Eagle: Efficient Privacy Preserving Smart Contracts

Carsten Baum[1], James Hsin-yu Chiang[1], Bernardo David[2], Tore Kasper Frederiksen[3],

[1] Technical University of Denmark, Denmark
cabau@dtu.dk, jchi@dtu.dk
[2] IT University of Copenhagen, Denmark
bernardo@bmdavid.com
[3] Alexandria Institute, Denmark
tore.frederiksen@alexandra.dk

Abstract. The proliferation of Decentralised Finance (DeFi) and Decentralised Autonomous Organisations (DAO), which in current form are exposed to front-running of token transactions and proposal voting, demonstrate the need to shield user inputs and internal state from the parties executing smart contracts. In this work we present "Eagle", an efficient UC-secure protocol which efficiently realises a notion of privacy preserving smart contracts where both the amounts of tokens and the auxiliary data given as input to a contract are kept private from all parties but the one providing the input. Prior proposals realizing privacy preserving smart contracts on public, permissionless blockchains generally offer a limited contract functionality or require a trusted third party to manage private inputs and state. We achieve our results through a combination of secure multi-party computation (MPC) and zero-knowledge proofs on Pedersen commitments. Although other approaches leverage MPC in this setting, these incur impractical computational overheads by requiring the computation of cryptographic primitives within MPC. Our solution achieves security without the need of any cryptographic primitives to be computed inside the MPC instance and only require a constant amount of exponentiations per client input.

1 Introduction

Ethereum introduced the first implementation of Turing-complete smart contracts for blockchains, widely adopted for financial and contracting applications since its introduction in 2015. Smart contracts offer auditability and correctness guarantees, and as a consequence expose both their state and any submitted inputs to all participants of the blockchain network. This lack of privacy not only leaks user data but also gives rise to concrete attacks. For example, current Decentralised Finance (DeFi) and Decentralised Autonomous Organisations (DAO) are vulnerable to front-running [27] in token transactions and proposal voting. This motivates the need to shield user inputs and internal contract state from the very parties who execute smart contracts in a decentralized environment.

Challenges. Hawk [44] introduced the first notion of general-purpose privacy preserving smart contracts, which required users to privately submit both input strings and confidential balances to a trusted contract manager. Upon evaluation of the contract over private inputs, the contract manager settles the confidential outputs to a confidential ledger, proving in zero knowledge that these outputs have been obtained according to the contract's instructions. Importantly, in order to accommodate real-world applications such as DeFi or DAO's, we must extend the Hawk notion of confidential contracts as follows:

1. Distribute the role of the trusted third party in an efficient manner, avoiding a single point of failure without significantly sacrificing performance.
2. Only require clients to be online during a short input phase, as in the standard client-blockchain interaction model where clients only broadcast signed inputs.
3. Allow privacy preserving smart contracts to be long-running applications over indefinite rounds, as is the case in standard, public smart contracts.

Our Contributions. In this work we present "Eagle", a Universally Composable [20] protocol for achieving efficient privacy preserving smart contracts, which handles all the three challenges explained above: (1) is achieved by evaluating the contract's instructions via an outsourced secure multi-party computation (MPC) protocol, where clients provide private inputs and servers execute the bulk of the protocol to compute a function on these inputs without learning them. We use a MPC protocol that allows a public verifier to identify servers aborting at the output phase, so that cheating servers can be identified and financially punished, incentivizing fairness (*i.e.* if a server gets the output, all servers/clients also get it). (2) is accomplished with a novel input protocol which pre-processes data necessary for the servers to generate private outputs (*e.g.* token amounts) that are posted directly to the public ledger but can only be read by specific clients. (2) facilitates (3), realized by a *reactive* version of our MPC protocol, which maintains secret state over multiple rounds. Here, we contribute a model of long-running, privacy preserving contracts, which at the onset of each round accepts new inputs from any subset of clients. At the end of each round, clients get public outputs and servers keep a secret internal state, allowing evaluation to take place in a continuous, multi-round fashion, even if clients are offline (2).

Applications. Several general applications for privacy preserving smart contracts have already been proposed. **Auctions:** can be realized securely on-chain with privacy preserving smart contracts, as auctions implemented without privacy are vulnerable to front-running (miners can trivially observe individual bids posted to the ledger). **Identity management:** Decentralized Identity (DID) management considers the setting where user-attributes are posted to a ledger, in a certified, yet hidden manner. DID implemented with privacy preserving smart contracts enables proofs and computations on private identity attributes, facilitating their integration with blockchain applications. **KYC Mixing:** We can construct a privacy preserving smart contract to realize a mixer that enforces Anti Money Laundering (AML) policies. For example, such a mixer could use

DID to integrate Know Your Customer (KYC) information to either limit user permissions or the quantity of mixed tokens allowed per month. **Side-chains:** The MPC servers alone could be considered a privacy preserving side-chain. Multiple sets of MPC servers could work together with a single smart contract to realize a privacy preserving sharding scheme on any layer 1 chain with Turing complete smart contracts. **AMMs and DeFi via Cross-chain contracts:** Using ideas of P2DEX [9], we show that the MPC servers can interact with smart contracts on many different ledgers. Hence, privacy preserving smart contracts can work *across* multiple ledgers and different native tokens. This realizes cross-chain, front-running resistant automated market makers (AMMs) with strong privacy guarantees. We discuss these applications in more detail in Appendix F.

Our Techniques. We sketch our protocol in Fig. 1. We assume a set of clients $\mathcal{C}$ and MPC servers $\mathcal{P}$, both interacting with a ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$. The ledger hosts two deployed smart contract instances: $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ maintains a confidential ledger and is extended with $\mathcal{X}_{\mathsf{Lock}}$, which locks and redistributes confidential balances, output and jointly signed, by the MPC servers. Concretely our protocol runs the following phases:

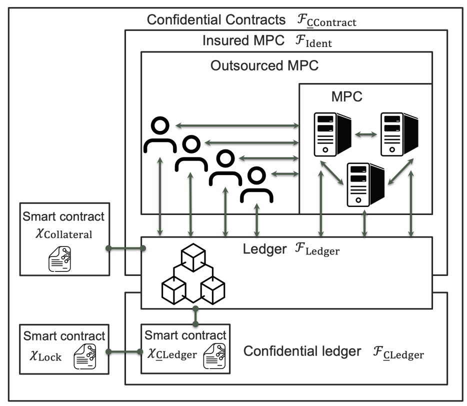


Fig. 1: Outline of our protocol for confidential contracts and how the different functionalities are wrapped and interact.

Init Before any execution, the servers setup the system by sampling a threshold signature key pair and provide sufficient collateral for the insured MPC execution, and setup smart contact $\mathcal{X}_{\mathsf{Lock}}$, administered by the distributed signature key.

Enroll Each client transfers the confidential tokens to $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ which they wish to use as input to the confidential smart contract $\underline{\mathsf{C}}\mathsf{Contract}$. The client then gives any auxiliary input, along with the opening information to the commitment containing their confidential balance v, to the insured MPC functionality $\mathcal{F}_{\mathsf{Ident}}$, extended with a secure client input interface (See Appendix B.1). Each client constructs an appropriate amount of "mask" commitments; one for each round of confidential contract computation, for which they wish their input to be used. A masking commitment is simply a commitment to a random value.

Verify input The servers validate the input received from the clients using outsourced MPC, and ensure that $\mathcal{X}_{\mathsf{Lock}}$ has also received the appropriate confidential tokens. The servers and the clients also execute a proof to ensure that the opening information supplied by clients are indeed valid for the confidential token commitments. They do this following a standard Σ-protocol where each client commits to a random commitment a and servers select a random challenge γ and ask the client to open $\text{com}(c) = \text{com}(a) \oplus (\gamma \odot \text{com}(v))$. Similarly the servers use MPC to securely open $[c] = [a] + \gamma \cdot [v]$ and check consistency[4].

Evaluate After the checks are completed the servers evaluate the circuit expressing the private smart contract $\underline{\mathsf{C}}\mathsf{Contract}$, using insured MPC. For the clients who are supposed to get output from this round of computation, shares of messages and randomness for a new commitment for *each* client are computed, and blinded with the "masking" values the clients provided during *Enroll*. If this goes well, the servers distributedly sign a message saying that they have reached this stage and post it to $\mathcal{X}_{\mathsf{Lock}}$.

Open For clients that receive output after this round of computation the servers open the masked output. They publish these values and sign them, as part of the transcript of the current round execution, and post this to $\mathcal{X}_{\mathsf{Lock}}$. Note that $\mathcal{X}_{\mathsf{Lock}}$ can generate the output coins in commitment form, due to the homomorphism of the commitments and since it obtained the mask commitments from the clients in *Enroll*. $\mathcal{X}_{\mathsf{Lock}}$ can then transfer the new confidential tokens back to the client's address. We show an extension to our protocol (Section 3.2) that ensures no token minting can occur even if all servers are corrupted.

Withdraw Based on the masks they constructed, the clients who are supposed to receive outputs can compute the coin commitment openings from their masked outputs signed and posted to $\mathcal{X}_{\mathsf{Lock}}$ by servers during *Open*.

Abort In case a server stops responding or acts maliciously, an honest server can request the entering of an abort phase. Any server can do this, either by submitting a proof that the malicious server sent wrong information or by requesting missing information from the accused servers. At this point the accused malicious server has a constant amount of time left to prove to the smart contract that they did not abort, by submitting the message that the accusing server claims they didn't get. If they don't, they will have their collateral revoked and it will be shared among the honest servers and clients, and the contract state will roll back one round, i.e. to the contract state preceding *Evaluate*. Concretely $\mathcal{X}_{\mathsf{Lock}}$ will refund the clients their input funds, plus a compensation obtained from the cheating servers' collateral.

