Etherless Ethereum Tokens: Simulating Native Tokens in Ethereum

Citation for published version:
Andrews, J, Ciampi, M & Zikas, V 2022, Etherless Ethereum Tokens: Simulating Native Tokens in Ethereum. in S Dolev, A Meisels & J Katz (eds), *Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, Proceedings.* Lecture Notes in Computer Science, vol. 13301, Springer, Cham, pp. 388-407, 6th International Symposium on Cyber Security Cryptography and Machine Learning, CSCML 2022, Beer Sheva, Israel, 30/06/22. https://doi.org/10.1007/978-3-031-07689-3_29

Digital Object Identifier (DOI):
10.1007/978-3-031-07689-3_29

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, Proceedings

Etherless Ethereum Tokens:
Simulating Native Tokens in Ethereum

John Andrews[1], Michele Ciampi[2], and Vassilis Zikas*[3]

[1]Sunday Group, jandrews@sundaygroupinc.com
[2]The University of Edinburgh, michele.ciampi@ed.ac.uk
[3]Purdue University, 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 and counterintuitive—at least for less tech-savvy users—user experience, especially when the token creator intends to switch to their own blockchain down the line, or wishes the flexibility of transferring the token to a different smart-contract enabled 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 creator to allow its users to transact in a closed-economy manner, i.e., having only tokens on their wallet and paying any transaction fees in token units rather than 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. We have implemented and benchmarked our system and compared it to another solution for obtaining similar functionality in Ethereum, i.e., the Gas Station Networks (GSN); in addition to being the first system with a rigorous security analysis, we demonstrate that EETs are not only far easier to deploy, but are also far 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. The paradigm in which such a need is most prominent is in the creation of Ethereum tokens (e.g., ERC-20 tokens [VB15]). The usual implementation of such tokens requires that, for a token-holder to use them (exchange them with other tokens, or transact with other holders of the same token,) they need to also hold Ether, to be used to fuel the Ethereum transaction. This poses a challenge both to the token creator and the users: 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, which

*Work supported in part by NSF grant no. 2055599 and by Sunday Group.

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. In order to hold their promise to the investors, the token creator would then have the investors create a new token-specific 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 propose 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 can be a result of independent interest. 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* [Gri18, AB20, gsn]. However, to our knowledge, our work is the first formal treatment and security analysis of the concept.

At a less technical level, we believe that in addition to offering a more intuitive, closed-economy user experience, such a mechanism 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. Nonetheless, the study of this market effect is outside the scope of our current work. 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) [gsn], both in terms of simplicity of deployment and in terms of gas usage. We also compare such a deployment

[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.

with how a native token could perform on Ethereum and demonstrate that the overhead makes the flexibility which is offered by 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:

A. A universally composable (UC) [Can01] treatment of ledgers supporting a broad class of smart contracts, which includes token contracts (e.g. ERC 20).

B. A design and UC security analysis of EETs.

C. 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 [BMTZ17, BGK+18, GKL15, PSs17, GKL17, DPS19, PS17]. Badertscher *et al.* [BMTZ17] 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 $\mathsf{Validate}$. 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 $\texttt{state}$, 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 [BGK+18, KKKZ19].

UC [BMTZ17] 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 guarantrees.[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 (cf. [KZZ16]).

[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.

2.1.1 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 [Can03] would. In a nutshell, instead of having $\mathsf{Validate}$ 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 [Can03]. $\mathcal{F}_{\text{T-LEDGER}}$ accepts transactions with the format $\texttt{tx} := (v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{fee})$ 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 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 $\mathsf{fee} \geq f(\texttt{tx})$, 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 [BMTZ17]—using a specific existentially-unforgeable signatures scheme—to prove security of our ledger for a standard Bitcoin-style blockchain protocol, such as 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.

2.1.2 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, we define an augmented functionality, which we denote $\mathcal{F}_{\text{TSC-LEDGER}}$. $\mathcal{F}_{\text{TSC-LEDGER}}$ 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 $\texttt{cstate}$—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.

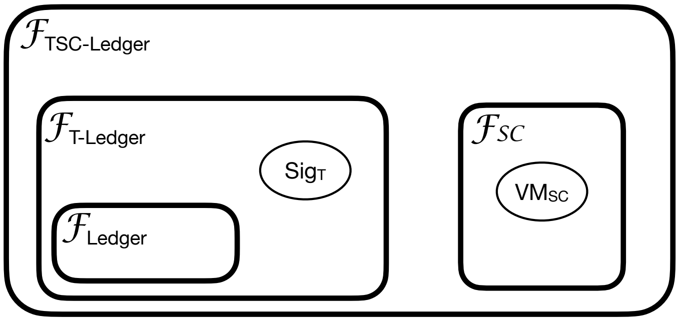

Figure 1: The Smart-Contract-Enabled Transaction Ledger Functionality $\mathcal{F}_{\text{TSC-LEDGER}}$

The construction of $\mathcal{F}_{\text{TSC-LEDGER}}$ from its components is illustrated in Figure 1. $\mathcal{F}_{\text{TSC-LEDGER}}$ accepts either standard transactions in the native currency $\texttt{E}$ (that are forwarded to $\mathcal{F}_{\text{T-LEDGER}}$) or

[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.

inputs/transactions that are intended as queries to the contract $\mathcal{F}_{\mathrm{SC}}$. Upon receiving such a query for the smart contract, $\mathcal{F}_{\text{TSC-LEDGER}}$ forwards the query to $\mathcal{F}_{\mathrm{SC}}$, 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 $\mathcal{F}_{\mathrm{SC}}$ 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, $\mathcal{F}_{\mathrm{SC}}$ 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.

2.1.3 Tokens as Smart Constracts

Given the above smart-contract-enabled ledger, it is straightforward to capture a smart contract for creating a standard (e.g. ERC 20 [VB15]) Ethereum token by instantiating $\mathcal{F}_{\text{TSC-LEDGER}}$ with contract functionality that stores and updates the state (balances for different addresses) of such a token. Note that this results in a token-enabled transaction ledger $\mathcal{F}^{\mathsf{Token}}_{\mathrm{LEDGER}}$ which allows parties both to issue transactions in the native coin `E`, and to exchange tokens `T`.

In more detail, $\mathcal{F}^{\mathsf{Token}}_{\mathrm{LEDGER}}$ instantiates $\mathcal{F}_{\text{TSC-LEDGER}}$ with a token-contract $\mathcal{F}^{\mathsf{T}}_{\mathrm{SC}}$ which works as follows: $\mathcal{F}^{\mathsf{T}}_{\mathrm{SC}}$ 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 $(v, \mathsf{addr}^{\mathsf{T}}_i, \mathsf{addr}^{\mathsf{T}}_j)$, 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, $\mathcal{F}^{\mathsf{T}}_{\mathrm{SC}}$ 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 `E`. Indeed, the issuer of the token transaction, in order to query the contract $\mathcal{F}^{\mathsf{T}}_{\mathrm{SC}}$, needs to possess coins of type `E`.

2.1.4 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 `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 `E`-to-`T` rate. This is captured by tweaking the token-enabled ledger $\mathcal{F}^{\mathsf{Token}}_{\mathrm{LEDGER}}$ toward an EET-enabled ledger, denoted as $\mathcal{F}^{\mathsf{EET}}_{\mathrm{LEDGER}}$, which supports an additional input called SUBMIT-DELEGATION. Upon receiveing SUBMIT-DELEGATION, $\mathcal{F}^{\mathsf{EET}}_{\mathrm{LEDGER}}$ allows the user to issue a token transaction which pays a fee, in `T`, to a special party, called *intermediary* (that we

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

[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.

denote with M), in exchange for the intermediary submitting the token transaction to $\mathcal{F}_{\mathrm{SC}}^{\mathtt{T}}$ and paying the $\mathtt{E}$ needed for the token contract to process the transaction.

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 cover for the fee can issue a transaction $\mathtt{tx} = (0, \mathsf{addr}_i, 0^\lambda, (\mathsf{aux}, \sigma), \mathsf{fee})$, where $\mathsf{aux} = (v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}})$ and σ is a signature of aux that verifies under $\mathsf{addr}_i^{\mathtt{T}}$.[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 $\mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}$ be the token wallet address of M. To delegate a transaction, the sender P_i creates a special token transaction $\mathsf{aux} = ([v, \mathsf{del\text{-}fee}], \mathsf{addr}_i^{\mathtt{T}}, [\mathsf{addr}_j^{\mathtt{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}])$ (where $\mathsf{del\text{-}fee}$ is a fee expressed in $\mathtt{T}$ 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 $\mathsf{addr}_i^{\mathtt{T}}$ to $\mathsf{addr}_j^{\mathtt{T}}$, and the second moves $\mathsf{del\text{-}fee}$ from $\mathsf{addr}_i^{\mathtt{T}}$ to $\mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}$. 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 $(v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}})$ is valid if the sum of tokens with receiver address $\mathsf{addr}_i^{\mathtt{T}}$ 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 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.

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

The target contract, in turn, is allowed to extract any payment it wishes from the user, e.g. tokens. 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 exhange, 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.

Remark (Contract-based vs Native Tokens). Recently, the blockchain/cryptocurrency community has been entertaining the idea of making tokens native to the cryptocurrency chain. For example, [Kia] outlines a plan to introduce such a mechanism, which would allow a user to post a token transaction along with a token-to-native exchange rate he is willing to pay; any miner/minter who agrees with the rate can pick this up and create a transaction in which they “foot the bill” in terms of native-currency fees, in exchange for tokens at the proposed exchange rate. This clearly yields an advantage in terms of fees needed for the transaction, but it does come at a cost: (1) 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. (2) 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 Appendix A, 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.

3 Preliminaries

We use “=” to denote equality of two different elements (i.e. $a = b$ then...) and “$\leftarrow$” as the assignment operator (e.g. to assign to a the value of b we write $a \leftarrow b$). A randomized assignment is denoted with $a \overset{\$}{\leftarrow} A$, where A is a randomized algorithm and the randomness used by A is not explicit. We call a function $\nu : \mathbb{N} \rightarrow \mathbb{R}^+$ *negligible* if for every positive polynomial $p(\kappa)$, there exists a $\kappa_0 \in \mathbb{N}$ such that for all $\kappa > \kappa_0 : \nu(\kappa) < 1/p(\kappa)$.

3.1 Signatures

Definition 3.1 (Signature scheme [Can03])**.** A triple of PPT algorithms $(\texttt{Kgen}, \texttt{Sign}, \texttt{Ver})$ is called a *signature scheme* if it satisfies the following properties.

Completeness: For every pair $(s, v) \overset{\$}{\leftarrow} \texttt{Kgen}(1^\lambda)$, and every $m \in \{0,1\}^\lambda$, we have that $\Pr[\texttt{Ver}(v, m, \texttt{Sign}(s, m)) = 0] < \nu(\lambda)$.

