mathtt{E}} := (0, \mathsf{addr}_i^{\mathtt{E}}, 0^{\lambda}, (\mathsf{Flag}_{\mathsf{C}}, \mathsf{aux}, \mathtt{cstate}, \mathcal{F}_{\mathrm{SC}}.\mathsf{id}), \mathsf{actualfee})$ using the command SUBMIT-TRAPDOOR, where we recall that aux is the payload of tx, $\mathsf{Flag}_{\mathsf{C}} \in \{\mathtt{ok}, \mathtt{ko}\}$, and $\mathcal{F}_{\mathrm{SC}}.\mathsf{id}$ is the identifier of SC. We note that the transaction $\mathtt{tx}^{\mathtt{E}}$ is a standard $\mathcal{F}_{\text{T-Ledger}}$ transaction that contains in its payload the updated state of the contract (or the old state if the fee was not sufficient), the input used to eventually update the contract's state, and the fee actualfee such that:

- if $\mathsf{Flag}_{\mathsf{C}} = \mathtt{ko}$ (i.e. the fee specified by the contract's caller was not sufficient to update the contract state) then $\mathsf{actualfee} = \mathsf{fee}$
- if $\mathsf{Flag}_{\mathsf{C}} = \mathtt{ok}$ (i.e. fee was sufficient to update the contract's state) then $\mathsf{actualfee} \leq \mathsf{fee}$.

Note that it might be that $\mathsf{actualfee} < \mathsf{fee}$ in the case where the fee required to update the contract state is less that fee. That is, $\mathcal{F}_{\text{TSC-Ledger}}$ only charges the contract caller exactly for the fee required to run the contract. When fee is insufficient to complete execution of the contract, the issuer of the transaction pays the full amount of fee even though no change to the contract state is committed. (This is consistent with Ethereum and other blockchains that support Turing-complete smart contracts.) We refer to the full version [3] for a more detailed description of $\mathcal{F}_{\text{TSC-Ledger}}$ and for the abstraction of $\mathcal{F}_{\mathrm{SC}}$.

6 The EET Ledger

We can now define the functionality $\mathcal{F}_{\text{Ledger}}^{\mathsf{EET}}$. $\mathcal{F}_{\text{Ledger}}^{\mathsf{EET}}$ internally runs $\mathcal{F}_{\text{TSC-Ledger}}$, parametrized by a contract $\mathcal{F}_{\mathrm{SC}}^{\mathtt{T}}$. $\mathcal{F}_{\mathrm{SC}}^{\mathtt{T}}$ maintains a token T, and allows parties to issue transactions with respect to such a token. Any party that has some tokens can sent it to another party by querying the contract $\mathcal{F}_{\mathrm{SC}}^{\mathtt{T}}$. However, invoking the contract requires payment of a fee in the native currency E, even if the transaction involves only tokens. To mitigate this problem, our functionality allows a sender P_i to send tokens to another party P_j, even if P_i does not have native coins. In particular, the sender will pay a fee of at least del-fee tokens T to a special party M, called the intermediary, and M will pay the fee in E on the behalf of the sender (del-fee is a fixed amount of tokens that parametrizes our functionality). The functionality guarantees that either the transaction by P_i becomes part of the ledger state *and* M gets a fixed amount of tokens del-fee, or nothing happens. We propose a more detailed description of $\mathcal{F}_{\text{Ledger}}^{\mathsf{EET}}$ and $\mathcal{F}_{\mathrm{SC}}^{\mathtt{T}}$ to the full version [3], and provide a more high level (but still formal) description of those functionalities below. The functionality $\mathcal{F}_{\text{Ledger}}^{\mathsf{EET}}$, interacts with a set of parties, with the adversary, and with a special party that we denote with M (the intermediary), and manages the *token wallet* addresses of the registered parties. We assume that a party P_0 initially holds