Related Works. A long line of work realizes notions of privacy preserving smart contracts that sacrifice privacy [44,59,38,48,41,60,58,24] or flexibility [15,16]. Zexe [15] extends the ZCash model of confidential transactions to enable Bitcoin Script-like stateless privacy preserving smart contracts supporting only very simple logic. Zether [16] implements confidential transactions on top of Ethereum, allowing for very simple privacy preserving applications (*e.g.* auctions). zkay [59] allows for computing on encrypted private inputs by means of fully homomorphic encryption, requiring a trusted third party to hold decryption keys. Follow-up work, Zeestar [58] adds support for non-interactive zero-knowledge proofs to increase the expressiveness of supported contracts, but still needs a trusted third party. Secret Network [60] and Ekiden [24] implement

[4] In our full protocol we optimize this by batching client input checks.

general purpose contracts but rely on notoriously vulnerable trusted execution environments (*e.g.* Intel SGX [51]) for privacy and correctness. Arbitrum [38] relies on a full quorum of parties (the servers in our setting) being honest to achieve privacy for general purpose contracts. Finally, Kachina [41] subsumes these approaches with a framework based on state oracles [48] that yields privacy preserving smart contracts, where either flexibility is limited (*i.e.* contract state is only updated by one client's private input at a time) or privacy is compromised (*i.e.* a trusted third party must learn clients' private inputs in order to update the state).

Combining MPC with blockchain based cryptocurrencies and smart contracts has been investigated in a long line of works [1,2,12,46,45,47,43,25,13,11,31,7,9,8] aiming at achieving fairness in the dishonest majority setting via financial punishments. The core idea of these works is having all parties, who execute the MPC protocol, provide a collateral deposit, which is taken from them in case they are caught cheating. Thus incentivizing honest behavior. However, this approach publicly reveals the amount of collateral deposited by each party, which falls short of achieving our notion of privacy preserving smart contracts, where both auxiliary data *and* the amount of tokens given as input to the contract must remain private. Notice that revealing the deposit amount is an issue in applications where this amount is directly related to the client's private input, *e.g.* in sealed-bid auctions, where the collateral deposit must be equal to at least the client's private bid. An auction protocol using collateral deposits with private amounts was proposed in [32] but it cannot be generalized to other tasks.

A recent line of work, named zkHawk [4] and HawkNess [5], taking departure in Hawk [44], have also focused on realising privacy preserving smart contracts without a trusted contract manager. They do so via secure multi-party computation (MPC) but still exhibit limitations. Firstly, expensive cryptographic primitives such as non-interactive zero knowledge (NIZK) proofs are generated inside the MPC, incurring high overheads that degrade throughputs. Secondly, clients submitting inputs to a confidential contract are required to actively participate in the MPC protocol. Although it was shown in [39] that integrating NIZKs with MPC can be done without degrading performance too much, there is still a performance hit. Moreover, this approach does not support multi-round contracts. We argue that Eagle is a significant improvement to the state-of-the-art in bringing long-running, confidential contracts into practice.

In Appendix A we further discuss related works.

2 Preliminaries

Let $y \leftarrow\$ F(x)$ denote running the randomized algorithm F with input x and implicit randomness, and obtaining the output y. Similarly, $y \leftarrow F(x)$ is used for a deterministic algorithm. For a set $\mathcal{X}$, let $x \leftarrow\$ \mathcal{X}$ denote x chosen uniformly at random from $\mathcal{X}$. s denotes the computational and κ the statistical security parameter. Let $[\,x\,]$ denote secret x maintained in an MPC instance: we lift the $[\,\cdot\,]$ notation to any object that can be encoded over secrets securely input to an MPC scheme, e.g. $[\,g\,]$, where g is an arithmetic circuit over field $\mathbb{F}$.

We use a group $\mathbb{G}$ where the discrete log problem is hard, and which is a source group of pairing scheme. For simplicity we assume $|\mathbb{G}| = |\mathbb{F}| = p$. Unless noted otherwise we use log to denote the logarithm to base 2, rounded up. We use $\bar{v}_{\mathsf{max}}$ to denote the maximum amount of tokens we want to represent and say $l = \log(\bar{v}_{\mathsf{max}})$. For simplicity, we assume $|\mathcal{C}| \cdot \bar{v}_{\mathsf{max}} < |\mathbb{G}|$, where $\mathcal{C}$ is the set of participating clients. We denote the set $\{1, 2, \ldots, n\}$ by $[n]$ and denote vectors by bold faced Latin letters, e.g. $\mathbf{v}, \mathbf{w}$.

Table 1: Notation.

$\mathcal{P}$	The set of servers.
$\mathcal{C}$	The set of clients.
n	Amount of servers $n = \lvert\mathcal{P}\rvert$.
m	Amount of clients; $m = \lvert\mathcal{C}\rvert$.
l	Amount of bits representing balances.
z	Amount of input/output per client.
κ	Computational security parameter.
s	Statistical security parameter.
$\mathcal{F}$	An ideal functionality.
Π	A protocol.
$\mathcal{L}$	A ledger map indexed by vk.
$\mathcal{X}$	A smart contract program.
g	A smart contract in circuit form.
vk	A public key for signature verification.
x	A client input.
y	A client output.
$\bar{\mathbf{v}}$	A token balance.
$\bar{v}_{\mathsf{max}}$	The maximum permitted balance.
$\bar{\mathbf{v}}_{\mathsf{max}}$	A vector of the maximum permitted balance.

2.1 Security Model and Building Blocks

We analyse our results in the the (Global) Universal Composability or (G)UC framework [21,23]. We consider static malicious adversaries. Our protocols work in a synchronous communication setting, which is modelled by assuming parties have access to a global clock ideal functionality $\mathcal{F}_{\mathsf{Clock}}$ as in [3,40,43]. The core component of our protocols is publicly verifiable MPC with cheater identification in the output phase, which is modelled as an ideal functionality $\mathcal{F}_{\mathsf{Ident}}$ as in [7,9]. This functionality produces a proof that either a certain output was obtained after the MPC or that a certain party has misbehaved in the output phase, while cheating before the output phase causes an abort without cheater identification. We further extend this functionality to handle reactive computation as in [30,29] and an outsourced computation with inputs provided by clients and computation done by servers as in [37]. Moreover, we use Pedersen Commitments [53], digital signatures represented by an ideal functionality $\mathcal{F}_{\mathsf{Sig}}$ as in [22], threshold signatures represented by an ideal functionality $\mathcal{F}_{\mathsf{TSig}}$ as in [9] and non-interactive zero knowledge proofs represented by $\mathcal{F}_{\mathsf{NIZK}}$ as in [36]. Further discussion on our security model and building blocks is presented in Appendix B.

2.2 Ledgers & Smart Contracts

We model a ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ in Appendix C.1 featuring a smart contract virtual machine which is adapted from an authenticated, public bulletin board functionality, an approach adopted from [7,9]. For this work, we emphasize accurate modelling of confidential balances, which are implemented on a public ledger, and omit the full consensus details in our UC model, such as in [3] or other simplified UC formulations of consensus [43].

Token universe. $\mathcal{F}_{\mathsf{Ledger}}$ supports a token universe consisting of t token types: $\mathbb{T} = (\tau_1, ..., \tau_t)$. A ledger in $\mathcal{F}_{\mathsf{Ledger}}$ maintains a map from signature verification key to balances of each token type: $\mathcal{L} : \{0,1\}^* \to \mathbb{Z}^t$. We write $\bar{\mathbf{v}} = (v_1, ..., v_t)$ for a balance over all supported token types. In addition to balances associated to signature verification keys, $\mathcal{F}_{\mathsf{Ledger}}$ also maintains token balances for each deployed smart contract instance. The ledger functionality enforces the preservation of token supplies over $\mathbb{T}$.

A model of smart contracts. $\mathcal{F}_{\mathsf{Ledger}}$ parses authenticated messages which can authorize the deployment and activation of smart contracts, each modelled with a state transition function encoded as an arithmetic circuit T of maximum depth d_T, thereby enforcing a notion of bounded termination. Each contract maintains a public state fragment $\gamma \in \{0,1\}^*$ that is updated by circuit T upon the evaluation of each authenticated CallContract message. Each contract also maintains a balance $\bar{\mathbf{w}}$ of $\mathbb{T}$. We sketch the evaluation of a smart contract call with parameters $\mathsf{cn}, \mathsf{fn}, x, \bar{\mathbf{v}}$ authorized by signature verification key vk:

- *Contract identifier* cn selects the contract instance for evaluation.
- *Function selector* fn is an input that identifies the *contract interface* being evaluated, facilitating the logical separation of contract descriptions.
- *Input string* $x \in \{0,1\}^*$ denotes parameters input to circuit T: it is logically evaluated by the contract interface selected by fn.
- *Token balance* $\bar{\mathbf{v}}$ is provided to the contract call and is subtracted from the ledger entry associated with verification key vk.

The circuit T associated with contract instance cn is then evaluated on input $(\nu \,|\, \gamma \,|\, \bar{\mathbf{w}} \,|\, \mathsf{cn}, \mathsf{fn}, x, \bar{\mathbf{v}}, \mathsf{vk})$, where ν denotes $\mathcal{F}_{\mathsf{Clock}}$ round at the time of the call, and γ denotes the contract state stored by $\mathcal{F}_{\mathsf{Ledger}}$. Upon completed evaluation of T, $\mathcal{F}_{\mathsf{Ledger}}$ reads the encoding of a state transition $\mathcal{L}|\gamma|\bar{\mathbf{w}} \rightarrow^{\mathsf{ts}} \mathcal{L}'|\gamma'|\bar{\mathbf{w}}'$ from the output gates of evaluated T, thereby updating ledger, contract state and contract balance. Here, $\mathcal{F}_{\mathsf{Ledger}}$ asserts token supplies over $\mathbb{T}$ are preserved and that non-calling account balances cannot decrease from applying update ts.

We note the presence of call-back gates permitted in contract circuits deployed to $\mathcal{F}_{\mathsf{Ledger}}$, related to a UC-modelling technicality described in more detail in Appendix C.1. Concretely, these gates permit the UC functionality $\mathcal{F}_{\mathsf{Ledger}}$ to forward verification calls to hybrid functionalities $\mathcal{F}_{\mathsf{Ident}}$ and $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ via an honest majority committee $\mathcal{Q}$. Thus, in the hybrid $\mathcal{F}_{\mathsf{Ident}}$, $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$-setting, the simulator maintains the ability to equivocate and efficiently extract inputs from dishonest parties.

In this work, we present smart contracts as human-readable programs (e.g. $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}, \mathcal{X}_{\mathsf{Lock}}, \mathcal{X}_{\mathsf{Collateral}}$), and assume the presence of a compiler which translates program $\mathcal{X}$ to a valid circuit T and initial state γ_{init}.