Consistency (non-repudiation): For any m, the probability that $\texttt{Kgen}(1^\lambda)$ generates (s, v) and $\texttt{Ver}(v, m, \sigma)$ generates two different outputs in two independent invocations is smaller than $\nu(\lambda)$.

Unforgeability: For every PPT $\mathcal{A}$, there exists a negligible function ν, such that for all auxiliary input $z \in \{0,1\}^\star$ it holds that:

$$\Pr[(s, v) \overset{\$}{\leftarrow} \texttt{Kgen}(1^\lambda); (m, \sigma) \overset{\$}{\leftarrow} \mathcal{A}^{\texttt{Sign}(s,\cdot)}(z, v) \wedge \\ \texttt{Ver}(v, m, \sigma) = 1 \wedge m \notin Q] < \nu(\lambda)$$

where Q denotes the set of messages whose signatures were requested by $\mathcal{A}$ from the oracle $\texttt{Sign}(s, \cdot)$.

4 The Model

Following the recent line of works proving composable security of blockchain ledgers [BMTZ17, BGK+18], we provide our protocols and security proofs in Canetti's universal composition (UC) framework [Can01]. In this section we discuss the main components of our real-world model (including the associated hybrids). We assume that the reader is familiar with simulation-based security and has basic knowledge of the UC framework. We review all the aspects of the execution model that are needed for our protocols and proof, but omit some of the low-level details and refer the interested reader to relevant works wherever appropriate.

We now recall the mechanics of activations in UC. In a UC protocol execution, an honest party (ITI) gets activated either by receiving an input from the environment, or by receiving a message from one of its hybrid functionalities (or from the adversary). Any activation results in the activated ITI performing some computation on its view of the protocol and its local state, and ends with either the party sending a message to some of its hybrid functionalities, sending an output to the environment, or not sending any message at all. In any of these cases, the party loses the activation.[7] We denote the identities of parties by P_i, i.e. $P_i = (\mathsf{pid}_i, \mathsf{sid}_i)$, and call P_i a party for short. The index i is used to distinguish two identifiers, i.e., $P_i \neq P_j$, and otherwise carries no meaning. We will assume a central adversary $\mathcal{A}$ who gets to corrupt miners and might use them to attempt to break the protocol's security. As is common in (G)UC, the resources available to the parties are described as hybrid functionalities.

Our protocols are synchronous (G)UC protocols [BMTZ17, KMTZ13]: parties have access to a (global) clock setup, denoted by $\mathcal{F}_{\text{CLOCK}}$. and can communicate over a network of authenticated multicast channels.

[7] In the latter case the activation goes to the environment by default.

We adopt the *dynamic availability* model implicit in [BMTZ17] which was fleshed out in [BGK+18]. We next sketch its main components: All functionalities, protocols, and setups have a dynamic party set. I.e., they all include special instructions allowing parties to register and deregister, and allow the adversary to learn the current set of registered parties. Additionally, global setups allow any other setup (or functionality) to register and deregister with them, and also allow other setups to learn their set of registered parties (we refer to Appendix B for the formal treatment).

We conclude this section by elaborating on the main hybrid functionality used in our paper. For self containment we have included formal descriptions of the ideal functionalities we consider in Appendices C and D.

4.1 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 [BMTZ17, BGK+18] in order to describe a transaction ledger and its properties. As proved in [BMTZ17, BGK+18], 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 [GKL15, PSs17]:

- *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 [BMTZ17, BGK+18], 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 Figure 11 in Appendix D and [BMTZ17, BGK+18]. 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 $\mathsf{Validate}$, $\mathsf{ExtendPolicy}$ and $\mathsf{Blockify}$. At a high level, anyone (honest miner or the adversary) may submit a transaction to $\mathcal{F}_{\text{LEDGER}}$. The trasaction is validated by means of a filtering predicate $\mathsf{Validate}$, 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 $\mathsf{Blockify}$, 3) creates a block including the output of $\mathsf{Blockify}$, 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 [GKL15, PSs17] 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 $\mathtt{state}$ as its output.[8]

To enforce transaction liveness and chain-quality, $\mathcal{F}_{\text{LEDGER}}$ relies on the function $\mathsf{ExtendPolicy}$. At a high level, $\mathsf{ExtendPolicy}$ 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, $\mathsf{ExtendPolicy}$ takes the current contents of the buffer, along with the adversary's recommendation $\mathtt{NxtBC}$, and the block-insertion times vector $\tau_{\mathtt{state}}$. The latter is a vector listing the times when each block was inserted into the state. The output of $\mathsf{ExtendPolicy}$ 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 $\mathsf{Blockify}$.

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 Appendix D for a detailed description of the functionality.

- The input $(\text{READ}, \text{sid})$ is used to request the content of the ledger's $\mathtt{state}$. 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) $\mathtt{state}$ to the caller.

- The input $(\text{SUBMIT}, \text{sid}, \mathtt{tx})$ is used to request that a transaction $\mathtt{tx}$ be added to the buffer. That is, upon receiving a $(\text{SUBMIT}, \text{sid}, \mathtt{tx})$ message from any party (or the adversary), the ledger adds the transaction $\mathtt{tx}$ to the buffer $\mathtt{buffer}$. If the validation predicate $\mathsf{Validate}$, on input $\mathtt{state}, \mathtt{buffer}, \mathtt{tx}$ outputs 1, then $\mathtt{tx}$ will be included in $\mathtt{state}$.[9] The time required for the transaction to be part of $\mathtt{state}$ and visible to all honest parties who query $\mathcal{F}_{\text{LEDGER}}$ depends on the transaction liveness parameter defined in $\mathsf{ExtendPolicy}$.

5 Define and Instantiate a New Cryptocurrency from $\mathcal{F}_{\text{ledger}}$

The ledger $\mathcal{F}_{\text{LEDGER}}$ does not itself realize a cryptocurrency. We use $\mathtt{E}$ to denote the symbol of the coins that will be maintaned by our ledger $\mathcal{F}_{\text{T-LEDGER}}$, and show how to realize $\mathcal{F}_{\text{T-LEDGER}}$ from $\mathcal{F}_{\text{LEDGER}}$

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 $\mathtt{E}$, 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

[8] As observed in [BMTZ17], 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 $\mathtt{windowSize}$.

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

that is part of the description of $\mathcal{F}_{\text{T-LEDGER}}$. f takes as input the transaction $\texttt{tx}$ 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 $\texttt{E}$[10]. To do so, $\mathcal{F}_{\text{T-LEDGER}}$ generates an address addr_0 and sends $(\text{SUBMIT}, \text{sid}, \texttt{tx})$ to the wrapped $\mathcal{F}_{\text{LEDGER}}$ with $\texttt{tx} := (V, 0^\lambda, \mathsf{addr}_0, \perp, 0)$, 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 $\mathcal{T}$.

$\mathcal{F}_{\text{T-LEDGER}}$, 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 $\mathsf{Blockify}$ works. $\mathsf{Blockify}$ 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 $\mathsf{ExtendPolicy}$ works, as any realization of $\mathsf{ExtendPolicy}$ can be used in our formalization. We provide a formal description of $\mathcal{F}_{\text{T-LEDGER}}$ in Figure 2.

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 ganerated 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.

[10] It is easy to intialize the functionlity 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.

$\mathcal{F}_{\textbf{T-Ledger}}$

Initialization

1. Parameters: the trapdoor functionality $\mathcal{F}_{\mathsf{trap}}$ and the fee function f.
2. Send $(\textsc{register}, P_0)$ to $\mathcal{A}$.
3. Upon receiving addr_0 from $\mathcal{A}$, if $\mathsf{addr}_0 = 0^\lambda$, then ignore the command and stop, else add (P_0, addr_0) to $\mathcal{T}$.
4. Initialize the functionality $\mathcal{F}_{\textsc{ledger}}$ with a registered party P_0.

Registration

- Upon receiving $(\textsc{register})$ from a party P_i, send $(\textsc{register}, P_i)$ to $\mathcal{A}$. Upon receiving addr_i from $\mathcal{A}$, if there is already an entry $(P_j, \mathsf{addr}_i) \in \mathcal{T}$ for some $P_j \in \mathcal{P}$, then ignore the command, else add (P_i, addr_i) to $\mathcal{T}$), register P_i to $\mathcal{F}_{\textsc{ledger}}$, and send (addr_i) to P_i.

Transactions

- Upon receiving $(\textsc{submit}, \mathrm{sid}, \mathtt{tx})$ from a party P_i, parse $\mathtt{tx}$ as $(v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{aux}, \mathsf{fee})$. If there exists an entry (P_i, addr_i) in $\mathcal{T}$ and $\mathsf{fee} \geq f(\mathtt{tx})$, then continue with the following steps, else ignore the command.
  - Get $\mathtt{state}$ and $\mathtt{buffer}$ from $\mathcal{F}_{\textsc{ledger}}$, initialize $\mathsf{balance} \leftarrow 0$ and for each $\mathtt{tx}^\star$ in $\mathtt{buffer}$ and in $\mathtt{state}$.
    - If $\mathtt{tx}^\star = (v^\star, \mathsf{addr}_i, \mathsf{addr}, \mathsf{aux}^\star, \mathsf{fee}^\star)$, then compute $\mathsf{balance} \leftarrow \mathsf{balance} - v^\star - \mathsf{fee}^\star$.
    - If $\mathtt{tx}^\star = (v^\star, \mathsf{addr}^\star, \mathsf{addr}_i, \mathsf{aux}^\star, \mathsf{fee}^\star)$, then compute $\mathsf{balance} \leftarrow \mathsf{balance} + v^\star$.
  - If $\mathsf{balance} \geq v + f(\mathsf{fee})$, then send $(\textsc{submit}, \mathrm{sid}, (v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{aux}, f(\mathsf{fee})))$ to $\mathcal{F}_{\textsc{ledger}}$ on behalf of P_i.

Trapdoor input Upon receiving $(\textsc{submit-trapdoor}, \mathrm{sid}, \mathtt{tx}, P_i)$ from $\mathcal{F}_{\mathsf{trap}}$, send $(\textsc{submit}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{F}_{\textsc{ledger}}$ on behalf of P_i.

Getting state and other commands

- Upon receiving $(\textsc{read}, \mathrm{sid})$ from P_i, send $(\textsc{read}, \mathrm{sid})$ to $\mathcal{F}_{\textsc{ledger}}$. Upon receiving $(\textsc{read}, \mathrm{sid}, \mathtt{state})$, forward $(\textsc{read}, \mathrm{sid}, \mathtt{state})$ to P_i
- Upon receiving any other input from an honest party $P_i \in \mathcal{P}$ (resp. from $\mathcal{A}$), forward it to $\mathcal{F}_{\textsc{ledger}}$ on behalf of P_i (resp. $\mathcal{A}$). Upon receiving a reply to a command sent on behalf of a party P_i (resp. from $\mathcal{A}$), forward it to P_i (resp. $\mathcal{A}$).