all of the available tokens[11]. We denote the token wallet addresses of P_0 and M with $\mathsf{addr}_0^{\mathtt{T}}$ and $\mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}$ respectively. Any time $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ receives a registration command from a party P_i, it registers P_i to the ledger $\mathcal{F}_{\text{TSC-LEDGER}}$, thus obtaining $\mathsf{addr}_i^{\mathtt{E}}$. It then generates a token wallet address $\mathsf{addr}_i^{\mathtt{T}}$ and returns $(\mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathtt{T}})$ to P_i. $(\mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathtt{T}})$ represents respectively the wallet addresses for the native currency $\mathtt{E}$ and for the token $\mathtt{T}$. $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ tolerates two types of transactions: *standard* and *delegated* transactions. Any registered party P_i can issue a standard transaction $\mathtt{tx}^{\mathtt{T}} := (v, \mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{fee})$, where v denotes the amount of tokens, $(\mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathtt{T}})$ are the addresses of the sender, $\mathsf{addr}_j^{\mathtt{T}}$ is the token wallet address of the receiver, and fee is the fee expressed in coins of type $\mathtt{E}$. $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ takes $\mathtt{tx}^{\mathtt{T}}$ and creates a transaction $\mathtt{tx}^{\mathtt{E}}$ for the ledger $\mathcal{F}_{\text{TSC-LEDGER}}$ that (1) has as a sender address $\mathsf{addr}_i^{\mathtt{E}}$, (2) has a fee fee, and (3) calls the contract $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ and includes in its payload what we call a *token transaction* $\mathtt{tx}' := (v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}})$.[12] $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ then forwards $\mathtt{tx}^{\mathtt{E}}$ to the ledger $\mathcal{F}_{\text{TSC-LEDGER}}$ on behalf of P_i. The contract $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ maintains a set $\mathtt{token\text{-}set}$ as part of its state, and if the fee specified in $\mathtt{tx}^{\mathtt{E}}$ is sufficient, it updates its state by adding $\mathtt{tx}'$ to $\mathtt{token\text{-}set}$ and returns $(\mathtt{ok}, \mathtt{cstate}, \mathsf{actualfee})$. Note that this means that the $\mathtt{tx}'$ is part of the contract state and appears in the $\mathcal{F}_{\text{TSC-LEDGER}}$'s state by definition. To complete this first part of the description of $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, it remains to specify the function f_{filter} (and f_{CFee}, which we describe later in this section) of $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$. f_{filter} receives as input the contract state and the state of $\mathcal{F}_{\text{TSC-LEDGER}}$ (which we denote $\mathtt{state}$) and, for each transaction $\mathtt{tx}$ in $\mathtt{state}$ such that $\mathtt{tx}^{\mathtt{E}} := (0^{\lambda}, \mathsf{addr}_i^{\mathtt{E}}, 0^{\lambda}, \mathsf{aux}^{\mathtt{E}}, \mathsf{fee}^{\mathtt{E}})$ (where $\mathsf{aux}^{\mathtt{E}} = (\mathtt{ok}, \mathtt{tx}', \mathtt{cstate}^{\star}, \mathcal{F}_{\text{SC}}^{\mathtt{T}}.\mathsf{id})$), adds $\mathtt{tx}'$ to $\mathtt{state}^{\mathtt{T}}$ if and only if:

1. $\mathtt{tx}'$ appears in $\mathtt{token\text{-}set}$ (which is part of the token state).
2. $\mathtt{tx}' := (v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}})$ and the sum of tokens in the token transactions stored so far in $\mathtt{state}^{\mathtt{T}}$ with receiver address $\mathsf{addr}_i^{\mathtt{T}}$, minus the sum of coins in the state with sender address $\mathsf{addr}_i^{\mathtt{T}}$, is greater than or equal to v.