Overview of smart contracts. The following smart contract programs are deployed in the protocol which realizes the proposed confidential contract functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$.

- $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ (Figure 11) describes a smart contract which implements a confidential token wrapper for each token in $\mathbb{T}$ supported on the base ledger $\mathcal{F}_{\mathsf{Ledger}}$.
- $\mathcal{X}_{\mathsf{Lock}}$ (Figure 14) is an extension to $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. It permits the locking and redistribution of confidential balances authorized by verifying threshold signatures generated by the servers (via global functionality $\mathcal{F}_{\mathsf{TSig}}$).

- $\mathcal{X}_{\mathsf{Collateral}}$ (Figure 15) accepts collateral deposits from servers, which upon being identified as cheating parties lose their collateral to clients.

2.3 Confidential ledgers from $\mathcal{F}_{\mathsf{Ledger}}$

We briefly describe a confidential ledger functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, presented in full detail in Appendix C.2, that can be implemented from a hybrid $\mathcal{F}_{\mathsf{Ledger}}$ functionality, enabling both confidential balances and the confidential transfer of default tokens types $\mathbb{T}$ exposed by the underlying public ledger $\mathcal{F}_{\mathsf{Ledger}}$. This modeling choice maximizes the generality of our construction, as it can be implemented on any standard ledger and a basic smart contract machine.

Confidential ledger. Confidential coins in $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ are identifiable by a unique public id, and a confidential balance $\bar{\mathbf{v}}$ over $\mathbb{T}$, as in [54]. Each confidential token is publicly associated with an account verification key vk, owned by a party that generated it with GenAcct. A confidential transfer consumes two input coins $(\mathsf{id}_1, \mathsf{id}_2)$ with confidential balances $(\bar{\mathbf{v}}_1, \bar{\mathbf{v}}_2)$ and mints fresh output coins $(\mathsf{id}'_1, \mathsf{id}'_2)$ with confidential balances $(\bar{\mathbf{v}}'_1, \bar{\mathbf{v}}'_2)$, such that $(\bar{\mathbf{v}}_1 + \bar{\mathbf{v}}_2 = \bar{\mathbf{v}}'_1 + \bar{\mathbf{v}}'_2)$. Here, coin id'_1 is now held by the owner of the receiving account, who also learns the confidential amount $\bar{\mathbf{v}}'_1$.

Functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ exposes Mint and Redeem interfaces: a mint activation *locks* a public amount of tokens $\mathbb{T}$ and generates a fresh confidential token of the same balance. Conversely, a redeem activation will *release* the balance of a confidential coin back to the public ledger.

Realizing a confidential ledger. A confidential token is realized in protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ described in full detail in Appendix D.1 with Pedersen Commitments [53]. Let $g, g_1, ..., g_t, h$ denote generators of group $\mathbb{G}$ of safe prime order p, such that s_i in $g_i = g^{s_i}$ and w in $h = g^w$ are given by $\mathcal{F}_{\mathsf{Setup}}$ (parameterized with $g \in \mathbb{G}$) that publicly outputs $g_1, ..., g_t, h$. The commitment to a balance $\bar{\mathbf{v}} = (v_1, ..., v_t)$ over tokens $\mathbb{T}$ with blinding r is $\mathsf{com}(\bar{\mathbf{v}}, r) = \mathbf{g}^{\bar{\mathbf{v}}} h^r = g_1^{v_1} ... g_t^{v_t} h^r$. Pedersen commitments are additively homomorphic: $\mathsf{com}(\bar{\mathbf{v}}_1, r_1) \circ \mathsf{com}(\bar{\mathbf{v}}_2, r_2) = \mathsf{com}(\bar{\mathbf{v}}_1 + \bar{\mathbf{v}}_2, r_1 + r_2)$. Thus, during a confidential transfer, the sum equality between consumed input and freshly constructed output coin commitments holds if total token balances are preserved and r'_1 and r'_2 are correlated such that $r_1 + r_2 = r'_1 + r'_2$.

$$\mathsf{com}(\bar{\mathbf{v}}_1, r_1) \circ \mathsf{com}(\bar{\mathbf{v}}_1, r_1) = \mathsf{com}(\bar{\mathbf{v}}'_1, r'_1) \circ \mathsf{com}(\bar{\mathbf{v}}'_2, r'_2) \tag{1}$$

However, since the equality above holds for any $\bar{\mathbf{v}}_1 + \bar{\mathbf{v}}_2 \equiv \bar{\mathbf{v}}'_1 + \bar{\mathbf{v}}'_2 \mod p$ and correlated r'_1, r'_2, an additional p units of each token in $\mathbb{T}$ can be minted: $\bar{v}_1 + \bar{v}_2 + p \equiv \bar{v}'_1 + \bar{v}'_2 \mod p$. Thus, each confidential token is associated with NIZK π which proves $\mathcal{R}(c; \bar{\mathbf{v}}, r) = \{c = \mathsf{com}(\bar{\mathbf{v}}, r) \wedge \bar{\mathbf{v}} \leq \bar{\mathbf{v}}_{\mathsf{max}} = 2^l - 1\}$, such that such wrap-around never occurs undetected.

We present a protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ which GUC-realizes $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ in Appendix D.1, where we also prove the following statement:

Theorem 1. *Protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ GUC-realizes functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ in the $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{Setup}}$, $\mathcal{F}_{\mathsf{Sig}}$-hybrid model against any PPT-adversary corrupting any minority of committee $\mathcal{Q}$.*

Recall from Section 2.2, that we assume an honest majority committee $\mathcal{Q}$, which forwards verification calls from $\mathcal{F}_{\mathsf{Ledger}}$ to hybrid functionalities $\mathcal{F}_{\mathsf{Ident}}$ and $\mathcal{F}_{\mathsf{NIZK}}$. We detail this interaction of $\mathcal{F}_{\mathsf{Ledger}}$ with UC-functionalities in Appendix C.1.

3 Confidential contracts

We present our formal model of confidential contracts. We assume m clients $\{C_1, \ldots, C_m\}$ and servers $\{P_1, \ldots, P_n\}$ that interact with $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$, which extends $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. For simplicity of presentation, we first present a single-round confidential contract functionality in Figure 2, and subsequently illustrate how it is easily extended to a multi-round contract functionality where clients can selectively choose to participate in specific rounds.

The choice of modelling $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ as an extension of $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ arises from the relation between underlying protocols: confidential coins in $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ that are committed to a confidential contract evaluation must be *locked* and subsequently *replaced* by a new set of output coins reflecting a new distribution of balances, determined by $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$. However, this requires verification operations over the homomorphic *commitment* representation of coins in $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$, which are not exposed by $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$.

We provide a brief sketch of the interface exposed by $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. Upon initialization with an arithmetic circuit g encoding only the contract logic, users can enroll, specifying input string x and a confidential coin to input, identified by its id. Upon a completed *Enroll*, the functionality is prompted by servers to evaluate circuit g on both client input strings, interpreted as numerical values, and input balances, with checks to ensure g does not mint tokens. $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ permits clients to withdraw anytime to retrieve the private output string and output balance. $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ permits the simulator to abort and indicate cheating servers, which are then penalized by the functionality.

Model of confidential contracts. Unlike public smart contracts deployed to $\mathcal{F}_{\mathsf{Ledger}}$, an instance of $\mathcal{F}_{\mathsf{Ident}}$ permits the computation of any arithmetic circuit on both private and public inputs.

We model a confidential contract as an arithmetic circuit over a field $\mathbb{F}_p$ consistent with the domain that $\mathcal{F}_{\mathsf{Ident}}$ is realized with. A well-formed confidential contract permits the writing of both *numerical* and *financial* inputs from each client to its input gates. Further, we enforce a maximum circuit depth d_T prior to the circuit evaluation to bound the rounds of interaction in the MPC instance.

$$\big(([\,y_1\,],[\,\bar{\mathbf{w}}_1\,]),...,([\,y_m\,],[\,\bar{\mathbf{w}}_m\,])\big) \leftarrow \mathsf{eval}_g\big(([\,x_1\,],[\,\bar{\mathbf{v}}_1\,])....,([\,x_m\,],[\,\bar{\mathbf{v}}_m\,])\big)$$

Upon confidential evaluation of a contract circuit g with well-formed depth and gates, the following assertion must be performed at each run-time over confiden-

Functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$, extends $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

$\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ interacts with clients $\mathcal{C} = \{C_1, ..., C_m\}$ and servers $\mathcal{P} = \{P_1, ..., P_n\}$.
The functionality exposes interfaces and and accesses the state of $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. It is parameterized with max. circuit depth d_T, and collateral balance $\bar{\mathbf{v}}_{\mathsf{coll}}$.

Init: On $(\textsc{Init}, sid, g)$ from $P_i \in \mathcal{P}$ forward messages to $\mathcal{S}$. If $\mathcal{S}$ continues.
1. Run **GenAcct** and **Init** procedures on $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$.
2. Assert that g is a circuit and that $\mathsf{depth}(g) \leq d_T$, store g.
3. Assert $\mathsf{vk} \in \mathcal{K}[P_i]$ and $\mathcal{L}[\mathsf{vk}] \geq \bar{\mathbf{v}}_{\mathsf{coll}}$.
4. Set $\mathcal{L}[\mathsf{vk}] \leftarrow \mathcal{L}[\mathsf{vk}] - \bar{\mathbf{v}}_{\mathsf{coll}}$.
   - If all servers have successfully called **Init**, set state to `enroll`, tick $\mathcal{F}_{\mathsf{Clock}}$.

Enroll: Upon input $(\textsc{Enroll}, sid, x, \mathsf{id}, \mathsf{vk})$ from client $C_j \in \mathcal{C}$,
1. Assert $\mathsf{vk} \in \mathcal{K}[C_j]$ and $\langle \mathsf{id}, \bar{\mathbf{v}} \rangle \in \mathcal{L}[\mathsf{vk}]$.
2. Forward $(\textsc{Enroll}, sid, \mathsf{id}, \mathsf{vk})$ to $\mathcal{S}$, if $\mathcal{S}$ aborts, run **Abort**. Otherwise, continue.
3. Assert state is in `enroll` and $\exists \langle \mathsf{id}, \bar{\mathbf{v}} \rangle \in \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}]$: then remove $\langle \mathsf{id}, \bar{\mathbf{v}} \rangle$.
4. Store input $(x_j, \bar{\mathbf{v}}_j)$.
   - If all clients have successfully called **Enroll**, tick $\mathcal{F}_{\mathsf{Clock}}$.