Figure 2: This ledger allows exchanging coins of type `E` between parties.

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.

6 How to handle smart-contracts

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 $\mathcal{F}_{\mathrm{SC}}$ is a small functionality managed by $\mathcal{F}_{\text{TSC-LEDGER}}$ that maintains its own state $\mathtt{cstate}$. The behavior of $\mathcal{F}_{\mathrm{SC}}$ 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 $\mathtt{cstate}$, 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 $\mathtt{cstate}$, 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[11]. If this check is successful, then $\mathcal{F}_{\text{TSC-LEDGER}}$ forwards the transaction and the current ledger state to $\mathcal{F}_{\mathrm{SC}}$, which does the following: It uses f_{CFee} to check whether the fee specified in $\mathtt{tx}$ minus the fee required to query the contract (denoted with Fee) would be sufficient to update the contract state using 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 $\mathtt{cstate}$ by running f_{trans} on input the payload of $\mathtt{tx}$ (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 $\mathtt{tx}$, $\mathsf{Flag}_{\mathsf{C}} \in \{\mathtt{ok}, \mathtt{ko}\}$, 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 $\mathsf{actualfee}$ 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 $\mathsf{actualfee} = \mathsf{fee}$

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

- 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 Figure 3 for the formal description of $\mathcal{F}_{\text{TSC-Ledger}}$ and for the abstraction of $\mathcal{F}_{\text{SC}}$.

$\mathcal{F}_{\textbf{TSC-Ledger}}$

Parameters. Minimum fee Fee for contract calls.
Initialization. Initialize the contract functionality $\mathcal{F}_{\text{SC}}$ with identifier $\mathcal{F}_{\text{SC}}.\mathsf{id}$, and $\mathcal{F}_{\text{T-Ledger}}$ with $\mathcal{F}_{\mathsf{trap}} = \mathcal{F}_{\text{TSC-Ledger}}.\mathsf{id}$.
Registration

- Upon receiving (REGISTER) from a party P_i register P_i to $\mathcal{F}_{\text{T-Ledger}}$ thus obtaining $\mathsf{addr}^{\mathtt{E}}_i$ and send $\mathsf{addr}^{\mathtt{E}}_i$ to P_i.

Transactions

- *(Standard transaction).* Upon receiving $(\text{SUBMIT}, \text{sid}, \mathtt{tx})$ from a party P_i, parse it as $(v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{E}}_j, \mathsf{aux}, \mathsf{fee}, \mathsf{type})$.

  If $\mathsf{type} = \mathtt{E}$, then send $(v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{E}}_j, \perp, \mathsf{fee})$ to $\mathcal{F}_{\text{T-Ledger}}$ on behalf of P_i.
  If $\mathsf{type} = \mathsf{SC}$ and $\mathsf{fee} \geq \mathsf{Fee}$, then:
  - Get the state and the buffer of $\mathcal{F}_{\text{T-Ledger}}$ and check if P_i has at least Fee coins of type $\mathtt{E}$. If this is not the case then reject the command. Otherwise, continue as follows.
  - Define $\mathtt{tx}' := (v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{E}}_j, \mathsf{aux}, \mathsf{fee})$.
  - Send $(\text{SUBMIT}, \text{sid}, P_i, \mathtt{tx}', \mathtt{state})$ to $\mathcal{F}_{\text{SC}}$.
  - Upon receiving $(\mathsf{Flag}_\mathsf{C}, \mathtt{cstate}, \mathsf{actualfee})$ from $\mathcal{F}_{\text{SC}}$, define $\mathtt{tx}^{\mathtt{E}} := (0, \mathsf{addr}^{\mathtt{E}}_i, 0^\lambda, (\mathsf{Flag}_\mathsf{C}, \mathsf{aux}, \mathtt{cstate}, \mathcal{F}_{\text{SC}}.\mathsf{id}), \mathsf{actualfee})$ and send $(\text{SUBMIT-TRAPDOOR}, \text{sid}, \mathtt{tx}^{\mathtt{E}}, P_i)$ to $\mathcal{F}_{\text{T-Ledger}}$.

Getting states

- Upon receiving $(\text{READ}, \text{sid}, \mathsf{type})$ from P_i forward the command $(\text{READ}, \text{sid})$ to $\mathcal{F}_{\text{T-Ledger}}$ on behalf of P_i.
- Upon receiving $\mathtt{state}$ from $\mathcal{F}_{\text{T-Ledger}}$, if $\mathsf{type} = \mathtt{E}$ then:
  - Initialize an empty list $\mathtt{state}^{\mathtt{E}}$.
  - For each $\mathtt{tx} \in \mathtt{state}$ such that $\mathtt{tx} = (v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{E}}_j, \perp, \mathsf{fee})$, add $\mathtt{tx}$ to $\mathtt{state}^{\mathtt{E}}$.
  - Return $\mathtt{state}^{\mathtt{E}}$.

  If $\mathsf{type} = \mathsf{SC}$, then send $(\text{FILTER}, \text{sid}, \mathtt{state})$ to $\mathcal{F}_{\text{SC}}$, and send to P_i what $\mathcal{F}_{\text{SC}}$ returns.

$\mathcal{F}_{\text{SC}}$ abstraction

- $\mathcal{F}_{\text{SC}}$ is initialized with the fee function f_{CFee}, the state-transition function f_{trans}, the filtering function f_{filter}, and an initial contract state $\mathtt{cstate}$.
- Upon receiving $(\text{SUBMIT}, \text{sid}, P_i, \mathtt{tx}', \mathtt{state})$:
  - Parse $\mathtt{tx}'$ as $(v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{E}}_j, \mathsf{aux}, \mathsf{fee})$.
  - Check if $(\mathsf{fee} - \mathsf{Fee})$ is sufficient to run the contract, computing $\mathsf{fee}^{\mathsf{SC}} \leftarrow f_{\mathsf{CFee}}(\mathtt{cstate}, \mathtt{state}, \mathsf{aux}, \mathsf{fee} - \mathsf{Fee})$ (i.e. $\mathsf{fee}^{\mathsf{SC}}$ represents the actual fee required to run the contract or $\perp$ if fee is not sufficient to update the contract's state).
  - If $\mathsf{fee}^{\mathsf{SC}} = \perp$, then return $(\mathtt{ko}, \mathtt{cstate}, \mathsf{fee})$.
  - Otherwise, compute $\mathtt{cstate} \leftarrow f_{\mathsf{trans}}(\mathsf{aux}, \mathtt{state}, \mathtt{cstate})$ and return $(\mathtt{ok}, \mathtt{cstate}, \mathsf{fee}^{\mathsf{SC}} + \mathsf{Fee})$
- Upon receiving $(\text{FILTER}, \text{sid}, \mathtt{state})$, return $f_{\mathsf{filter}}(\mathtt{state}, \mathtt{cstate})$.

Figure 3: This ledger tolerates any type of contract abstracted by $\mathcal{F}_{\text{SC}}$.

$\mathcal{F}_{\mathbf{SC}}^{\mathrm{T}}$ functions

Initialization. The contract is parametrized by the functions $f^{\mathsf{T}}_{\mathsf{CFee}}$, f_{trans}, and $f^{\mathsf{T}}_{\mathsf{filter}}$ described below. $\texttt{cstate}$ consists of the following components:

- Constants: $y, \mathsf{addr}^{\mathsf{T}}_0$, Fee, identifier $\mathcal{F}^{\mathrm{T}}_{\mathrm{SC}}.\mathsf{id}$.
- Empty set $\texttt{token-set}$.

Functions and helper procedures

$f_{\mathsf{trans}}(\mathsf{aux}^{\mathsf{E}}, \texttt{state}, \texttt{cstate})$

- Add ,$\mathsf{aux}^{\mathsf{E}}$ to $\texttt{token-set}$ of $\texttt{cstate}$ and return the updated $\texttt{cstate}$.

$f_{\mathsf{CFee}}(\texttt{cstate}, \texttt{state}, \texttt{tx}, \mathsf{fee})$

- Define and initialize $\mathsf{tfee} \leftarrow 0$.
- Parse $\texttt{tx} := (0, \mathsf{addr}^{\mathsf{E}}_i, 0^\lambda, \mathsf{aux}^{\mathsf{E}}, \mathsf{fee})$.
- Parse $\texttt{tx}^{\mathsf{T}}$ as $(\mathsf{v}, \mathsf{addr}^{\mathsf{T}}_i, \mathsf{addrs}, \mathsf{id})$ and compute $\mathsf{tfee} \leftarrow \mathsf{tfee} + \mathsf{Fee}|\mathsf{v}|$.
- If $\mathsf{tfee} > \mathsf{fee}$, then return $\perp$; otherwise, return tfee.

$f_{\mathsf{filter}}(\texttt{cstate}, \texttt{state})$

- Initialize the empty list $\texttt{state}^{\mathsf{T}}$ and set $\texttt{temp-buffer} \leftarrow \texttt{token-set}$.
- For each $\texttt{tx}^{\mathsf{E}} = (0^\lambda, \mathsf{addr}^{\mathsf{E}}_i, 0^\lambda, \mathsf{aux}^{\mathsf{E}}, \mathsf{fee}^{\mathsf{E}})$ in $\texttt{state}$ where $\mathsf{aux}^{\mathsf{E}} = (\texttt{ok}, (\mathsf{v}^\star, \mathsf{addr}^\star_i, \mathsf{addrs}^\star, \mathsf{id}^\star_i), \texttt{cstate}^\star, \mathcal{F}^{\mathrm{T}}_{\mathrm{SC}}.\mathsf{id})$:
  - Define $\mathsf{aux} := (\mathsf{v}^\star, \mathsf{addr}^\star_i, \mathsf{addrs}^\star, \mathsf{id}^\star_i)$.
  - If $\mathsf{verify}(\mathsf{aux}, \texttt{state}^{\mathsf{T}}) = 1$ and $\mathsf{aux} \in \texttt{temp-buffer}$, then add aux to $\texttt{state}^{\mathsf{T}}$ and remove aux from $\texttt{temp-buffer}$.
- Return $(\textsc{read}, \mathrm{sid}, \texttt{state})$.