$\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ captures the main characteristics of a token, relying on the smart contract to filter out invalid transactions. Unfortunately, the mechanism that we have discussed so far has a major drawback: if a party wants to issue a token transaction, they must have the required amount of coins of type $\mathtt{E}$ to query the contract. To get rid of this requirement, $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ admits what we call *delegated transactions*. A party that wants to issue a delegated transaction submits $\mathtt{tx}^{\mathtt{T}} := (v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{fee}^{\mathtt{T}})$ to $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, which in turns asks the special party denoted M to pay the fee in $\mathtt{E}$ in exchange of (at least) $\mathsf{del\text{-}fee}$ tokens $\mathtt{T}$, which will be taken from P_i's account. If M is honest and $\mathsf{fee}^{\mathtt{T}} \geq \mathsf{del\text{-}fee}$, (where we recall that $\mathsf{del\text{-}fee}$ is the minimum fee required for the delegation to be considered,) then $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ submits a call to the contract $\mathcal{F}_{\text{SC}}^{\mathtt{T}}$ on behalf of M with the input (the payload of the transaction) $\mathsf{aux} := (([v, \mathsf{fee}^{\mathtt{T}}], \mathsf{addr}_i^{\mathtt{T}}, [\mathsf{addr}_j^{\mathtt{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}]))$. If M

[11] As before, we could have multiple addresses having different amounts of tokens, but for simplicity, we assume that only one party initially holds tokens.

[12] The payload also includes an identifier chosen by the adversary, which we omit in this informal description.

has enough coins of type $\mathtt{E}$ to afford the call to $\mathcal{F}_{\mathrm{SC}}^{\mathsf{T}}$, then aux will become part of the contract state. To accommodate for this special input, we modify the filtering function f_{filter} of $\mathcal{F}_{\mathrm{SC}}^{\mathsf{T}}$ in such a way that the value aux can also be understood as two atomic token transactions: the first moves v tokens from the wallet address $\mathsf{addr}_i^{\mathsf{T}}$ to the wallet address $\mathsf{addr}_j^{\mathsf{T}}$, and the second moves $\mathsf{fee}^{\mathsf{T}}$ from the wallet address $\mathsf{addr}_i^{\mathsf{T}}$ to the wallet address $\mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}$. It remains to specify how the contract computes the fee. The function f_{CFee} charges Fee coins of type $\mathtt{E}$ for each token transaction encoded in aux (the input that is used to update the contract state). Hence, for a non-delegated token transaction, f_{CFee} would return Fee, and for a delegated token transaction, it would return $2\mathsf{Fee}$. In addition to this fee, we need to consider the fee required simply to query the contract. Hence, the total cost of a non-delegated transaction would be of $2\mathsf{Fee}\mathtt{E}$, and the total cost of a delegated transaction would be $3\mathsf{Fee}\mathtt{E}$. We stress that this is a simplified method of computing the fee, and that a more fine-grained calculation could be used to capture what actually happens in the real world.

Constructions and Experimental Evaluation. We have already highlighted how our construction works in Sect. 2.2. We compared our system with GSN and with users that make only *self-funded transactions* (i.e., user that do not want to interact with the intermediary and have his own Ether to afford for the token transaction). Our experiments indicate a 4-5x overhead in gas consumption when using the GSN as opposed to using our EET contract. This is the cost of the complexity of the GSN, a cost that is very unattractive for projects that do not require the genericity of the GSN. We also show that our contract consumes less than twice the gas of a standard self-funded token transaction, which we believe is a reasonable compromise for the added user experience. We refer the reader to the full version [3] for more detail on our experimental results.