Execute: Upon input $(\textsc{Execute}, sid)$ from $P_i \in \mathcal{P}$,
1. If $\textsc{Execute}$ received from all $\mathcal{P}$ and $\mathcal{F}_{\mathsf{Clock}}$ ticked since state update to `enroll`, forward $(\textsc{Execute}, sid)$ to $\mathcal{S}$ and wait for *Ok* or *Abort*. If *Ok*, continue.
   a. Evaluate circuit g over current user inputs $\{(x_j, \bar{\mathbf{v}}_j)\}_{j \in [m]}$ and client state.
   c. Store client states $\{(y_j, \bar{\mathbf{w}}_j)\}_{j \in [m]}$ read from output gates of g.
      - Assert $\sum_{j \in [m]} \bar{\mathbf{v}}_j = \sum_{j \in [m]} \bar{\mathbf{w}}_j$. Tick $\mathcal{F}_{\mathsf{Clock}}$.
2. Forward $(\textsc{Evaluate}, sid)$ to $\mathcal{S}$ and wait for *Ok* or *Abort*.
   - If *Ok* returned, set state to `evaluated` and tick $\mathcal{F}_{\mathsf{Clock}}$.
3. Send $(\textsc{Output}, sid, \{y_j, \bar{\mathbf{w}}_j\}_{j \in [m]})$ to $\mathcal{S}$ and wait for *Ok* or *Abort*.
   - If $\mathcal{S}$ aborts, it provides cheating server set $\mathcal{J}$, run **Abort** with $\mathcal{J}$.
4. For $j \in [m]$, get a unique id'_j from $\mathcal{S}$, and set $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_j] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_j] \cup \{\langle \mathsf{id}'_j, \bar{\mathbf{w}}_j \rangle\}$.
5. Set state to `enroll` and tick $\mathcal{F}_{\mathsf{Clock}}$.

Withdraw: Upon $(\textsc{Withdraw}, sid)$ from $C_j \in \mathcal{C}$, obtain newly stored outputs since last *Withdraw* by $C_j \in \mathcal{C}$. Return $((y_{j,1}, \langle \mathsf{id}'_{j,1}, \bar{\mathbf{w}}_{j,1} \rangle), ..., (y_{j,l}, \langle \mathsf{id}'_{j,l}, \bar{\mathbf{w}}_{j,l} \rangle))$.

Abort: Tick $\mathcal{F}_{\mathsf{Clock}}$,
a. If state is `enroll`, return server and client funds: update $\mathcal{L}$, $\mathcal{L}_{\mathsf{Conf}}$.
b. Else if state in `evaluated`, obtain cheating servers $\mathcal{J}$ from $\mathcal{S}$:
   - If $\mathcal{J} \neq \emptyset$, reimburse clients $\mathcal{C}$ and honest servers $\mathcal{P} \backslash \mathcal{J}$, then distribute $\mathcal{J}$'s collateral amongst $\mathcal{C}$: update $\mathcal{L}$, $\mathcal{L}_{\mathsf{Conf}}$ accordingly.
   - Else if $\mathcal{J} = \emptyset$, obtain $\{(y_j, \bar{\mathbf{v}}_j)\}_{j \in [m]}$ from last evaluation of circuit g.
      - For $C_j \in \mathcal{C}'$, sample $\mathsf{id}_j \leftarrow\$ \mathbb{F}$ and set $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_j] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_j] \cup \{\langle \mathsf{id}_j, \bar{\mathbf{v}}_j \rangle\}$.
      - Return collateral for all $\mathcal{P}$: update $\mathcal{L}$ accordingly.
c. Terminate.

Fig. 2: Functionality for Confidential Contracts

tial inputs and outputs of evaluated g: namely, that token supplies have been preserved.

$$\sum_{i \in [m]} [\, \bar{\mathbf{v}}_i \,] =^? \sum_{i \in [m]} [\, \bar{\mathbf{w}}_i \,] \qquad (2)$$

One-round client-server interaction. Upon providing inputs to a confidential contract execution, clients can go off-line and retrieve confidential outputs with *Withdraw* at any later point in time.

3.1 Realizing the confidential contract functionality

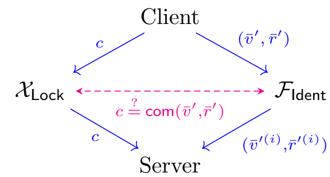
Overview of Protocol. Having provided a high-level overview of the protocol phase in Section 1, we now proceed to detail the individual protocol phases and refer to Appendix D.2 for the full protocol description and UC-security proof.

Setup of contracts. Servers deploy instances of $\mathcal{X}_{\mathsf{Lock}}[\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}]$, $\mathcal{X}_{\mathsf{Collateral}}$ on $\mathcal{F}_{\mathsf{Ledger}}$. Since wrapper $\mathcal{X}_{\mathsf{Lock}}$ extends $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, both are deployed and initialized as a single contract instance on $\mathcal{F}_{\mathsf{Ledger}}$ with shared contract id ($\mathsf{cn}_{\mathsf{Lock}}$) and shared state such as the confidential ledger ($\mathcal{L}_{\mathsf{Conf}}$). Here, the function of $\mathcal{X}_{\mathsf{Lock}}$ is to lock the confidential coins of clients input to the confidential contract evaluation, and to *replace* these with a new confidential distribution according to result of the contract evaluation. Further, $\mathcal{X}_{\mathsf{Lock}}$ is initialized with a threshold signature verification key $\mathsf{vk}_{\mathsf{TSig}}$, jointly generated by all servers via $\mathcal{F}_{\mathsf{TSig}}$: whenever servers agree on a new status of the contract evaluation in $\mathcal{F}_{\mathsf{Ident}}$, this agreement can be settled in $\mathcal{X}_{\mathsf{Lock}}$ with a threshold signature jointly generated via global functionality $\mathcal{F}_{\mathsf{TSig}}$. $\mathcal{X}_{\mathsf{Collateral}}$ is parameterized by $\mathsf{cn}_{\mathsf{Lock}}$ and is activated each time $\mathcal{F}_{\mathsf{Clock}}$ progresses: it obtains collateral from all participating servers. It observes any recorded cheating servers $\mathcal{J}$ stored in the state of contract instance $\mathsf{cn}_{\mathsf{Lock}}$ and enforces penalties accordingly.

Client enrollment. Clients interact with $\mathcal{X}_{\mathsf{Lock}}$ to enroll a confidential coin it controls to the contract evaluation, and send both the coin commitment opening and numerical input x to an instance of $\mathcal{F}_{\mathsf{Ident}}$. Enrolled coins are removed from the confidential ledger $\mathcal{L}_{\mathsf{Conf}}$ maintained by $\mathcal{X}^{\underline{\mathsf{C}}\mathsf{Ledger}}$ and moved to a dedicated ledger $\mathcal{L}_{\mathsf{Lock}}$ for funds committed to a pending MPC computation in $\mathcal{F}_{\mathsf{Ident}}$.

Clients must also commit to a *output mask* during enrollment, which enables the subsequent redistribution of confidential coins without client interaction in the output phase of the contract evaluation. Here each client with confidential coin input c and numerical input x performs the following:
- Samples $\hat{y} \leftarrow\!\$\, \mathbb{F}$ as a numerical output mask and sends to $\mathcal{F}_{\mathsf{Ident}}$.
- Samples $\hat{\mathbf{w}} \leftarrow\!\$\, \mathbb{F}^{|\mathbb{T}|}$, $\hat{s} \leftarrow\!\$\, \mathbb{F}$, and computes mask commitment $\hat{c} \leftarrow \mathsf{com}(\hat{\mathbf{w}}, \hat{s})$.
- Sends mask commitment $\hat{c}$ to $\mathcal{X}_{\mathsf{Lock}}$ on $\mathcal{F}_{\mathsf{Ledger}}$.
- Sends mask commitment openings $(\hat{\mathbf{w}}, \hat{s})$ of $\hat{c}$ to $\mathcal{F}_{\mathsf{Ident}}$.

Input verification. Upon enrollment of clients, servers must verify that the confidential coin c and mask commitment $\hat{c}$ sent to $\mathcal{X}_{\mathsf{Lock}}$ are consistent with their respective openings $(\bar{\mathbf{v}}, \bar{r})$ and $(\hat{\mathbf{v}}, \hat{r})$ sent to $\mathcal{F}_{\mathsf{Ident}}$ during enrollment. For simplicity of presentation, we illustrate the batched input verification of input confidential coins and their openings assuming a token universe size of $|\mathbb{T}| = 1$, such that $c = g^{\bar{v}} h^{\bar{r}}$. Input verification for output masks $\hat{c}$ and their openings submitted to $\mathcal{F}_{\mathsf{Ident}}$ follow similarly.



Each server obtains both confidential coin c from $\mathcal{X}_{\mathsf{Lock}}$ and *additive shares* of submitted openings thereof from $\mathcal{F}_{\mathsf{Ident}}$, namely $(\bar{v}'^{(i)}, \bar{r}'^{(i)})$. We write $\bar{v}'^{(i)} = (\bar{v}+\epsilon)^{(i)}$ and similarly for $\bar{r}'^{(i)}$, where the ϵ denotes the error or discrepancy that the adversary can introduce to $\bar{v}$. We employ a standard technique of evaluating a random linear combination over client inputs to verify consistency.