$\mathsf{verify}(\mathsf{aux}^{\mathsf{E}}, \texttt{state}^{\mathsf{T}})$

- Parse $\mathsf{aux}^{\mathsf{E}}$ as $(\mathsf{v}^{\mathsf{T}}, \mathsf{addr}^{\mathsf{T}}_i, \mathsf{addrs}, \mathsf{id}^{\mathsf{T}}_i)$
- If $\mathsf{addr}^{\mathsf{T}}_0 = \mathsf{addr}^{\mathsf{T}}_i$, then initialize $\mathsf{balance} \leftarrow y$; otherwise, $\mathsf{balance} \leftarrow 0$.
- For each $\texttt{tx}^\star = (\mathsf{v}^\star, \mathsf{addr}^\star_i, \mathsf{addrs}^\star, \mathsf{id}^\star_i)$ in $\texttt{state}^{\mathsf{T}}$:
  - If $\mathsf{addr}^\star = \mathsf{addr}^{\mathsf{T}}_i$, then compute $\mathsf{balance} \leftarrow -\sum_k \mathsf{v}^\star[k]$
  - For each k such that $\mathsf{addrs}^\star[k] = \mathsf{addr}^{\mathsf{T}}_i$, compute $\mathsf{balance} \leftarrow +v^\star[k]$.
- If $\sum_k \mathsf{v}^{\mathsf{T}}[k] \geq \mathsf{balance}$ then return 1 else return 0.

Figure 4: Smart contract for the creation of a new token. The fee required to run the contract is computed by multiplying the number of token transactions encoded in the payload of the input $\texttt{tx}$ and Fee.

7 Our ideal functionality $\mathcal{F}^{\mathsf{EET}}_{\mathbf{Ledger}}$

In this section we can finally define the functionality $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$. $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ internally runs $\mathcal{F}_{\text{TSC-}\textsc{Ledger}}$, parametrized by a contract $\mathcal{F}^{\mathrm{T}}_{\mathrm{SC}}$. $\mathcal{F}^{\mathrm{T}}_{\mathrm{SC}}$ maintains a token $\texttt{T}$, and allows parties to issue transactions with respect to such a token. At a high level, any party that has some tokens can sent it to another party by querying the contract $\mathcal{F}^{\mathrm{T}}_{\mathrm{SC}}$. However, invoking the contract requires payment of a fee in the native currency $\texttt{E}$, 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 $\mathsf{del\text{-}fee}$ tokens $\texttt{T}$ to a special party M, called the intermediary, and M will pay the fee in $\texttt{E}$ on the behalf of the sender ($\mathsf{del\text{-}fee}$ 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 $\mathsf{del\text{-}fee}$, or nothing happens. We propose the formal description of $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ in Figure 5 and the description of

$\mathcal{F}^{\mathsf{EET}}_{\textbf{Ledger}}$

Initialization

- Initialize an empy set $\mathcal{T}^{\mathtt{T}}$.
- Send $(\textsc{register}, P_0)$ to $\mathcal{A}$.
- Upon receiving $\mathsf{addr}^{\mathtt{T}}_0$ from $\mathcal{A}$, add $(P_0, \mathsf{addr}^{\mathtt{T}}_0)$ to $\mathcal{T}^{\mathtt{T}}$, run the initialization procedure of $\mathcal{F}^{\mathtt{T}}_{\mathrm{SC}}$, and initialize the wrapped functionality $\mathcal{F}_{\text{TSC-Ledger}}$ with the contract $\mathcal{F}^{\mathtt{T}}_{\mathrm{SC}}$, using identifier $\mathcal{F}^{\mathtt{T}}_{\mathrm{SC}}.\mathsf{id}$
- Send $(\textsc{register}, \mathsf{M})$ to $\mathcal{A}$.
- Upon receiving $\mathsf{addr}^{\mathtt{T}}_{\mathsf{M}}$ from $\mathcal{A}$, add $(\mathsf{M}, \mathsf{addr}^{\mathtt{T}}_{\mathsf{M}})$ to $\mathcal{T}^{\mathtt{T}}$.

Registration

- Upon receiving $(\textsc{register})$ from a party P_i, send $(\textsc{register}, P_i)$ to $\mathcal{A}$.
- Upon receiving $\mathsf{addr}^{\mathtt{T}}_i$ from $\mathcal{A}$, if there is already an entry $(P_j, \mathsf{addr}^{\mathtt{T}}_i) \in \mathcal{T}^{\mathtt{T}}$ for some $P_j \in \mathcal{P}$, then ignore the command; otherwise, add $(P_i, \mathsf{addr}^{\mathtt{T}}_i)$ to $\mathcal{T}^{\mathtt{T}}$ and register P_i to $\mathcal{F}_{\text{TSC-Ledger}}$, thus obtaining $\mathsf{addr}^{\mathtt{E}}_i$, and send $(\mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{T}}_i)$ to P_i.

Transactions

- *(Standard transaction). Upon receiving* $(\textsc{submit}, sid, \boldsymbol{tx}^{\mathtt{T}})$ from a party P_i, parse $\mathtt{tx}^{\mathtt{T}}$ as $(v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{E}}_j, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{fee}, \mathsf{Coin})$.

  If $\mathsf{Coin} = \mathtt{T}$, and there exists an entry $(P_i, \mathsf{addr}^{\mathtt{T}}_i)$ in $\mathcal{T}^{\mathtt{T}}$, and $\mathsf{fee} \geq 2\mathsf{Fee}$, then do the following :

  – Send $(\textsc{req-trx}, P_i, \mathtt{tx}^{\mathtt{T}})$ to $\mathcal{A}$ and, upon receiving id_i, define $\mathsf{aux} := (v, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{id}_i)$ and $\mathtt{tx} = (0, \mathsf{addr}^{\mathtt{E}}_i, 0^{\lambda}, \mathsf{aux}, \mathsf{fee}, \mathsf{SC})$.

  If $\mathsf{Coin} = \mathtt{E}$ and $\mathsf{fee} \geq \mathsf{Fee}$, then:

  Define $\mathtt{tx} := (v, \mathsf{addr}^{\mathtt{E}}_i, \mathsf{addr}^{\mathtt{E}}_j, \perp, \mathsf{fee}, \mathtt{E})$
- Send $(\textsc{submit}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{F}_{\text{TSC-Ledger}}$ on the behalf of P_i.
- *(Delegatable transaction).* Upon receiving $(\textsc{submit-delegation}, \mathrm{sid}, \mathtt{tx}^{\mathtt{T}})$ from a party P_i, parse $\mathtt{tx}^{\mathtt{T}}$ as $(v, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{fee}^{\mathtt{T}})$. If there exists an entry $(P_i, \mathsf{addr}^{\mathtt{T}}_i)$ in $\mathcal{T}^{\mathtt{T}}$ and $\mathsf{fee}^{\mathtt{T}} \geq \mathsf{del\text{-}fee}$, then do the following, ignoring the command otherwise:

  – Send $(\textsc{req-trx-del}, P_i, \mathtt{tx}^{\mathtt{T}})$ to $\mathcal{A}$ and, upon receiving id_i, define $\mathsf{aux} := ([v, \mathsf{fee}^{\mathtt{T}}], \mathsf{addr}^{\mathtt{T}}_i, [\mathsf{addr}^{\mathtt{T}}_j, \mathsf{addr}^{\mathtt{T}}_{\mathsf{M}}], \mathsf{id}_i)$.

  – If M is honest, then define $\mathtt{tx} := (0, \mathsf{addr}_{\mathsf{M}}, 0^{\lambda}, \mathsf{aux}, 3\mathsf{Fee}, \mathsf{SC})$ and send $\mathtt{tx}$ to $\mathcal{F}_{\text{TSC-Ledger}}$ on behalf of M. If M is corrupted, do the following:

    - Send $(\textsc{delegate}, \mathsf{aux}, P_i)$ to M.
    - If M replies with $(\textsc{reject}, P_i)$, then send $\textsc{reject}$ to P_i. If M replies with $(\textsc{accept}, P_i, \mathsf{fee})$, then define $\mathtt{tx} := (0, \mathsf{addr}_{\mathsf{M}}, 0^{\lambda}, \mathsf{aux}, \mathsf{fee})$ and send $\mathtt{tx}$ to $\mathcal{F}_{\text{TSC-Ledger}}$ on behalf of M.

Getting states

- Upon receiving $(\textsc{read}, \mathrm{sid}, \mathsf{Coin})$ from P_i, forward the command to $\mathcal{F}_{\text{TSC-Ledger}}$ on behalf of P_i.
- Upon receiving $(\textsc{read}, \mathrm{sid}, \mathtt{state})$, forward it to P_i.

Forwarding queries to $\mathcal{F}_{\textbf{TSC-Ledger}}$.

- Upon receiving $(\textsc{inner-input}, \mathrm{sid}, m, P_i)$ from $\mathcal{A}$, if P_i is an honest party, then ignore the command. Otherwise, if P_i is corrupted, send m to $\mathcal{F}_{\text{TSC-Ledger}}$ on behalf of P_i.
- Upon receiving any other input from an honest party $P_i \in \mathcal{P}$ (resp. from $\mathcal{A}$), forward it to $\mathcal{F}_{\text{TSC-Ledger}}$ on behalf of P_i.
- Upon receiving a reply to a command sent on behalf of a party $P_i \in \mathcal{P}$ (resp. from $\mathcal{A}$), forward it to P_i (resp. $\mathcal{A}$).

Figure 5: This ledger allows parties with no coins of type $\mathtt{E}$ to post transactions using tokens of type $\mathtt{T}$ (we call this transaction a delegated transaction). In the case where M is honest and has enough coins of type $\mathtt{E}$ to pay the fee, the delegated transactions are always included in the ledger state.

$\mathcal{F}^{\mathtt{T}}_{\mathrm{SC}}$ in Figure 4, and provide a high level description of those below.

The functionality $\mathcal{F}^{\mathsf{EET}}_{\text{Ledger}}$, 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[12]. We denote the token wallet addresses of P_0 and M with $\mathsf{addr}_0^{\mathsf{T}}$ and $\mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}$ respectively. Any time $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ receives a registration command from a party P_i, it registers P_i to the ledger $\mathcal{F}_{\text{TSC-}\textsc{Ledger}}$, thus obtaining $\mathsf{addr}_i^{\mathtt{E}}$. It then generates a token wallet address $\mathsf{addr}_i^{\mathsf{T}}$ and returns $(\mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathsf{T}})$ to P_i. $(\mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathsf{T}})$ represents respectively the wallet addresses for the native currency $\mathtt{E}$ and for the token $\mathtt{T}$. $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ tolerates two types of transactions: *standard* and *delegated* transactions. Any registered party P_i can issue a standard transaction $\mathtt{tx}^{\mathsf{T}} := (v, \mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}}, \mathsf{fee})$, where v denotes the amount of tokens, $(\mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_i^{\mathsf{T}})$ are the addresses of the sender, $\mathsf{addr}_j^{\mathsf{T}}$ is the token wallet address of the receiver, and fee is the fee expressed in coins of type $\mathtt{E}$.