A Our Protocol: How to Realize $\mathcal{F}_{\mathrm{LEDGER}}^{\mathsf{EET}}$

Our protocol is described in the $\mathcal{F}_{\text{T-LEDGER}}$-hybrid world, where $\mathcal{F}_{\text{T-LEDGER}}$ is parametrized by $\mathcal{F}_{\mathsf{trap}} = \perp$, and the fee function f which, upon receiving an input transaction $\mathtt{tx}^{\mathtt{E}}$, does the following: 1) Parse $\mathtt{tx}$ as $(v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{aux}, \mathsf{fee})$; 2) if $\mathsf{aux} = \perp$, then return Fee; 3) Otherwise, return $\mathsf{Fee} + |\mathsf{aux}|/\kappa\mathsf{Fee}$. In a nutshell, the fee required for a transaction to settle in the $\mathcal{F}_{\text{T-LEDGER}}$'s state is Fee, plus and additional Fee for each κ bits contained in the payload, where Fee and κ are part of the description of f. We provide the formal description of our protocol in Fig. 2. At a very high level, the protocol works as follows: Each party registers with $\mathcal{F}_{\text{T-LEDGER}}$ and runs $\mathtt{Kgen}(1^{\lambda})$ to obtain $(\mathsf{sk}_i^{\mathsf{T}}, \mathsf{addr}_i^{\mathsf{T}})$, where $\mathsf{addr}_i^{\mathsf{T}}$ represents the token wallet address. A party P_i that wants to send $v\mathtt{T}$ to P_j and has at least $2\mathsf{Fee}$ coins of type $\mathtt{E}$ can do so by issuing a transaction for $\mathcal{F}_{\text{T-LEDGER}}$ that contains

in its payload $\mathsf{aux} := (v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{id}, \sigma_i^{\mathtt{T}})$, where id is a random value, and $\sigma_i^{\mathtt{T}}$ is a signature of $(v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{id})$ that verifies under the verification key $\mathsf{addr}_i^{\mathtt{T}}$. We require P_i to pay a fee of at least $2\mathsf{Fee}$ because we assume that, in this case, $|\mathsf{aux}| = \kappa$. When an honest party P_i receives the command $(\text{READ}, \mathsf{sid}, \mathtt{T})$, they shall retrieve $\mathcal{F}_{\text{T-LEDGER}}$'s state, filter out the payload of each transaction (thus obtaining only the information related to token transactions), and output only the *valid* token transactions. A token transaction $(v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{id}, \sigma_i^{\mathtt{T}})$ is valid if $\mathsf{addr}_i^{\mathtt{T}}$ has received at least v tokens, $\sigma_i^{\mathtt{T}}$ is a signature of $(v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{id})$ that verifies under the verification key $\mathsf{addr}_i^{\mathtt{T}}$, and there does not exist any other token transaction with the same sender address and identifier id. Our protocol allows any party P_i that does not have coins of type $\mathtt{E}$ to delegate the payment of the fee to M, paying M with at least $\mathsf{del\text{-}fee}$ tokens $\mathtt{T}$. To do so, P_i creates $m := ([v, \mathsf{del\text{-}fee}], \mathsf{addr}_i^{\mathtt{T}}, [\mathsf{addr}_j^{\mathtt{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}], \mathsf{id})$ and signs it, thus obtaining $\sigma_i^{\mathtt{T}}$. P_i then sends $(m, \sigma_i^{\mathtt{T}})$ to M. The honest M then creates a transaction for $\mathcal{F}_{\text{T-LEDGER}}$ that includes $(m, \sigma_i^{\mathtt{T}})$ in its payload and has a fee of at least $3\mathsf{Fee}$, and submits it. We require M to pay a fee of at least $3\mathsf{Fee}\mathtt{E}$ because we assume that, in this case, the payload of the transaction is 2κ bits (as, indeed, the payload of this type of transaction contains more information). The honest M would immediately create and submit such a transaction, whereas the corrupted M might decide when (and if) to create the transaction. We require each token transaction to contain a random identifier in order to avoid replay attacks; without such an identifier, the adversary could take the payload of any transaction from $\mathcal{F}_{\text{T-LEDGER}}$'s state, (for instance, the payload of a transaction that moves v tokens from the address $\mathsf{addr}_i^{\mathtt{T}}$ of an honest party to some potentially adversarial address,) copy this payload, and use it to generate a new transaction for $\mathcal{F}_{\text{T-LEDGER}}$. In this way, the adversary could empty the token wallet of the honest party without their knowledge. The other advantage of using identifiers is that an honest party that has delegated a transaction to a malicious intermediary can at any point decide to withdraw the delegation. Indeed, if M is not responding to a party that has delegated the transaction $m := ([v, \mathsf{del\text{-}fee}], \mathsf{addr}_i^{\mathtt{T}}, [\mathsf{addr}_j^{\mathtt{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}], \mathsf{id})$ for a long time, and m does not appear in the payload of any transaction that appears in the ledger's state, then P_i can withdraw the delegation by submitting (or delegating) a token transaction with the same identifier; then, at most one of these transactions will be valid and accepted by the functionality. We refer to Fig. 2 for the formal description of Π^{Token}.