1. Servers jointly sample $\gamma, \alpha, \beta \leftarrow\$ \mathbb{F}$ and open γ.
2. Each server locally computes the following on the inputs from m clients.
   - $\bar{v}'^{(i)}_{\mathsf{lin}} = \alpha^{(i)} + \gamma\, \bar{v}'^{(i)}_1 + \ldots + \gamma^m\, \bar{v}'^{(i)}_m$ and $r'^{(i)}_{\mathsf{lin}} = \beta^{(i)} + \gamma\, r'^{(i)}_1 + \ldots + \gamma^m\, r'^{(i)}_m$
   - Subsequently, it sends $\bar{v}'^{(i)}_{\mathsf{lin}}$ and $\bar{v}'^{(i)}_{\mathsf{lin}}$ to all other servers.
3. Each server locally reconstructs $\bar{v}'_{\mathsf{lin}} = \prod_{i\in[n]} \bar{v}'^{(i)}_{\mathsf{lin}}$ and $r'_{\mathsf{lin}} = \prod_{i\in[n]} r'^{(i)}_{\mathsf{lin}}$
4. Servers locally verify: $\prod_{i\in[n]} g^{\alpha^{(i)}} h^{\beta^{(i)}} \prod_{j\in[m]} c_j^{\gamma^j} \stackrel{?}{=} g^{\bar{v}'_{\mathsf{lin}}} h^{r'_{\mathsf{lin}}}$

Note that $\bar{v}'^{(i)}_{\mathsf{lin}}$ and $r'^{(i)}_{\mathsf{lin}}$ are shares held by servers and do not reveal the values of user inputs. We write $\bar{v}'_{\mathsf{lin}} = \alpha + \gamma\,(\bar{v}_1 + \epsilon_{\bar{v}_1}) + \ldots + \gamma^m\,(\bar{v}_m + \epsilon_{\bar{v}_m})$ and similarly for r'_{lin} to expose ϵ's introduced by the adversary. If ϵ values are committed to by the adversary before α, β, γ are sampled, we can interpret $\bar{v}'_{\mathsf{lin}} - \bar{v}_{\mathsf{lin}} = 0$ and $r'_{\mathsf{lin}} - r_{\mathsf{lin}} = 0$ as m - degree polynomials with coefficients chosen by the adversary that are later evaluated at some random coordinate γ: since verification step (4) implies exactly these assertions, the probability for an undetected non-zero error is therefore $m/|\mathbb{G}|$, where m is the number of polynomial roots, by the Schwartz-Zippel Lemma.

Execute. Servers call the **Evaluate** interface on $\mathcal{F}_{\mathsf{Ident}}$ to evaluate circuit g with input gates set to client inputs.

$$([\,x_1\,], [\,\hat{y}_1\,], [\,\bar{\mathbf{v}}_1\,], [\,r_1\,], [\,\hat{\mathbf{w}}_1\,], [\,\hat{s}_1\,]), \ldots, ([\,x_m\,], [\,\hat{y}_m\,], [\,\bar{\mathbf{v}}_m\,], [\,r_m\,], [\,\hat{\mathbf{w}}_m\,], [\,\hat{s}_m\,])$$

Upon secure evaluation, outputs in form of numerical values and balances are written to the output gates of g: $\big(([\,y_1\,], [\,\bar{\mathbf{w}}_1\,]), \ldots, ([\,y_m\,], [\,\bar{\mathbf{w}}_m\,])\big)$. Before masking these for opening, the servers then perform a confidential consistency check to ensure the preservation of tokens as shown in Equation (2).

Masked output values are obtained by applying the masking values input by users, $[\,y'_j\,] = [\,y_j\,] + [\,\hat{y}_j\,]$ and similarly for balances, $[\,\bar{\mathbf{w}}'_j\,] = [\,\bar{\mathbf{w}}_j\,] + [\,\hat{\mathbf{w}}_j\,]$ and generating a joint signature $\sigma_{\mathsf{vk}_{\mathsf{TSig}}}(\mathsf{evaled})$ via $\mathcal{F}_{\mathsf{TSig}}$, that is sent to $\mathcal{X}_{\mathsf{Lock}}$ on $\mathcal{F}_{\mathsf{Ledger}}$. Upon verification, the $\mathcal{X}_{\mathsf{Lock}}$ contract updates the state of protocol execution, reflecting completion of the *Execute* phase.

Open. Servers run **Optimistic Reveal** in $\mathcal{F}_{\mathsf{Ident}}$ to open masked numerical outputs and balances $\big((y'_1, \bar{\mathbf{w}}'_1), \ldots, (y'_m, \bar{\mathbf{w}}'_m)\big)$. Should all servers agree on the successful completion of the contract evaluation, they jointly sign all *masked* outputs and send these to $\mathcal{X}_{\mathsf{Lock}}$ (on $\mathcal{F}_{\mathsf{Ledger}}$), which then computes the *unmasked* confidential coins for clients with the newly computed distribution as follows.

Given the masked output balance $\bar{\mathbf{w}}'$ from $\mathcal{F}_{\mathsf{Ident}}$ and the coin mask $\hat{c}$ sampled by a client in **Enroll**, contract $\mathcal{X}_{\mathsf{Lock}}$ computes

(a) The masked confidential coin: $c^{\mathsf{out}\,\prime} \leftarrow \mathbf{g}^{\bar{\mathbf{w}}'} h^0$
(b) The unmasked confidential coin: $c^{\mathsf{out}} \leftarrow c^{\mathsf{out}\,\prime} \cdot \hat{c}^{-1}$

We rewrite (b) as $c^{\mathsf{out}} = \mathbf{g}^{\bar{\mathbf{w}}' - \hat{\mathbf{w}}} h^{-\hat{s}} = \mathsf{com}(\bar{\mathbf{w}}, -\hat{s})$ to expose the unmasking of the output coin without any knowledge of the final balance. $\mathcal{X}_{\mathsf{Lock}}$ subsequently stores unmasked output coin c^{out} in the confidential ledger in $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, thereby settling the output balance distribution read from output gates of contract circuit g. Should $\mathcal{X}_{\mathsf{Lock}}$ successfully verify the signed outputs, $\mathcal{X}_{\mathsf{Collateral}}$ will infer from the state of $\mathcal{X}_{\mathsf{Lock}}$ the completion of a successful round and return the deposited collateral to the servers.

Withdraw. Upon a successful **Open**, the output of the confidential contract evaluation has completed. Each client can obtain their masked output $(y', \bar{\mathbf{w}}')$ from $\mathcal{X}_{\mathsf{Lock}}$ and newly minted c_{out} from $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ anytime following a successful execution of a contract evaluation. Let $\hat{y}$ and $(\hat{\mathbf{w}}, \hat{s})$ be the output masks generated by the client in **Enroll**. The withdrawing client obtains

(a) The numerical output: $y \leftarrow y' - \hat{y}$
(b) The opening of the output coin: $(\bar{\mathbf{w}}, s) \leftarrow (\bar{\mathbf{w}}' - \hat{\mathbf{w}}, -\hat{s})$

Abort. If the protocol aborts prior to the completion of the **Execute** phase, client funds are simply returned by $\mathcal{X}_{\mathsf{Lock}}$ and collateral deposited to $\mathcal{X}_{\mathsf{Collateral}}$ is returned. If servers have agreed upon the completion of **Execute**, honest servers can interact with $\mathcal{F}_{\mathsf{Ident}}$ to either (a) obtain shares that are verifiable and enable reconstruction of the output or (b) identify cheating servers (Figure 7). Thus, $\mathcal{X}_{\mathsf{Lock}}$ as a registered public verifier, can identify cheating servers by either verifying shares with $\mathcal{F}_{\mathsf{Ident}}$, or obtaining the identities of servers $\mathcal{J}$ that refuse to participate in revealing their shares and allowing their verification. Cheating servers lose their collateral held by $\mathcal{X}_{\mathsf{Collateral}}$ which is redistributed to clients.

We present the full protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$ which GUC-realizes $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ in Appendix D.2 and prove the following statement.

Theorem 2. *$\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}[\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}]$ realizes $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}[\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}]$ in the $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Ident}}$, $\mathcal{F}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{Setup}}$, $\mathcal{F}_{\mathsf{Sig}}$, $\mathcal{F}_{\mathsf{TSig}}$-hybrid model against any PPT-adversary corrupting at most $n-1$ of the n servers $\mathcal{P}$ statically and any minority of $\mathcal{Q}$.*

3.2 Confidential contract extensions

Multi-round confidential contracts. We now demonstrate how our default model of confidential contracts, shown in previous sections, can be extended to a multi-round model, where clients can provide fresh inputs and obtain continuous outputs in a long-running confidential blockchain application.

This is facilitated by our model of confidential contracts, which does not require server-client interaction beyond the *Open* phase, along with the reactive interface of our MPC functionality (Appendix B.1), which permits the selective opening of secrets and indefinite number of circuit evaluations on stored secret values.

More concretely, let the confidential state of a client be a tuple consisting of a numerical value and balance: $[\,s_j\,] = ([\,y_i\,], [\,\bar{\mathbf{w}}_i\,])$. Further, let the *confidential contract state* be defined over all confidential client states $\mathsf{st} = ([s_1], ..., [s_m])$, which is stored from the previous contract evaluation round or given as the initial confidential contract state. We define a *confidential contract state transition* that consumes a fresh set of confidential contract inputs $\mathsf{st}^{\mathsf{in}} = ([s_1^{\mathsf{in}}], ..., [s_m^{\mathsf{in}}]) = (([\,x_1^{\mathsf{in}}\,], [\,\bar{\mathbf{v}}_1^{\mathsf{in}}\,]), ..., ([\,x_m^{\mathsf{in}}\,], [\,\bar{\mathbf{v}}_m^{\mathsf{in}}\,]))$, such that the the *current contract circuit* $\mathbf{g}$ is evaluated over both st and $\mathsf{st}^{\mathsf{in}}$ to obtain a new confidential state st' and an encoding of the updated circuit $\mathbf{g}'$ to be evaluated in the next round.

$$\big([\,\mathbf{g}'\,], [\,s_1'\,], ..., [\,s_m'\,]\big) \leftarrow \mathsf{eval}_{\mathbf{g}}\big([s_1^{\mathsf{in}}], ..., [s_m^{\mathsf{in}}], [\,s_1\,], ..., [\,s_m\,]\big)$$

Upon successful completion of a round evaluation, circuit $\mathbf{g}'$ will be securely opened, stored and evaluated by the servers in the following round. Each client can retrieve its new state by executing *Withdraw*. A discussion of mask generation and selective participation in the multi-round model is given in Appendix E.

Mitigation of token minting. Under full server corruption, it is possible for the adversary to mint confidential balances beyond the supply of underlying tokens wrapped by $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. This is because in our default protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$ shown in Figure 12, any output coin distribution accompanied by a verifying threshold signature will be accepted by $\mathcal{X}_{\mathsf{Lock}}$; no coin sum-checks or range-proofs enforce the preservation of confidential token supplies (See Equation 1). This can mitigated by following extensions to our model of confidential contracts.