$\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ takes $\mathtt{tx}^{\mathsf{T}}$ and creates a transaction $\mathtt{tx}^{\mathtt{E}}$ for the ledger $\mathcal{F}_{\text{TSC-}\textsc{Ledger}}$ that 1) has as a sender address $\mathsf{addr}_i^{\mathtt{E}}$, 2) has a fee fee, and 3) calls the contract $\mathcal{F}_{\text{SC}}^{\mathsf{T}}$ and includes in its payload what we call a *token transaction* $\mathtt{tx}' := (v, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}})$.[13] $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ then forwards $\mathtt{tx}^{\mathtt{E}}$ to the ledger $\mathcal{F}_{\text{TSC-}\textsc{Ledger}}$ on behalf of P_i.

The contract $\mathcal{F}_{\text{SC}}^{\mathsf{T}}$ 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 $(\mathsf{ok}, \mathsf{cstate}, \mathsf{actualfee})$. Note that this means that the $\mathtt{tx}'$ is part of the contract state and appears in the $\mathcal{F}_{\text{TSC-}\textsc{Ledger}}$'s state by definition.

To complete this first part of the description of $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$, it remains to specify the function f_{filter} (and f_{CFee}, which we describe later in this section) of $\mathcal{F}_{\text{SC}}^{\mathsf{T}}$. f_{filter} receives as input the contract state and the state of $\mathcal{F}_{\text{TSC-}\textsc{Ledger}}$ (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 $\mathsf{aux}^{\mathtt{E}} = (\mathtt{ok}, \mathtt{tx}', \mathtt{cstate}^{\star}, \mathcal{F}_{\text{SC}}^{\mathsf{T}}.\mathsf{id})$), adds $\mathtt{tx}'$ to $\mathtt{state}^{\mathsf{T}}$ if and only if:

1. $\mathtt{tx}'$ appears in $\mathtt{token\text{-}set}$ (which is part of the token state).

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

$\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ 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, $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ admits what we call *delegated transactions*. A party that wants to issue a delegated transaction submits $\mathtt{tx}^{\mathsf{T}} := (v, \mathsf{addr}_i^{\mathsf{T}}, \mathsf{addr}_j^{\mathsf{T}}, \mathsf{fee}^{\mathsf{T}})$ to $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$, 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 $\mathsf{fee}^{\mathsf{T}} \geq \mathsf{del\text{-}fee}$, (where we recall that $\mathsf{del\text{-}fee}$ is the minimum fee required for the delegation to be considered,) then $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ submits a call to the contract $\mathcal{F}_{\text{SC}}^{\mathsf{T}}$ on behalf of M with the input (the payload of the transaction) $\mathsf{aux} := (([v, \mathsf{fee}^{\mathsf{T}}], \mathsf{addr}_i^{\mathsf{T}}, [\mathsf{addr}_j^{\mathsf{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathsf{T}}]))$. If M has enough coins of type $\mathtt{E}$ to afford the call to $\mathcal{F}_{\text{SC}}^{\mathsf{T}}$, then aux will become part of the contract state. To accommodate for this special input, we modify the filtering function f_{filter} of $\mathcal{F}_{\text{SC}}^{\mathsf{T}}$ 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

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

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

address $\mathsf{addr}^{\mathtt{T}}_j$, and the second moves $\mathsf{fee}^{\mathtt{T}}$ from the wallet address $\mathsf{addr}^{\mathtt{T}}_i$ to the wallet address $\mathsf{addr}^{\mathtt{T}}_{\mathsf{M}}$. 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 $\mathtt{aux}$ (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. We use this mechanism only to simplify the description of the functionality, and later the security proof. In the Section 9, we provide experimental results to estimate the cost of executing these transactions in a real world realization of $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$.

8 Our Protocol: how to realize $\mathcal{F}^{\mathsf{EET}}_{\mathbf{Ledger}}$

Our protocol is described in the $\mathcal{F}_{\textsc{T-Ledger}}$-hybrid world, where $\mathcal{F}_{\textsc{T-Ledger}}$ is parametrized by $\mathcal{F}_{\mathsf{trap}} = \bot$, and the fee function f which, upon receiving an input transaction $\mathtt{tx}^{\mathtt{E}}$, does the following:

- Parse $\mathtt{tx}$ as $(v, \mathsf{addr}_i, \mathsf{addr}_j, \mathsf{aux}, \mathsf{fee})$.
- If $\mathsf{aux} = \bot$, then return Fee.
- Otherwise, return $\mathsf{Fee} + |\mathsf{aux}|/\kappa\mathsf{Fee}$.

In a nutshell, the fee required for a transaction to settle in the $\mathcal{F}_{\textsc{T-Ledger}}$'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 Figure 6. At a very high level, the protocol works as follows: Each party registers with $\mathcal{F}_{\textsc{T-Ledger}}$ and runs $\mathtt{Kgen}(1^\lambda)$ to obtain $(\mathsf{sk}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_i)$, where $\mathsf{addr}^{\mathtt{T}}_i$ 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 $\mathcal{F}_{\textsc{T-Ledger}}$ that contains in its payload $\mathsf{aux} := (v, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{id}, \sigma^{\mathtt{T}}_i)$, where id is a random value, and $\sigma^{\mathtt{T}}_i$ is a signature of $(v, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{id})$ that verifies under the verification key $\mathsf{addr}^{\mathtt{T}}_i$. 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 $(\textsc{read}, \mathrm{sid}, \mathtt{T})$, they shall retrieve $\mathcal{F}_{\textsc{T-Ledger}}$'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 $(v, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{id}, \sigma^{\mathtt{T}}_i)$ is valid if $\mathsf{addr}^{\mathtt{T}}_i$ has received at least v tokens, $\sigma^{\mathtt{T}}_i$ is a signature of $(v, \mathsf{addr}^{\mathtt{T}}_i, \mathsf{addr}^{\mathtt{T}}_j, \mathsf{id})$ that verifies under the verification key $\mathsf{addr}^{\mathtt{T}}_i$, 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 $m := ([v, \mathsf{del\text{-}fee}], \mathsf{addr}^{\mathtt{T}}_i, [\mathsf{addr}^{\mathtt{T}}_j, \mathsf{addr}^{\mathtt{T}}_{\mathsf{M}}], \mathsf{id})$ and signs it, thus obtaining $\sigma^{\mathtt{T}}_i$. P_i then sends $(m, \sigma^{\mathtt{T}}_i)$ to M. The honest M then creates a transaction for $\mathcal{F}_{\textsc{T-Ledger}}$ that includes $(m, \sigma^{\mathtt{T}}_i)$ 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

Protocol Π^{Token}

Initialization

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

Registration

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

Transactions

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

    Define $\mathtt{tx} := (v, \mathsf{addr}_i, \mathsf{addr}_j, \perp, \mathsf{fee})$.
  - Send $I = (\textsc{submit}, \text{sid}, \mathtt{tx})$ to $\mathcal{F}_{\textsc{T-Ledger}}$.

Getting states

- Upon receiving $(\textsc{read}, \text{sid}, \mathsf{type})$, P forwards the command $(\textsc{read}, \text{sid})$ to $\mathcal{F}_{\textsc{T-Ledger}}$.
- Upon receiving $\mathtt{state}$ from $\mathcal{F}_{\textsc{T-Ledger}}$, 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 $\mathtt{tx} = (v, \mathsf{addr}_i^{\mathtt{E}}, \mathsf{addr}_j^{\mathtt{E}}, \perp, \mathsf{fee})$, add $\mathtt{tx}$ to $\mathtt{state}^{\mathtt{E}}$.
  - Return $(\textsc{read}, \text{sid}, \mathtt{state}^{\mathtt{E}})$.

  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 $(\textsc{read}, \text{sid}, \mathtt{state}^{\mathtt{T}})$.

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

- 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.

Figure 6: Our protocol.

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.

Theorem 8.1. *The protocol Π^{Token} realizes $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ in the $\mathcal{F}_{\text{T-LEDGER}}$-hybrid model.*

Proof. In our proof we consider the more involved case where M is colluding with an arbitrary set of parties $S \subset \mathcal{P}$ and P_0 is honest; in any other case the proof follow from similar arguments. The simulator $\mathsf{Sim}_{\mathsf{M}}$ internally runs the corrupted parties, emulating for them the functionality $\mathcal{F}_{\text{T-LEDGER}}$.

Moreover, $\mathsf{Sim}_{\mathsf{M}}$ reacts as follows on the inputs it receives:

1. Upon receiving $(\text{REGISTER}, P_i)$ from $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, compute $(\mathsf{sk}_i^{\mathtt{T}}, \mathsf{addr}_i^{\mathtt{T}}) \xleftarrow{\$} \mathtt{Kgen}(1^\lambda)$ and send $(P_i, \mathsf{addr}_i^{\mathtt{T}})$ to $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$.

2. Upon receiving $(\text{DELEGATE}, \mathsf{aux}, P_i)$ from $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, compute $\sigma_i^{\mathtt{T}} \xleftarrow{\$} \mathtt{Sign}(\mathsf{sk}_i^{\mathtt{T}}, \mathsf{aux})$ and send $(\mathsf{delegate}, \mathsf{aux}, \sigma_i^{\mathtt{T}})$ to M.

3. Upon receiving $(\text{REQ-TRX-DEL}, P_i, \mathtt{tx}^{\mathtt{T}})$ from $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, sample $\mathsf{id}_i \xleftarrow{\$} \{0,1\}^\lambda$ and send id_i to $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$.

4. Upon receiving $(\text{REQ-TRX}, P_i, \mathtt{tx}^{\mathtt{T}})$ from $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$, parse $\mathtt{tx}^{\mathtt{T}}$ as $(v, \mathsf{addr}_i, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{fee}, \mathsf{Coin})$ and do the following:
   - If $\mathsf{Coin} = \mathtt{T}$ and $\mathsf{fee} \geq 2\mathsf{Fee}$:
     - Sample $\mathsf{id} \xleftarrow{\$} \{0,1\}^\lambda$, define $m := ((v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}), \mathsf{id})$, and compute $\sigma_i^{\mathtt{T}} \leftarrow \mathtt{Sign}(\mathsf{sk}, m)$.
     - Define $\mathtt{tx} = (0, \mathsf{addr}_i, 0^\lambda, (m, \sigma_i^{\mathtt{T}}), \mathsf{fee})$.
   - If $\mathsf{Coin} = \mathtt{E}$ and $\mathsf{fee} \geq \mathsf{Fee}$, define $\mathtt{tx} := (v, \mathsf{addr}_i, \mathsf{addr}_j, \bot, \mathsf{fee})$.
   - Send $I = (\text{SUBMIT}, \text{sid}, \mathtt{tx})$ to $\mathcal{F}_{\text{T-LEDGER}}$ and return id to $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$.