Protocol Π^{Token}

- The issuer P_0 does the following:
  1. Register to $\mathcal{F}_{\text{T-LEDGER}}$, thus obtaining addr_0.
  2. Compute $(\mathsf{sk}_0^{\mathsf{T}}, \mathsf{addr}_0^{\mathsf{T}}) \xleftarrow{\$} \mathtt{Kgen}(1^\lambda)$.
- The intermediary M registers to $\mathcal{F}_{\text{T-LEDGER}}$, thus obtaining $\mathsf{addr}_{\mathsf{M}}$, and computes $(\mathsf{sk}_{\mathsf{M}}^{\mathsf{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}) \xleftarrow{\$} \mathtt{Kgen}(1^\lambda)$.

- Upon receiving $(\text{REGISTER}, \mathsf{sid})$, the party P_i sends $(\text{REGISTER}, \mathsf{sid})$ to $\mathcal{F}_{\text{T-LEDGER}}$, thus obtaining addr_i, and computes $(\mathsf{sk}_i^{\mathsf{T}}, \mathsf{addr}_i^{\mathsf{T}}) \xleftarrow{\$} \mathtt{Kgen}(1^\lambda)$.

- P_i, upon receiving $(\text{SUBMIT-DELEGATION}, \mathsf{sid}, \mathtt{tx}^{\mathsf{T}})$, parses $\mathtt{tx}^{\mathsf{T}}$ as $(v, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}}, \mathsf{fee}^{\mathsf{T}})$ and does the following:
  1. If $\mathsf{fee}^{\mathsf{T}} < \mathsf{del\text{-}fee}$, then ignore the command. Otherwise, continue.
  2. Sample $\mathsf{id} \xleftarrow{\$} \{0,1\}^\lambda$ and define $m := (([v, \mathsf{fee}], \mathsf{addr}_i^{\mathsf{T}}, [\mathsf{addr}_j^{\mathsf{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}]), \mathsf{id})$.
  3. Compute $\sigma_i^{\mathsf{T}} \xleftarrow{\$} \mathtt{Sign}(\mathsf{sk}_i^{\mathsf{T}}, m)$.
  4. Send $(\mathsf{delegate}, m, \sigma_i^{\mathsf{T}})$ to M.
- M, upon receiving $(\mathsf{delegate}, m, \sigma_i^{\mathsf{T}})$ from P_i, does the following:
  1. Parse m as $(([v, \mathsf{fee}^{\mathsf{T}}], \mathsf{addr}_i^{\mathsf{T}}, [\mathsf{addr}_j^{\mathsf{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}]), \mathsf{id})$.
  2. If $\mathtt{Ver}(\mathsf{addr}_i^{\mathsf{T}}, m, \sigma_i^{\mathsf{T}}) = 0$ or $\mathsf{fee}^{\mathsf{T}} < \mathsf{del\text{-}fee}$, then ignore the message. Otherwise, continue.
  3. Define $\mathtt{tx}_{\mathsf{M}} = (0, \mathsf{addr}_{\mathsf{M}}, 0^\lambda, (m, \sigma_i^{\mathsf{T}}), 3\mathsf{Fee})$.
  4. Send (ACCEPT, P_i) to P_i and $(\text{SUBMIT}, \mathsf{sid}, \mathtt{tx}_{\mathsf{M}})$ to $\mathcal{F}_{\text{T-LEDGER}}$.
- P_i, upon receiving $(\text{SUBMIT}, \mathsf{sid}, \mathtt{tx}^{\mathsf{T}})$, parses $\mathtt{tx}^{\mathsf{T}}$ as $(v, \mathsf{addr}_i, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j, \mathsf{addr}_j^{\mathsf{T}}, \mathsf{fee}, \mathsf{Coin})$ and does the following:
  - If $\mathsf{Coin} = \mathtt{T}$ and $\mathsf{fee} \geq 2\mathsf{Fee}$ then:
    - Sample $\mathsf{id} \xleftarrow{\$} \{0,1\}^\lambda$, define $m := ((v, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}}), \mathsf{id})$ and compute $\sigma_i^{\mathsf{T}} \leftarrow \mathtt{Sign}(\mathsf{sk}, m)$.
    - Define $\mathtt{tx} = (0, \mathsf{addr}_i, 0^\lambda, (m, \sigma_i^{\mathsf{T}}), \mathsf{fee})$.
  - If $\mathsf{Coin} = \mathtt{E}$ and $\mathsf{fee} \geq \mathsf{Fee}$ then
    Define $\mathtt{tx} := (v, \mathsf{addr}_i, \mathsf{addr}_j, \bot, \mathsf{fee})$.
  - Send $I = (\text{SUBMIT}, \mathsf{sid}, \mathtt{tx})$ to $\mathcal{F}_{\text{T-LEDGER}}$.