1. **Bit decomposition of coins.** We propose a protocol extension that bit-decomposes both output coin commitments and their masks. The number of bits l in this representation ensures that each coin balance does not exceed $\bar{v}_{\mathsf{max}} \leq 2^l - 1$. The zero-sum property of input and output coin commitments (Equation 1) is ensured by a proof to $\mathcal{X}_{\mathsf{Lock}}$, constructed by the servers *without* the need for computing cryptographic primitives inside the MPC circuit. Thus we preserve efficiency, without requiring client interaction during the output phase. We detail this extension in Appendix E where we also state its impact on protocol complexity.
2. **Outputs with client interaction.** If clients can be available during the output phase, they can alternatively retrieve their respective outputs and subsequently generate range-proofs and a zero-sum proof over input and output coins. This approach to mitigate token minting under full server corruption violates our stated goal of minimizing client interaction, but is stated here for completeness.

4 Efficiency

We note that since previous works focus on using zero knowledge proofs and a trusted contract manager, we refrain from directly comparing our efficiency to their works. The closest previous works [4,5] to ours do not provide an efficiency analysis, making it hard to provide a meaningful comparison. Notice, however, that these protocols require computing cryptographic primitives *inside* MPC, whereas we only use these primitives externally. In the following analysis, we assume Bulletproofs for range proofs and standard Fiat-Shamir Schnorr proofs of knowledge of exponents using elliptic curves. Although neither of these are UC-secure since knowledge extraction requires rewinding, there is evidence [34] that these techniques can be made non-malleable in the algebraic group model. Hence, for the purpose of efficiency we believe it is reasonable to forgo the formal UC security in this section. We use BLS threshold signatures and for simplicity we assume the size of the group used for BLS and commitments is the same, although it will in practice be slightly larger for BLS.

		Init	Execution	Abort
User	**exp**	2	2	0
Server	**exp**	$2+2(n-1)$	$6\|\mathcal{C}\|+2$	0
	pair	0	$n-1$	0
	mult	0	$z\|\mathcal{C}\|$	0
SC comp.	**exp**	0	$2\|\mathcal{C}\|z$	$\|\mathcal{C}\|$
	pair	0	2	0
SC call space	$\#\mathbb{G}$ elem.	3	$\|\mathcal{C}\|z$	$O(n\|\mathcal{C}\|z)$
Comm.	$\#\mathbb{G}$ elem.	$O(n)$	$O(n^2 \cdot z \cdot \|\mathcal{C}\|)$	$O(n^2 \cdot z \cdot \|\mathcal{C}\|)$

Table 2: Complexity of our protocol when executing one $\underline{\mathsf{C}}\mathsf{Contract}$, *excluding* the computation of contract circuit g in MPC. We assume $|\mathcal{C}|z > s$ for statistical security parameter s, where z is the amount of input/output for each client in the set of clients $\mathcal{C}$, including the hidden token amount. $n = |\mathcal{P}|$ is the amount of servers and **mult** denotes the number of multiplications in MPC.

We outline the amount of heavy computations needed for our core protocol in Table 2, *except* what is needed by the underlying MPC computation computing the contract circuit g, reflecting the privacy preserving smart contract $\underline{\mathsf{C}}\mathsf{Contract}$. Concretely we count the amount of group exponentiations when assuming that the Pedersen commitments are realized using elliptic curves, along with pairings assuming BLS [14] has been used for realizing distributed signatures. The table only contains the complexity of executing one instance of $\underline{\mathsf{C}}\mathsf{Contract}$, but we note that execution of multiple contracts is slightly sublinear in the complexity of a single execution. The *Abort* column illustrates the *additional* overhead associated with a cheating party.

	Mint	ConfTansfer	Redeem
User	4	$O(\log(\bar{v}_{\mathsf{max}}) \cdot \log(\log(\bar{v}_{\mathsf{max}})))$	3
SC comp.	3	$O(\log(\bar{v}_{\mathsf{max}}))$	3
SC space	3	$2\log(\bar{v}_{\mathsf{max}}) + 10$	4

Table 3: Complexity of $\underline{\mathsf{C}}\mathsf{Ledger}$ in group exponentiation and amount of group elements stored, when $\bar{v}_{\mathsf{max}}$ is the maximum amount of allowed tokens (Recall $|\mathcal{C}| \cdot \bar{v}_{\mathsf{max}} < |\mathbb{G}|$).

When it comes to our confidential token layer, we outline the complexity in Table 3. We note that the constant in the complexity of *Confidential Transfer* reflects two range proofs over $\log(|\mathbb{G}|/2)$, under the assumption that BulletProofs are used [17]. Although if the domain of the token amounts is further limited from $\mathbb{G}$ to $\bar{v}_{\mathsf{max}} < |\mathbb{G}|/|\mathcal{C}|$ then they can be reduced to range proofs of $[0; \bar{v}_{\mathsf{max}} - 1]$ and thus complexity $O(\bar{v}_{\mathsf{max}} \cdot \log(\bar{v}_{\mathsf{max}}))$.

In both tables the amount of smart contract space is only what needs to be submitted. The persistent space use needed is only $3+3|\mathcal{C}|$ group elements, if we assume that the storage used when posting to $\mathcal{X}_{\mathsf{Lock}}$ in *evaluate* and *open* gets overwritten the next time the servers call these methods.

The round complexity for all steps of both the confidential token layer protocols and our core protocol is constant, assuming g has constant multiplicative depth. Otherwise, the computation of g dominates the round complexity.

References

1. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Fair two-party computations via bitcoin deposits. In: FC 2014 Workshops (Mar 2014)
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multi-party computations on bitcoin. In: 2014 IEEE Symposium on Security and Privacy (May 2014)
3. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: Annual international cryptology conference. pp. 324–356. Springer (2017), https://doi.org/10.1007/978-3-319-63688-7_11
4. Banerjee, A., Clear, M., Tewari, H.: zkhawk: Practical private smart contracts from mpc-based hawk. In: 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 245–248. IEEE (2021), https://doi.org/10.1109/BRAINS52497.2021.9569822
5. Banerjee, A., Tewari, H.: Multiverse of HawkNess: A Universally-Composable MPC-based Hawk Variant. Cryptology ePrint Archive (2022), https://eprint.iacr.org/2022/421
6. Baum, C., Chiang, J., David, B., Frederiksen, T., Gentile, L.: Sok: Mitigation of front-running in decentralized finance. The 2nd Workshop on Decentralized Finance (DeFi'22) (01 2022)
7. Baum, C., David, B., Dowsley, R.: Insured MPC: Efficient secure computation with financial penalties. In: FC 2020 (Feb 2020)
8. Baum, C., David, B., Dowsley, R., Nielsen, J.B., Oechsner, S.: CRAFT: Composable randomness and almost fairness from time. Cryptology ePrint Archive, Report 2020/784 (2020), https://eprint.iacr.org/2020/784
9. Baum, C., David, B., Frederiksen, T.K.: P2DEX: Privacy-preserving decentralized cryptocurrency exchange. In: ACNS 21, Part I (Jun 2021)
10. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy (May 2014)

11. Benhamouda, F., Halevi, S., Halevi, T.: Supporting private data on hyperledger fabric with secure multiparty computation. IBM Journal of Research and Development **63**(2/3), 3–1 (2019), https://doi.org/10.1147/JRD.2019.2913621
12. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: CRYPTO 2014, Part II (Aug 2014)
13. Bentov, I., Kumaresan, R., Miller, A.: Instantaneous decentralized poker. In: ASIACRYPT 2017, Part II (Dec 2017)
14. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. Journal of Cryptology (4) (Sep 2004)
15. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: ZEXE: Enabling decentralized private computation. In: 2020 IEEE Symposium on Security and Privacy (May 2020)
16. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. In: FC 2020 (Feb 2020)
17. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy (May 2018)
18. Camenisch, J., Lehmann, A., Lysyanskaya, A., Neven, G.: Memento: How to reconstruct your secrets from a single password in a hostile environment. In: CRYPTO 2014, Part II (Aug 2014)
19. Campanelli, M., Hall-Andersen, M.: Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In: ASIACCS 22 (May / Jun 2022)
20. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS (Oct 2001)
21. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001), https://doi.org/10.1109/SFCS.2001.959888
22. Canetti, R.: Universally composable signature, certification, and authentication. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA. p. 219. IEEE Computer Society (2004). https://doi.org/10.1109/CSFW.2004.24, http://doi.ieeecomputersociety.org/10.1109/CSFW.2004.24
23. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Theory of Cryptography Conference. pp. 61–85. Springer (2007), https://doi.org/10.1007/978-3-540-70936-7_4
24. Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., Song, D.: Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P) (2019). https://doi.org/10.1109/EuroSP.2019.00023
25. Choudhuri, A.R., Green, M., Jain, A., Kaptchuk, G., Miers, I.: Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In: ACM CCS 2017 (Oct / Nov 2017)
26. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: Hartmanis, J. (ed.) Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA. pp. 364–369. ACM (1986). https://doi.org/10.1145/12130.12168, https://doi.org/10.1145/12130.12168
27. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner ex-