5. If $\mathcal{F}_{\text{T-LEDGER}}$ (emulated in the real-world) receives the input $(\text{SUBMIT}, \text{sid}, \mathtt{tx}_{\mathsf{M}})$ from M, and all of the following are true:
   - $\mathtt{tx}_{\mathsf{M}} = (0, \mathsf{addr}_{\mathsf{M}}, 0^\lambda, (m, \sigma_i^{\mathtt{T}}), \mathsf{fee})$, where $m := (([v, \mathsf{fee}^{\mathtt{T}}], \mathsf{addr}_i^{\mathtt{T}}, [\mathsf{addr}_j^{\mathtt{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}], \mathsf{id})$,
   - An honest party P_i has queried $\mathcal{F}_{\text{LEDGER}}^{\mathsf{EET}}$ with $(\text{SUBMIT-DELEGATION}, \text{sid}, \mathtt{tx}^{\mathtt{T}})$, where $\mathtt{tx}^{\mathtt{T}} = (v, \mathsf{addr}_i^{\mathtt{T}}, \mathsf{addr}_j^{\mathtt{T}}, \mathsf{del\text{-}fee})$ and $\mathtt{Ver}(\mathsf{addr}_i^{\mathtt{T}}, ([v, \mathsf{fee}^{\mathtt{T}}], \mathsf{addr}_i^{\mathtt{T}}, [\mathsf{addr}_j^{\mathtt{T}}, \mathsf{addr}_{\mathsf{M}}^{\mathtt{T}}], \mathsf{id}), \sigma_i^{\mathtt{T}})$,

- $(\textsc{accept}, P_i, \mathsf{fee})$ has not been sent yet to $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$,

then send $(\textsc{accept}, P_i, \mathsf{fee})$ to $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$.

6. Upon receiving any other command m on the adversarial interface of $\mathcal{F}_{\text{T-}\textsc{Ledger}}$, $\mathsf{Sim}_{\mathsf{M}}$ forwards it to $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$.

We now argue that if the real and the ideal world are distinguishable, then there exists an adversary that breaks either the security of the signature scheme or the security of $\mathcal{F}_{\text{T-}\textsc{Ledger}}$. There are few cases that could make the simulator to fail, thus allowing the environment to distinguish the two worlds. We now summarize these cases and argue that none of them occur with more than negligible probability.

- In step 4, the simulator does not have the secret key $\mathsf{sk}^{\mathsf{T}}_i$ to generate the signature σ^{T}_i. We recall that $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ forwards the command $\textsc{req-trx}$ to $\mathcal{A}$ only if a party P_i is querying $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ with either $(\textsc{submit-delegation}, \mathrm{sid}, \mathtt{tx}^{\mathsf{T}})$ or $(\textsc{submit}, \mathrm{sid}, \mathtt{tx}_{\mathsf{M}})$, with $(P_i, \mathsf{addr}_i) \in \mathcal{T}^{\mathsf{T}}$.

  If that is not the case, (implying that the calling party is malicious and has generated his own address,) then we do not have the secret key. However, we do not need it, as $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ would ignore the message for the reason above.

- The simulator computes an address addr (upon receiving the command $(\textsc{register}, P_i)$), and there already exists an entry (P_j, addr) in $\mathcal{T}^{\mathsf{T}}$. If this happens with non-negligible probability, then we can construct a reduction to the signature scheme. The reduction takes as input a signature verification key addr for Σ, and computes $(s, v) \overset{\$}{\leftarrow} \mathtt{Kgen}(1^{\lambda})$. If $\mathsf{addr} \neq v$, the reduction aborts, and otherwise computes and outputs $\mathtt{Sign}(s, 0^{\lambda})$.

  Since, by contradiction, $\mathsf{addr} = v$ with non-negligible probability, and the reduction has never queried the signing oracle, then we can claim that we have broken the security of Σ.

- Let $\mathtt{state}^{\mathsf{ideal}}$ be the output obtained when an honest party P_i queries $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$ with the input $(\textsc{read}, \mathrm{sid}, \mathsf{SC})$. Let $\mathtt{state}^{\mathsf{real}}$ be the output that an honest P_i would obtain when running Π^{Token} with the same input. The simulation fails if $\mathtt{state}^{\mathsf{real}}$ contains $\mathtt{tx} = (v, \mathsf{addr}^{\mathsf{T}}_i, \mathsf{addr}^{\mathsf{T}}_j, \mathsf{id}^{\mathsf{T}})$ and $\mathtt{state}^{\mathsf{ideal}}$ does not. Note that if $\mathtt{tx} = (v, \mathsf{addr}^{\mathsf{T}}_i, \mathsf{addr}^{\mathsf{T}}_j, \mathsf{id}^{\mathsf{T}})$ appears in $\mathtt{state}^{\mathsf{real}}$, it means that the state of $\mathcal{F}_{\text{T-}\textsc{Ledger}}$ contains a transaction whose payload contains $(\mathtt{tx}, \sigma^{\mathsf{T}}_i)$ such that $\mathtt{Ver}(\mathsf{addr}^{\mathsf{T}}_i, \mathtt{tx}, \sigma^{\mathsf{T}}_i) = 1$.

  This can happen in the following two scenarios:

  - The simulator has never signed $\mathtt{tx}$. In this case, we make a reduction to the security of Σ, since the reduction would not need to query the signing oracle with the input $\mathtt{tx}$.
  - The simulator has signed $\mathtt{tx}$. This means that $\mathtt{tx}$ was submitted by P_i through $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$'s interface. By construction of $\mathcal{F}_{\text{T-}\textsc{Ledger}}$ and $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$, if a transaction is accepted by $\mathcal{F}^{\mathsf{EET}}_{\textsc{Ledger}}$, it must be accepted by $\mathcal{F}_{\text{T-}\textsc{Ledger}}$ as well. If this is not the case, then we can make a reduction to the security of $\mathcal{F}_{\text{T-}\textsc{Ledger}}$.

  The opposite scenario could also occur; that is, $\mathtt{state}^{\mathsf{ideal}}$ contains $\mathtt{tx} = (v, \mathsf{addr}^{\mathsf{T}}_i, \mathsf{addr}^{\mathsf{T}}_j, \mathsf{id}^{\mathsf{T}})$ and $\mathtt{state}^{\mathsf{ideal}}$ does not. This can happen in the case where the simulator has already sampled the identifier id^{T} for a previous transaction. However, the probability that this event occurs is negligible. As before, the only other reason we might have such a situation is if the security of $\mathcal{F}_{\text{T-LEDGER}}$ has been compromised. In this case, we could again make a reduction to the security of $\mathcal{F}_{\text{T-LEDGER}}$.

□

9 Implementation, Benchmarks, and Comparisons

We implemented our EET via an Ethereum smart contract, measured its gas consumption, and compared it with other approaches. Our EET conforms to the ERC-20 standard. In its testing mode, our contract has the added functionality of allowing unlimited minting of new tokens by any account. This feature is included for ease of testing rather than for actual use, and should be disabled when the EET is deployed in production. We wrote our contracts in Solidity 0.6.10, and tested them using Hardhat 2.0.8, a Javascript and TypeScript framework for Ethereum smart contract development and testing. We tested only on a locally-running test network, not on any live public network; however, since the Hardhat test node is a faithful implementation of the Ethereum protocol and virtual machine, this should not affect the amount of gas used on any given contract invocation.

To compare the gas useage of our EET with that of the Gas Station Network (GSN), we deployed the GSN infrastructure contracts (most notably the `RelayHub` contract) to our local test network and ran a local relay server. The OpenGSN project provides a testing infrastructure that automatically deploys the required contracts and runs a local relay server; our tests used version 2.1.0 of the OpenGSN repository. The experiment code itself is written in TypeScript, using the Ethers 5.0.26 library for blockchain and contract interaction.

For each of our evaluation and comparison experiments below, we first select sender, receiver, and other relevant addresses randomly (without replacement) from a pool of 20 addresses. After executing the relevant transaction, we record the amount of gas consumed by the transaction. For validation purposes, we also record the Ether and token balances of each address before and after the transaction, to ensure that the correct amounts are transferred. Each experiment was run 1000 times, selecting a new set of addresses for each run.

9.1 EET vs. GSN vs. Standard ERC-20 token

For our first comparison, we ran the following three experiments:

- **Self-funded token transactions:** These experiments test the gas usage of typical, non-delegated use of our EET contract, i.e. by a user that does not want to interact with the delegation server (denoted M in our formalization) and has his own sufficiently funded Ethereum address. The sending address transfers some amount of tokens to the receiving address, submitting the transaction themselves and using their own Ether to pay the transaction fee. In this case, the relevant addresses are the sender and the receiver.

- **Delegated token transactions:** These experiments test the gas usage of our EET delegation mechanism. The sending address transfers some amount of tokens to the receiving address, but a third delegate address (M, which may be another user, another address owned by the same user, or a dedicated server,) submits the transaction and pays the ether fee, automatically (by contract conversion and execution) receiving an equivalent amount of tokens from the sending address in the process. In this case, the relevant addresses are the sender, the receiver, and the delegate.

- **GSN token transactions** These experiments test the gas usage of delegation through the GSN. The sending address transfers some amount of tokens to the receiving address, but delegates to the locally-running relay server, which submits the transaction and pays the ether fee, receiving a repayment of ether from the token contract (indirectly, from the token contract's deposit with the `RelayHub` contract). The `EETPaymaster` contract then extracts an equivalent token fee from the sender. In this case, the relevant addresses are the sender, the receiver, and the relay server address. However, we only have control over the sender and receiver addresses; the relay server's address is determined by the GSN testing infrastructure and cannot be easily changed.

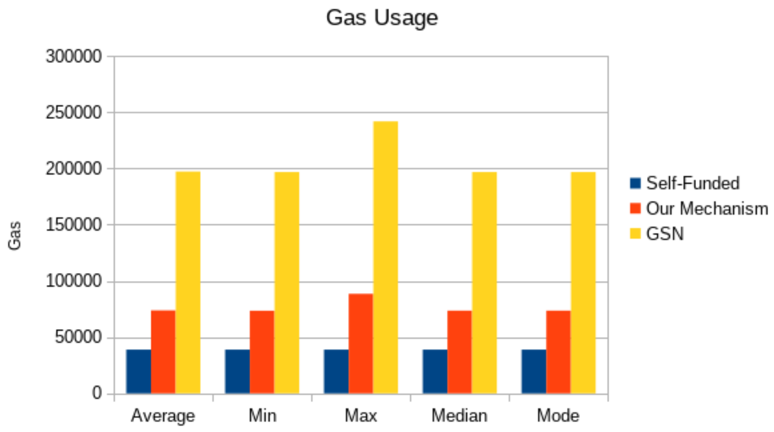

Figure 7: EETs vs GSNs

The results of our experiments are summarized in Figure 7. As one can observe, using the ethereless functionality (delegation mechanism) of 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. In contrast, using the GSN incurs a 4-5x increase in gas usage as compared to a self-funded transaction. This is the cost of the complexity of the GSN, a cost that is very unattractive for projects that do not require the extreme decentralization and genericity of the GSN.