- Upon receiving $(\text{READ}, \mathsf{sid}, \mathsf{type})$, P forwards the command $(\text{READ}, \mathsf{sid})$ to $\mathcal{F}_{\text{T-LEDGER}}$.
- Upon receiving $\mathtt{state}$ from $\mathcal{F}_{\text{T-LEDGER}}$, if $\mathsf{type} = \mathtt{E}$, then P does the following:
  - Initialize an empty list $\mathtt{state}^{\mathtt{E}}$.
  - For each $\mathtt{tx} \in \mathtt{state}$ such that $\mathtt{tx} = (v, \mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_j^{\mathtt{E}}, \bot, \mathsf{fee})$, add $\mathtt{tx}$ to $\mathtt{state}^{\mathtt{E}}$.
  - Return $(\text{READ}, \mathsf{sid}, \mathtt{state}^{\mathtt{E}})$.

  Otherwise, P does the following:
  - Initialize the the list $\mathtt{state}^{\mathtt{T}}$ with $(y, 0^\lambda, \mathsf{addr}_0, 0)$ and, for each $\mathtt{tx}^{\mathtt{E}} = (0, \mathsf{addr}_i^{\mathtt{E}}, 0^\lambda, \mathsf{aux}^{\mathtt{E}}, \mathsf{fee})$ in $\mathtt{state}$ where $\mathsf{aux}^{\mathtt{E}} = (\mathsf{v}^{\mathtt{T}}, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addrs}, \mathsf{id}^{\mathtt{T}}, \sigma_i^{\mathtt{T}})$, do the following:
    - If $\mathsf{checkvalidity}(\mathsf{aux}^{\mathtt{E}}, \mathtt{state}^{\mathtt{T}}) = 1$, then add $(\mathsf{v}^{\mathtt{T}}, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addrs}, \mathsf{id}^{\mathtt{T}})$ to $\mathtt{state}^{\mathtt{T}}$.
  - Return $(\text{READ}, \mathsf{sid}, \mathtt{state}^{\mathtt{T}})$.