tractable value, and consensus instability. In: 2020 IEEE Symposium on Security and Privacy (May 2020)
28. Damgård, I., Damgård, K., Nielsen, K., Nordholt, P.S., Toft, T.: Confidential benchmarking based on multiparty computation. In: FC 2016 (Feb 2016)
29. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits. In: European Symposium on Research in Computer Security. pp. 1–18. Springer (2013), https://doi.org/10.1007/978-3-642-40203-6_1
30. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO 2012 (Aug 2012)
31. David, B., Dowsley, R., Larangeira, M.: Kaleidoscope: An efficient poker protocol with payment distribution and penalty enforcement. In: FC 2018 (Feb / Mar 2018)
32. David, B., Gentile, L., Pourpouneh, M.: FAST: Fair auctions via secret transactions. In: ACNS 22 (Jun 2022)
33. da Gama, M.B., Cartlidge, J., Polychroniadou, A., Smart, N.P., Alaoui, Y.T.: Kicking-the-bucket: Fast privacy-preserving trading using buckets. Cryptology ePrint Archive (2021), to appear at FC'22. https://eprint.iacr.org/2021/1549
34. Ganesh, C., Orlandi, C., Pancholi, M., Takahashi, A., Tschudi, D.: Fiat-shamir bulletproofs are non-malleable (in the algebraic group model). In: EUROCRYPT 2022, Part II (May / Jun 2022)
35. Groth, J., Kohlweiss, M.: One-out-of-many proofs: Or how to leak a secret and spend a coin. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), https://doi.org/10.1007/978-3-662-46803-6_9
36. Groth, J., Ostrovsky, R., Sahai, A.: New techniques for noninteractive zero-knowledge. Journal of the ACM (JACM) **59**(3), 1–35 (2012), https://doi.org/10.1145/2220357.2220358
37. Jakobsen, T.P., Nielsen, J.B., Orlandi, C.: A framework for outsourcing of secure computation. In: Ahn, G., Oprea, A., Safavi-Naini, R. (eds.) Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014. pp. 81–92. ACM (2014). https://doi.org/10.1145/2664168.2664170
38. Kalodner, H.A., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: USENIX Security 2018 (Aug 2018)
39. Kanjalkar, S., Zhang, Y., Gandlur, S., Miller, A.: Publicly auditable mpc-as-a-service with succinct verification and universal setup. In: IEEE European Symposium on Security and Privacy Workshops, EuroS&P 2021, Vienna, Austria, September 6-10, 2021. pp. 386–411. IEEE (2021). https://doi.org/10.1109/EuroSPW54576.2021.00048, https://doi.org/10.1109/EuroSPW54576.2021.00048
40. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: Theory of Cryptography Conference. pp. 477–498. Springer (2013), https://doi.org/10.1007/978-3-642-36594-2_27
41. Kerber, T., Kiayias, A., Kohlweiss, M.: KACHINA - foundations of private smart contracts. In: CSF 2021 Computer Security Foundations Symposium (2021)
42. Khovratovich, D., Law, J.: Sovrin: digital identities in the blockchain era. Tech. rep., Sovrin Foundation, accessed: 2022-10-13
43. Kiayias, A., Zhou, H.S., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: EUROCRYPT 2016, Part II (May 2016)

44. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy (May 2016)
45. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: ACM CCS 2016 (Oct 2016)
46. Kumaresan, R., Moran, T., Bentov, I.: How to use bitcoin to play decentralized poker. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 195–206 (2015), https://doi.org/10.1145/2810103.2813712
47. Kumaresan, R., Vaikuntanathan, V., Vasudevan, P.N.: Improvements to secure computation with penalties. In: ACM CCS 2016 (Oct 2016)
48. Lee, J., Nikitin, K., Setty, S.T.V.: Replicated state machines without replicated execution. In: 2020 IEEE Symposium on Security and Privacy (May 2020)
49. Maram, D., Malvai, H., Zhang, F., Jean-Louis, N., Frolov, A., Kell, T., Lobban, T., Moy, C., Juels, A., Miller, A.: CanDID: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In: 2021 IEEE Symposium on Security and Privacy (May 2021)
50. Maxwell, G.: Confidential transactions. https://people.xiph.org/~greg/confidential_values.txt (2016)
51. Nilsson, A., Bideh, P.N., Brorsson, J.: A survey of published attacks on intel SGX. CoRR **abs/2006.13598** (2020), https://arxiv.org/abs/2006.13598
52. Noether, S.: Ring Signature Confidential Transactions for Monero. Cryptology ePrint Archive, Paper 2015/1098 (2015), https://eprint.iacr.org/2015/1098, https://eprint.iacr.org/2015/1098
53. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991). https://doi.org/10.1007/3-540-46766-1_9
54. Poelstra, A., Back, A., Friedenbach, M., Maxwell, G., Wuille, P.: Confidential assets. In: International Conference on Financial Cryptography and Data Security. pp. 43–63. Springer (2018), https://doi.org/10.1007/3-540-36178-2_26
55. Reistad, T.I., Toft, T.: Linear, constant-rounds bit-decomposition. In: Lee, D.H., Hong, S. (eds.) Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5984, pp. 245–257. Springer (2009). https://doi.org/10.1007/978-3-642-14423-3_17, https://doi.org/10.1007/978-3-642-14423-3_17
56. van Saberhagen, N.: CryptoNote v 2.0. https://web.archive.org/web/20201028121818/https://cryptonote.org/whitepaper.pdf (2013)
57. Sergio_Demian_Lerner: P2ptradex: P2p trading between cryptocurrencies. https://bitcointalk.org/index.php?topic=91843.0 (2012), accessed: 2022-10-13
58. Steffen, S., Bichsel, B., Baumgartner, R., Vechev, M.: ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1543–1543. IEEE Computer Society (2022), https://files.sri.inf.ethz.ch/website/papers/sp22-zeestar.pdf
59. Steffen, S., Bichsel, B., Gersbach, M., Melchior, N., Tsankov, P., Vechev, M.T.: zkay: Specifying and enforcing data privacy in smart contracts. In: ACM CCS 2019 (Nov 2019)
60. Team, T.S.N.: Secret network: A privacy-preserving secret contract & decentralized application platform. https://scrt.network/graypaper (2022)

A Extended related work

Privacy preserving transactions. The notion of confidential tokens was first proposed by Maxwell in [50], and subsequently formalized in [54]. Here, balances are concealed in additively homomorophic Pedersen commitments [53]. A valid transfer consumes and produces new commitments and is accompanied by a non-interactive range-proof [17] that each newly coin balance is below a threshold preventing the minting of tokens. Thus, if input and output commitments sum up, the ledger can verify that the coin supply was preserved. ZCash [10] breaks any relation between spent and newly minted tokens with general zkSNARK's proving both balance ranges and that confidential input coins have not previously been spent. Subsequent works [19] have improved on the complexity of NIZK's required. Zether [16] proposes a scheme which can be implemented on any public ledger with EVM-like smart contract machine, encrypting user balances with additive Elgamal encryption: importantly, for smaller balance ranges, this scheme permits the receiver to obtain the *opening* of privately received coins by brute-forcing an discrete-logarithm. Thus, unlike in other schemes, a sender does not need forward the private coin opening to the receiver. A weaker notion of privacy is achieved by Monero [56,52], which offers k-anonymity for senders.

Privacy preserving smart contracts. Hawk [44] introduces a general notion of smart contract evaluation over *private individual inputs* resulting in *private individual outputs*. The contract is evaluated over private numerical and private financial values, and is settled on a confidential ledger. Correctness and privacy is archived through the use of non-interactive zero-knowledge proofs. Thus, each party learns nothing about the inputs or outputs of other parties, other than what is implied by its own numerical and financial output. Although the notion of privacy here is limited to ledger and inter-party privacy, as a trusted third party is still required to carry out the private computation. This is quite different from what we normally consider private in the cryptographic community. Hence continuous research has been carried out, trying to remove this trusted third party, although it seems to require expensive, general non-interactive-zero-knowledge (NIZK) proofs to be computed inside the MPC function evaluation to permit private settlement on a confidential ledger. A recent line of work has by Banerjee *et al.* [4,5] contributes concrete efficiency improvements in realizing private smart contract notion of Hawk by realizing the trusted party using MPC, by reducing the complexity of NIZK's which are then needs to be computed inside the MPC. Recently, Kanjalkar et al. [39] presented an optimized ZK protocol to be proven inside MPC.

Zexe [15] extends the ZCash model of private transactions with a private, state-less contract model. Bitcoin Script-like contracts are hidden, as are contract inputs, enabling limited contracting functionality with privacy.

A notion of smart contracts with *data privacy* is proposed in zkay [59]. Here, contract execution on the public ledger is verified over *private data* encrypted with a fully homomorphic encryption (FHE) scheme thereby shielding selected fragments of the contract state. Follow-up work Zeestar [58] adds support for non-interactive zero-knowledge proofs to increase the expressiveness of supported contracts. However, this line of work does not explicitly provide privacy between contract participants: a party which holds the FHE encryption key can trivially observe the contract evaluation in the clear.

Fair MPC with public ledgers. We describe two closely related lines of work that integrate MPC protocols with a ledger functionality to achieve (1) fair MPC protocols, which identify and penalize cheating parties and (2) private smart contracts executed inside a MPC instance which finalize the *confidential* outcome on the ledger.

The first works to utilize the Bitcoin ledger to achieve fairness in lottery games was introduced in [1,2], where cheating parties can abort upon learning the output first but incur a financial penalty without requiring a trusted party to arbitrate. This notion of output fairness was generalized to any secure function evaluation by Bentov *et al.* [12] and to the reactive setting [46]. Subsequent works improve the efficiency of output fairness [45,47,13,11,31], culminating in Insured MPC [7], which formulates a UC-secure MPC functionality with identifiable aborts. Another line of work [43,25] focuses on stronger notions of fairness, identifying aborting parties' prior to the output phase.

P2DEX. At ANCS 2021 Baum *et al.* [9] introduced P2DEX, which extends Insured MPC [7] by allowing for cross-chain communication and privacy preserving smart contract computation. Although their privacy preservation only involves auxiliary input, and *not* the token amounts. Their idea is to first have user send the tokens to a burner address, generated in a distributed manner by a set of servers. These servers then run an Insured MPC protocol which computes a private smart contract based on auxiliary input privately given by the users. Based on the result, appropriate amount of tokens can be transferred from the burner addresses to the users by having the servers threshold sign these transactions. The authors use this to make a system for decentralized cross-chain exchange, preventing *both* miner and operator front-running.

B Extended preliminaries

Pedersen commitments. Let g, h denote random generators of $\mathbb{G}$ such that nobody knows the discrete logarithm of h base g, i.e., a value w such that $g^w = h$. The Pedersen commitment scheme [53] to an $s \in \mathbb{Z}_p$ is obtained by sampling $t \leftarrow\$ \mathbb{Z}_p$ and computing $\mathsf{com}(s,t) = g^s h^t$. Hence, the commitment $\mathsf{com}(s,t)$ is a value uniformly distributed in $\mathbb{G}$ and opening the commitment requires to reveal the values of s and t. The Pedersen commitments are additively homomorphic, i.e., starting from the commitment to $s_1 \in \mathbb{Z}_p$ and $s_2 \in \mathbb{Z}_p$, it is possible to compute a commitment to $s_1 + s_2 \in \mathbb{Z}_p$, i.e., $\mathsf{com}(s_1, t_1) \circ \mathsf{com}(s_2, t_2) = \mathsf{com}(s + 1 + s_2, t_1 + t_2)$.