References

[AB20] Ahmed Al-Balaghi. The state of meta transactions - 2020. https://medium.com/biconomy/the-state-of-meta-transactions-2020-506840e37e75, 2020.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, October 2018.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Can03] Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. https://eprint.iacr.org/2003/239.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

[DPS19] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 23–41. Springer, Heidelberg, February 2019.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 291–323. Springer, Heidelberg, August 2017.

[Gri18] Austin Griffith. Ethereum meta transactions, 2018. https://medium.com/@austin_48503/ethereum-meta-transactions-90ccf0859e84.

[gsn] Ethereum gas station network (gsn) documentation. https://docs.opengsn.org/.

[Kia] Aggelos Kiayias. Babel fees-denominating transaction costs in native tokens. https://iohk.io/en/blog/posts/2021/02/25/babel-fees/.

[KKKZ19] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy*, pages 157–174. IEEE Computer Society Press, May 2019.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.

[PS17] Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 380–409. Springer, Heidelberg, December 2017.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 643–673. Springer, Heidelberg, April / May 2017.

[VB15] Fabian Vogelsteller and Vitalik Buterin. Eip 20: Erc-20 token standard. 2015. https://eips.ethereum.org/EIPS/eip-20.

A Native vs. Contract-based Tokens

The experiments discussed below, use the same software and infrastructure setup as the first set (but different contracts).

- **Overhead of Native Tokens:** Adding native support for tokens on a cryptocurrency blockchain following the PPU principle means that every (even non-token) transaction processing will be slightly more (gas-)expensive than a transaction that does not support tokens. The reason is that miners/minters will at the very least need to check whether a transaction is native cryptocurrency (in which case it is added to a block as is) or a token transaction (in which case they will need to calculate if they are willing to fund its fees and compute the modified transaction to send to the network. As discussed, estimating this overhead in currently infeasible in lack of a relevant platform. Instead, here we attempt a lower-bound of this overhear, if it would be implemented in Ethereum. To this direction we implemented a simple contract `IfNoop` wich performs a conditional branch on the value of an input byte—corresponding to the check of whether it is a token or native cryptocurrency transaction—and then exits in either case of the branch. This approximates the overhead of a single 'if' statement, followed by native execution of either an ether or token transfer. We also implemented an equivalent contract, `IfNoopYul`, in Yul, which omits the overhead of setting up the Solidity runtime and performing method dispatch, and is therefore potentially a closer approximation of the true overhead.

- **Overhead of Smart-Contract Tokens:** Our second experiment considers a contract `IfFull` which performs a conditional branch on the value of an input byte, and in one case of the branch transfers the ether value of the calling transaction to an address specified in the remainder of the input data. This approximates the overhead of a contract implementing a rough equivalent of native tokens, i.e. handling both ether and token transfers. (The case of transfering tokens is already simulated by the self-funded transaction experiments above; the leading 'if' statement can be assumed to be simulated by the Solidity method dispatch at the beginning of the contract execution.) As above, to get a closer estimate we also implemented a contract `IfFullYul` in the lower level EVM language Yul with the same functionality as `IfFull`.

For each of the above contracts, we submitted identical input (a 1 byte, indicating an ether transfer for the contracts that perform it, and a fixed address to transfer to) 100 times over,

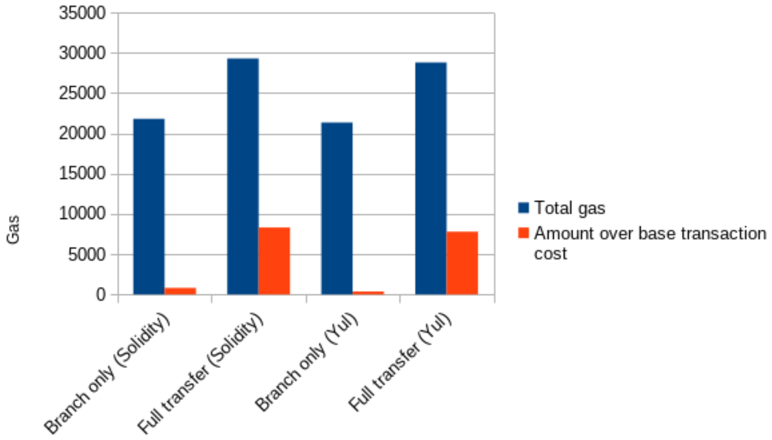

Figure 8: Native vs Smart-Contract-Simulated Tokens

measuring the gas usage for each. Unsurprisingly, since the contracts are deterministic and do not store or modify any state, each has a constant gas usage. Our experiments summarized, in Figure 8, demonstrate that even when only charging for the if-brach in a native implementation—which is clearly a favorable lower-bound on the overhead of every transaction on a native-tokens-enabled blockchain—the gas overhead of emulating tokens via a smart contract is $\approx 33\%$. We believe that this overhead is acceptable given the functionality and adaptability offered by smart contracts as opposed to natively-hardwired validation.

B Functionalities with Dynamic Party Sets

UC provides support for functionalities in which the set of parties that might interact with the functionality is dynamic. We make this explicit by means of the following mechanism, (which we describe almost verbatim from [BMTZ17, Sec. 3.1]): All the functionalities considered here include the following instructions that allow honest parties to join or leave the set $\mathcal{P}$ of players that the functionality interacts with, and inform the adversary about the current set of registered parties:

- Upon receiving $(\textsc{register}, \text{sid})$ from some party P_i (or from $\mathcal{A}$ on behalf of a corrupted P_i), set $\mathcal{P} := \mathcal{P} \cup \{p_i\}$. Return $(\textsc{register}, \text{sid}, p_i)$ to the caller.
- Upon receiving $(\textsc{de-register}, \text{sid})$ from some party $P_i \in \mathcal{P}$, the functionality updates $\mathcal{P} := \mathcal{P} \setminus \{P_i\}$ and returns $(\textsc{de-register}, \text{sid}, P_i)$ to P_i.
- Upon receiving $(\textsc{is-registered}, \text{sid})$ from some party P_i, return $(\textsc{register}, \text{sid}, b)$ to the caller, where the bit b is 1 if and only if $P_i \in \mathcal{P}$.
- Upon receiving $(\textsc{get-registered}, \text{sid})$ from $\mathcal{A}$, the functionality returns the response $(\textsc{get-registered}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

In addition to the above registration instructions, global setups (i.e. shared functionalities that are available both in the real and in the ideal world and allow parties connected to them to share state [CDPW07]) allow UC functionalities to register with them. Concretely, global setups include, in addition to the above party registration instructions, two registration/de-registration instructions for functionalities:

- Upon receiving $(\textsc{register}, \mathrm{sid}_G)$ from a functionality F with session-id sid, update $F := F \cup \{(F, \mathrm{sid})\}$.
- Upon receiving $(\textsc{de-register}, \mathrm{sid}_G)$ from a functionality F with session-id sid, update $F := F\{(F, \mathrm{sid})\}$.
- Upon receiving $(\textsc{get-registered}_F, \mathrm{sid}_G)$ from $\mathcal{A}$, return $(\textsc{get-registered}_F, \mathrm{sid}_G, F)$ to $\mathcal{A}$.

We use the expression sid_G to refer to the encoding of the session identifier of global setups. By default (and if not otherwise stated), the above four (or, in the case of global setups, seven) instructions will be part of the code of all ideal functionalities considered in this work. However, to keep the description simple, we will omit these instructions from the formal descriptions unless deviations are defined.

C Modeling Time and Clock-dependent Protocol Execution

Katz et al. [KMTZ13] proposed a methodology for casting synchronous protocols in UC by assuming they have access to an ideal functionality $\mathcal{F}_{\textsc{clock}}$, *the clock*, that allows parties to ensure that they proceed in synchronized rounds. Informally, the idea is that the clock keeps track of a round variable whose value the parties can request by sending $(\textsc{clock-read}, \mathrm{sid}_C)$ to $\mathcal{F}_{\textsc{clock}}$. This value is updated only once all honest parties send the clock a $(\textsc{clock-update}, \mathrm{sid}_C)$ command. We lift this idea to a shared setup: the global clock functionality $\mathcal{F}_{\textsc{clock}}$ is a shared clock that may interact with more than one protocol session. The global clock provides a means for parties to synchronize each of their sessions.[14] The clock can also be used as a local (not shared) hybrid functionality, in which case the number of sessions it will synchronize is simply one. The description is given in Figure 9.

Given a clock, the authors of [KMTZ13] describe how synchronous protocols can maintain their necessary round structure in UC: for every round ρ, each party first executes all of its round-ρ instructions, and then sends the clock a CLOCK-UPDATE command. Subsequently, whenever activated, it sends the clock a CLOCK-READ command and does not advance to round $\rho + 1$ until it sees that the clock's variable has been updated. This ensures that no honest party will start round $\rho + 1$ before every honest party has completed round ρ. In [KZZ16], this idea was transfered to the (G)UC setting by assuming that the clock is a global setup. This allows for different protocols to use the same clock, and this is the model we will also use here.

As argued in [KMTZ13], in order for an eventual-delivery (aka guaranteed termination) functionality to be UC-implementable by a synchronous protocol, it needs to keep track of the number of activations that an honest party gets, so that it knows when to generate output for honest parties. This requires that the protocol itself, when described as a UC interactive Turing-machine instance (ITI), has a predictable behavior when it comes to the pattern of activations that it needs before it

[14] The functionality presented here is different from shared clock functionalities used in prior work. We believe that the version here is closer to the spirit of the GUC/EUC version of UC.

Functionality $\mathcal{F}_{\textbf{clock}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e. parties $P = (\mathsf{pid}, \mathsf{sid})$. It also manages the set F of functionalities (together with their session identifiers). Initially, $\mathcal{P} := \emptyset$ and $F := \emptyset$.

For each session sid, the clock maintains a variable τ_{sid}. For each identity $P := (\mathsf{pid}, \mathsf{sid}) \in \mathcal{P}$, it maintains a variable d_P. For each pair $(\mathcal{F}, \mathsf{sid}) \in F$, it maintains a variable $d_{(\mathcal{F},\mathsf{sid})}$. All integer variables are initially 0.