$\mathsf{checkvalidity}(\mathsf{aux}^{\mathtt{E}}, \mathtt{state}^{\mathtt{T}})$

- Parse $\mathsf{aux}^{\mathtt{E}}$ as $(\mathsf{v}^{\mathtt{T}}, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addrs}, \mathsf{id}_i^{\mathtt{T}}, \sigma_i^{\mathtt{T}})$ and define $m := (\mathsf{v}^{\mathtt{T}}, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addrs}, \mathsf{id}_i^{\mathtt{T}})$.
- If $\mathtt{Ver}(\mathsf{addr}_i^{\mathtt{T}}, m, \sigma_i^{\mathtt{T}}) = 0$, then return 0. Otherwise, continue.
- Initialize $\mathsf{balance} \leftarrow 0$.
- For each $\mathtt{tx}^\star = (\mathsf{v}^\star, \mathsf{addr}_i^\star, \mathsf{addrs}^\star, \mathsf{id}_i^\star)$ in $\mathtt{state}^{\mathtt{T}}$:
  - If $\mathsf{addr}^\star = \mathsf{addr}_i^{\mathtt{T}}$ and $\mathsf{id}_i^\star = \mathsf{id}_i^{\mathtt{T}}$, then return 0.
  - If $\mathsf{addr}^\star = \mathsf{addr}_i^{\mathtt{T}}$, then compute $\mathsf{balance} \leftarrow -\sum_k \mathsf{v}^\star[k]$.
  - For each k such that $\mathsf{addrs}^\star[k] = \mathsf{addr}_i^{\mathtt{T}}$, compute $\mathsf{balance} \leftarrow +v^\star[k]$.
- If $\sum_k \mathsf{v}^{\mathtt{T}}[k] \geq \mathsf{balance}$, then return 1. Otherwise, return 0.

Fig. 2. Our protocol.

References

1. Ethereum gas station network (GSN) documentation. https://docs.opengsn.org/
2. Al-Balaghi, A.: The state of meta transactions - 2020 (2020). https://medium.com/biconomy/the-state-of-meta-transactions-2020-506840e37e75
3. Andrews, J., Ciampi, M., Zikas, V.: Etherless ethereum tokens: Simulating native tokens in ethereum. Cryptology ePrint Archive, Report 2021/766 (2021). https://ia.cr/2021/766
4. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: composable proof-of-stake blockchains with dynamic availability. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018, pp. 913–930. ACM Press (2018). https://doi.org/10.1145/3243734.3243848
5. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: a composable treatment. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 324–356. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-63688-7_11
6. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press (2001). https://doi.org/10.1109/SFCS.2001.959888
7. Canetti, R.: Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239 (2003). https://eprint.iacr.org/2003/239
8. Chakravarty, M.M., Karayannidis, N., Kiayias, A., Jones, M.P., Vinogradova, P.: Babel fees via limited liabilities. arXiv preprint arXiv:2106.01161 (2021)
9. Daian, P., Pass, R., Shi, E.: Snow white: robustly reconfigurable consensus and applications to provably secure proof of stake. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 23–41. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-32101-7_2
10. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_10
11. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 291–323. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_10
12. Griffith, A.: Ethereum meta transactions (2018). https://medium.com/@austin_48503/ethereum-meta-transactions-90ccf0859e84
13. Kerber, T., Kiayias, A., Kohlweiss, M.: KACHINA - foundations of private smart contracts. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, 21–25 June 2021, pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00002
14. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: Privacy-preserving proof-of-stake. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, 19–23 May 2019, pp. 157–174. IEEE (2019). https://doi.org/10.1109/SP.2019.00063
15. Kiayias, A., Litos, O.S.T.: A composable security treatment of the lightning network. In: 33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, 22–26 June 2020, pp. 334–349. IEEE (2020). https://doi.org/10.1109/CSF49147.2020.00031, https://doi.org/10.1109/CSF49147.2020.00031

16. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 643–673. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-56614-6_22
17. Pass, R., Shi, E.: The sleepy model of consensus. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part II. LNCS, vol. 10625, pp. 380–409. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-70697-9_14
18. Seres, I.A.: On blockchain metatransactions. In: 2020 IEEE International Conference on Blockchain (Blockchain), pp. 178–187. IEEE (2020)
19. Team, A.: Algorand developer documentation (2021). https://developer.algorand.org/docs/
20. Vogelsteller, F., Buterin, V.: Eip 20: Erc-20 token standard (2015). https://eips.ethereum.org/EIPS/eip-20