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $P \in \mathcal{P}$, set $d_P := 1$, execute *Round-Update*, and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, P)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \mathsf{sid}) \in F$, set $d_{(\mathcal{F},\mathsf{sid})} := 1$, execute *Round-Update*, and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to the sending instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal — shared or local — functionality), return $(\text{CLOCK-READ}, \text{sid}_C, \tau_{\mathsf{sid}})$ to the requestor, where sid is the sid of the calling instance.

Procedure Round-Update: For each session sid do: If $d_{(\mathcal{F},\mathsf{sid})} := 1$ for all $\mathcal{F} \in F$ and $d_P = 1$ for all honest parties $P = (\cdot, \mathsf{sid}) \in \mathcal{P}$, then set $\tau_{\mathsf{sid}} := \tau_{\mathsf{sid}} + 1$, and reset $d_{(\mathcal{F},\mathsf{sid})} := 0$ and $d_P := 0$ for all parties $P = (\cdot, \mathsf{sid}) \in \mathcal{P}$.

Figure 9: The shared/global clock functionality. We assume lazy creation of variables, i.e. a variable is only created once it is needed.

sends the clock an update command. We capture this property in a generic manner in Definition C.1 (the content of this section and of Appendix D are taken almost verbatim from [BMTZ17]).

To follow the definition, recall the mechanics of activations in UC. In a UC protocol execution, an honest party (ITI) gets activated either by receiving an input from the environment, or by receiving a message from one of its hybrid-functionalities (or from the adversary). Any activation results in the activated ITI performing some computation on its view of the protocol and its local state, and ends with the party either sending a message to some of its hybrid functionalities, sending an output to the environment, or sending no message at all. In any of these cases, the party loses the activation.[15]

For any given protocol execution, we define the *honest-input sequence* $\vec{\mathcal{I}}_H$ to consist of all inputs that the environment gives to honest parties in the given execution, in the order in which they were given, along with the identity of the party who received the input. For an execution in which the environment has given m inputs to the honest parties in session sid in total, $\vec{\mathcal{I}}_H$ is a vector of the form $((x_1, id_1), \ldots, (x_m, id_m))$, where x_i is the i-th input that was given in this execution, and id_i is the corresponding identity (i.e. $id_i = (\mathsf{pid}_i, \mathsf{sid})$ for some bitstring pid) of the party that received this input in this session. We further define the *timed honest-input sequence,* denoted as $\vec{\mathcal{I}}_H^T$, to be the honest-input sequence augmented with the respective clock time at which each input was given. If the timed honest-input sequence of an execution is $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$, this means that $((x_1, id_1), \ldots, (x_m, id_m))$ is the honest-input sequence corresponding to this execution, and for each $i \in [n]$, τ_i is the time of the global clock when input x_i was handed to id_i.

Definition C.1. A $\mathcal{F}_{\text{CLOCK}}$-hybrid protocol Π has a *predictable synchronization pattern* iff there exists an algorithm $\mathsf{predict\text{-}time}_{\Pi}(\cdot)$ such that, for any possible execution of Π in a session sid (i.e. for any adversary and environment and any choice of random coins), the following holds: if $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$ is the corresponding timed honest-input sequence for this

[15] In the latter case the activation goes to the environment by default.

Ledger Element	Description
$\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$	The party sets and categories: Registered, honest, and honest-but-desynchronized, respectively.
$\vec{\mathcal{I}}_H^T$	The timed honest-input sequence.
predict-time	The function to predict the real-world time advancement.
$\mathtt{state}$	The ledger state, i.e. a sequence of blocks containing the content.
$\mathtt{buffer}$	The buffer of submitted input values.
$\mathtt{pt}_i$, $\mathtt{state}_i$	The pointer of party P_i into state $\mathtt{state}$. This prefix is denoted $\mathtt{state}_i$ for brevity.
$\vec{\tau}_{\mathtt{state}}$	A vector containing for each state block the time when the block added to the ledger state.
τ_L	The current time as reported by the clock.
$\mathtt{NxtBC}$	Stores the current adversarial suggestion for extending the ledger state.
Validate	Decides on the validity of a transaction with respect to the current state. Used to clean the buffer of transactions.
ExtendPolicy	The function that specifies the ledger's guarantees in extending the ledger state (e.g., speed, content etc.).
Blockify	The function to format the ledger state output.
$\mathtt{windowSize}$	The window size (number of blocks) of the sliding window.
$\mathtt{Delay}$	A general delay parameter for the time it takes for a newly joining (after the onset of the computation) miner to become synchronized.

Figure 10: Overview of main ledger elements such as parameters and state variables.

session, then for any $i \in [m-1]$:

$$\mathsf{predict\text{-}time}_{\Pi}((x_1, id_1, \tau_1), \ldots, (x_i, id_i, \tau_i)) = \tau_{i+1},$$

where τ_{i+1} is the clock time for this session (cf. Figure 9).

As we argue, all synchronous protocols described in this work are designed to have a predictable synchronization pattern.

D The Basic Transaction-Ledger Functionality

Functionality $\mathcal{F}_{\text{ledger}}$

General: The functionality is parametrized by four algorithms $\mathsf{Validate}$, $\mathsf{ExtendPolicy}$, $\mathsf{Blockify}$, and $\mathsf{predict\text{-}time}$, along with two parameters $\texttt{windowSize}, \texttt{Delay} \in \mathbb{N}$. The functionality manages the variables $\texttt{state}$, $\texttt{NxtBC}$, $\texttt{buffer}$, τ_L, and $\vec{\tau}_{\texttt{state}}$, as described above. Initially, $\texttt{state} := \vec{\tau}_{\texttt{state}} := \texttt{NxtBC} := \varepsilon$, $\texttt{buffer} := \emptyset$, $\tau_L = 0$.
For each party $P_i \in \mathcal{P}$ the functionality maintains a pointer $\texttt{pt}_i$ (initially set to 1) and a current-state view $\texttt{state}_i := \varepsilon$ (initially set to empty). The functionality keeps track of the timed honest-input sequence $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$).

Party management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (following the definition in the previous paragraph). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. If a new honest party is already registered with the clock at the time it is registered with the ledger, it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$, and the time of registration is recorded. If the current time is $\tau_L > 0$, the new party is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from $\mathcal{P}$, and therefore also from $\mathcal{P}_{DS}$ and $\mathcal{H}$. The ledger maintains the invariant that it is registered (as a functionality) with the clock whenever $\mathcal{H} \neq \emptyset$. A party is considered fully registered if it is registered with both the ledger and the clock.

Upon receiving any input I from any party or from the adversary, send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{F}_{\text{CLOCK}}$ and, upon receiving response $(\text{CLOCK-READ}, \text{sid}_C, \tau)$, set $\tau_L := \tau$ and do the following:

1. Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously, with both ledger and clock) since time $\tau' < \tau_L - \texttt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$. On the other hand, for any synchronized party $P \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if P is not registered to the clock, then $\mathcal{P}_{DS} \cup \{P\}$.
2. If I was received from an honest party $P_i \in \mathcal{P}$:
   (a) Set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T || (I, P_i, \tau_L)$.
   (b) Compute $\vec{N} = (\vec{N}_1, \dots, \vec{N}_\ell) := \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$ and, if $\vec{N} \neq \varepsilon$, set $\texttt{state} := \texttt{state} || \mathsf{Blockify}(\vec{N}_1) || \dots || \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\texttt{state}} := \vec{\tau}_{\texttt{state}} || \tau_L^\ell$, where $\tau_L^\ell = \tau_L || \dots, || \tau_L$.
   (c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$, then delete $\texttt{BTX}$ from $\texttt{buffer}$.
   (d) If there exists $P_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ or $\texttt{pt}_j < |\texttt{state}_j|$, then set $\texttt{pt}_k := |\texttt{state}|$ for all $P_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.
3. Depending on the input I and the ID of the sender, execute the respective code:
   - *Submiting a transaction:*
     If $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ and I was received from a party $P_i \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party P_i), do the following:
     (a) Choose a unique transaction ID txid and set $\texttt{BTX} := (\texttt{tx}, \text{txid}, \tau_L, P_i)$.
     (b) If $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 1$, then $\texttt{buffer} := \texttt{buffer} \cup \{\texttt{BTX}\}$.
     (c) Send $(\text{SUBMIT}, \texttt{BTX})$ to $\mathcal{A}$.
   - *Reading the state:*
     If $I = (\text{READ}, \text{sid})$ is received from a fully registered party $P_i \in \mathcal{P}$, then set $\texttt{state}_i := \texttt{state}|_{\min\{\texttt{pt}_i, |\texttt{state}|\}}$ and return $(\text{READ}, \text{sid}, \texttt{state}_i)$ to the requestor. If the requestor is $\mathcal{A}$, then send $(\texttt{state}, \texttt{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.
   - *Maintaining the ledger state:*
     If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $P_i \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$, then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{F}_{\text{CLOCK}}$. Otherwise, send I to $\mathcal{A}$.
   - *The adversary proposing the next block:*
     If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \dots, \text{txid}_\ell))$ is sent from the adversary, update $\texttt{NxtBC}$ as follows:
     (a) Set $\text{listOfTxid} \leftarrow \epsilon$.
     (b) For $i = 1, \dots, \ell$, if there exists $\texttt{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, P_i) \in \texttt{buffer}$ with ID $\text{txid} = \text{txid}_i$, then set $\text{listOfTxid} := \text{listOfTxid} || \text{txid}_i$.
     (c) Finally, set $\texttt{NxtBC} := \texttt{NxtBC} || (\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.
   - *The adversary setting state-slackness:*
     If $I = (\text{SET-SLACK}, (P_{i_1}, \widehat{\texttt{pt}}_{i_1}), \dots, (P_{i_\ell}, \widehat{\texttt{pt}}_{i_\ell}))$, with $\{P_{i_1}, \dots, P_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:
     (a) If for all $j \in [\ell] : |\texttt{state}| - \widehat{\texttt{pt}}_{i_j} \leq \texttt{windowSize}$ and $\widehat{\texttt{pt}}_{i_j} \geq |\texttt{state}_{i_j}|$, set $\texttt{pt}_{i_1} := \widehat{\texttt{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$.
     (b) Otherwise, set $\texttt{pt}_{i_j} := |\texttt{state}|$ for all $j \in [\ell]$.
   - *The adversary setting the state for desychronized parties:*
     If $I = (\text{DESYNC-STATE}, (P_{i_1}, \texttt{state}'_{i_1}), \dots, (P_{i_\ell}, \texttt{state}'_{i_\ell}))$, with $\{P_{i_1}, \dots, P_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\texttt{state}_{i_j} := \texttt{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, ok)$ to $\mathcal{A}$.

Figure 11: The ledger functionality. We write $[n]$ to denote the set $\{1, \dots, n\}$.