(Global) Universal Composability. In this work, the (Global) Universal Composability or (G)UC framework [21,23] is used to analyze security. Due to space constraints, we refer interested readers to the aforementioned works for more details. We generally use $\mathcal{F}$ to denote an ideal functionality and Π for a protocol. We implicitly assume private and authenticated channel between each pair of parties.

Several functionalities in this work allow public verifiability. Following Badertscher *et al.* [3] we dynamically allow the construction of a set of verifiers $\mathcal{V}$ through register and de-register commands. The adversary, $\mathcal{S}$ will always be allowed to obtain the list of registered verifiers. Concretely we implicitly assume all functionalities with public verifiability include the following interfaces (which are omitted in the concrete boxes for simplicity):

Register: Upon receiving (REGISTER, sid) from some verifier $\mathcal{V}_i$, set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return (REGISTERED, $sid, \mathcal{V}_i$) to $\mathcal{V}_i$.

Deregister: Upon receiving (Deregister,sid) from some verifier $\mathcal{V}_i$, set $\mathcal{V} = \mathcal{V} \setminus \mathcal{V}_i$ and return (DEREGISTERED, $sid, \mathcal{V}_i$) to $\mathcal{V}_i$.

Is Registered: Upon receiving (IS-REGISTERED, sid) from $\mathcal{V}_i$, return (IS-REGISTERED, sid, b) to $\mathcal{V}_i$, where $b = 1$ if $\mathcal{V}_i \in \mathcal{V}$ and $b = 0$ otherwise.

Get Registered: Upon receiving (GET-REGISTERED, sid) from the ideal adversary $\mathcal{S}$, the functionality returns (GET-REGISTERED, $sid, \mathcal{V}$) to $\mathcal{S}$. The above instructions can also be used by other functionalities to register as a verifier of a publicly verifiable functionality.

Global clock. As some parts of our work are inherently synchronous, we model rounds using a global clock functionality $\mathcal{F}_{\mathsf{Clock}}$ as in [3,40,43]. In Fig. 3 we show the global UC clock functionality, $\mathcal{F}_{\mathsf{Clock}}$, we need, taken verbatim from the work of Baum *et al.* [9]. We note that in the real execution all parties will send messages to, and receive them, from $\mathcal{F}_{\mathsf{Clock}}$. Whereas in the simulated case only the ideal functionality, other global functionalities as well as the corrupted parties will do so. Throughout this work, we will write "update $\mathcal{F}_{\mathsf{Clock}}$" as a short-hand for "send (UPDATE, sid) to $\mathcal{F}_{\mathsf{Clock}}$".

Functionality $\mathcal{F}_{\mathsf{Clock}}$

$\mathcal{F}_{\mathsf{Clock}}$ is parameterized by a variable ν, sets $\mathcal{P}$,$\mathcal{F}$ of parties and functionalities respectively. It keeps a Boolean variable d_J for each $J \in \mathcal{P} \cup \mathcal{F}$, a counter ν as well as an additional variable u. All d_J, ν and u are initialized as 0.

Clock update: Upon receiving a message (UPDATE) from $J \in \mathcal{P} \cup \mathcal{F}$:
1. Set $d_J = 1$.
2. If $d_F = 1$ for all $F \in \mathcal{F}$ and $d_P = 1$ for all honest $P \in \mathcal{P}$, then set $u \leftarrow 1$ if it is 0.

Clock read: Upon receiving a message (READ) from any entity:
1. If $u = 1$ then first send (TICK, sid) to $\mathcal{S}$. Next set $\nu \leftarrow \nu + 1$, reset d_J to 0 for all $J \in \mathcal{P} \cup \mathcal{F}$ and reset u to 0.
2. Answer the entity with (READ, ν).

Fig. 3: Global UC functionality $\mathcal{F}_{\mathsf{Clock}}$ for the clock.

Signatures. We will implicitly assume access to a global digital signature ideal functionality $\mathcal{F}_{\mathsf{Sig}}$ as defined in [22] (where it is also shown any EUF-CMA signature scheme realizes it), which is used for signing transactions to a ledger. We also use a global UC secure threshold signature scheme which offers identifiable abort. We denote this functionality $\mathcal{F}_{\mathsf{TSig}}$ and define it in Fig. 4 (which is taken verbatim from the work of Baum *et al.* [9]). The functionality allows a set of n parties to collaboratively sign a message m, and allows the adversary to corrupt up to $n-1$ parties without being able to forge signatures. That is, we assume the full-threshold setting. Thus its behaviour matches that of $\mathcal{F}_{\mathsf{Sig}}$, although it additionally allows $\mathcal{S}$ to choose the string of shares that later get combined into a signature. Although under the constraint that $\mathcal{S}$ has to choose both the signature shares and the actual signature, σ, together. Although this allows $\mathcal{S}$ to always make a valid signature, it is never allowed to make an invalid signature in an honest execution of **Share Generation**. Based on the signature shares, the parties can learn σ from **Share Combination**, although if parties have been cheating in **Shares Generation** they will be exposed during **Share Combination**. We observe that the choice of shares binds $\mathcal{S}$ to a certain set of dishonest parties. Note that by assuming both $\mathcal{F}_{\mathsf{Sig}}$ and $\mathcal{F}_{\mathsf{TSig}}$ to be *global* UC functionalities, it allows other UC functionalities, both local and global, to verify signatures on them. This becomes essential to allow interaction with our, global, ledger functionalities.

Non-interactive zero-knowledge. We us non-interactive zero-knowledge arguments of knowledge, allowing any party to construct a proof that can later be validated by any verifier. We model this in the same way as done by Groth *et al.* [36] and formally define the functionality $\mathcal{F}_{\mathsf{NIZK}}$ for this in Fig. 5 in Sec. B. The functionality $\mathcal{F}_{\mathsf{NIZK}}$ allows any party to prove in zero knowledge that they know a witness w for a public statement x such that $(x, w) \in R$ for a NP relation R.

Functionality $\mathcal{F}_{\mathsf{TSig}}$

$\mathcal{F}_{\mathsf{TSig}}$ is parameterized with an ideal adversary $\mathcal{S}$, a set of signers $\mathcal{P}$ and functionalities $\mathcal{F}$, a verifiers $\mathcal{V}$ (which automatically contains $\mathcal{P}$ and $\mathcal{F}$) and a set of corrupted signers $I \subset \mathcal{P}$. $\mathcal{F}_{\mathsf{TSig}}$ has two internal lists `Sh` and `Sig`.

Key Generation: Upon receiving a message $(keygen)$ from each $\mathcal{P}_i \in \mathcal{P}$ or a functionality $\mathcal{F}_j \in \mathcal{F}$ hand $(keygen)$ to the adversary $\mathcal{S}$. Upon receiving $(verificationkey)\mathsf{vk}$ from $\mathcal{S}$, if $(\cdot, \mathsf{vk})$ was not recorded yet then output $(verificationkey)\mathsf{vk}$ to each $\mathcal{P}_i \in \mathcal{P}$ (or to $\mathcal{F}_j$), and record the pair $(\mathcal{P}, \mathsf{vk})$. If vk was recorded before then output $(Abort)$ to $\mathcal{S}$ and stop.

Share Generation: Upon receiving a message $(sign)m, \mathsf{vk}$ from all honest parties or a functionality $\mathcal{F}_j \in \mathcal{F}$ send $(sign)m$ to $\mathcal{S}$. Upon receiving $(signature)m, \rho, \sigma, J, f$ from $\mathcal{S}$, verify that
- no entry $(m, \rho, J', \mathsf{vk}')$ with $J' \neq J$ is recorded in `Sh`, and
- no entry $(m, \sigma, \mathsf{vk}, 0)$ is recorded in `Sig` if $J = \emptyset$.

If either is, then output an error message to $\mathcal{S}$ and halt. Else, let $f' = 1$ if $J = \emptyset$ and $f' = f$ otherwise, record the entry $(m, \rho, J, \mathsf{vk})$ in `Sh`, $(m, \sigma, \mathsf{vk}, f')$ in `Sig` and return $(shares)m, \rho$.

Share Combination: Upon receiving a message $(combine)m, \rho, \mathsf{vk}$ from any party in $\mathcal{P}$ or functionality $\mathcal{F}_j \in \mathcal{F}$, find $(m, \rho, J, \mathsf{vk})$ in `Sh` and $(m, \sigma, \mathsf{vk}, b)$ in `Sig`. If $J \neq \emptyset$ then return $(Failure)J$. If $J = \emptyset$ return $(combined)m, \sigma, \mathsf{vk}$. If no entry could be found in `Sh` and `Sig` then return $(Not - Generated)$.

Signature Verification: Upon receiving a message $(verify)m, \sigma, \mathsf{vk}'$ from some entity in $\mathcal{V}$, hand $(verify)m, \sigma, \mathsf{vk}'$ to $\mathcal{S}$. Upon receiving $(verified)m, \phi$ from $\mathcal{S}$ do:
1. If $\mathsf{vk}' = \mathsf{vk}$ and $(m, \sigma, \mathsf{vk}, 1) \in$ `Sig`, then set $f = 1$.
2. Else, if $\mathsf{vk}' = \mathsf{vk}$ and $(m, \sigma', \mathsf{vk}, 1) \notin$ `Sig` for any σ', then set $f = 0$ and record the entry $(m, \sigma, \mathsf{vk}, 0)$ in `Sig`.
3. Else, if there is an entry $(m, \sigma, \mathsf{vk}', f') \in$ `Sig` recorded, then let $f = f'$.
4. Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{vk}', \phi)$ in `Sig`.

Return $(verified)m, f$.

Fig. 4: Global UC functionality $\mathcal{F}_{\mathsf{TSig}}$ for Threshold Signatures.

Functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$

$\mathcal{F}_{\mathsf{NIZK}}$ interacts with parties $\mathcal{P} = \{P_1, ..., P_n\}$ and simulator $\mathcal{S}$ and is parameterized with relation R.

Proof: On input $(\textsc{Prove}, sid, x, w)$ from party P, ignore if $(x, w) \notin R$. Send $(\textsc{Prove}, x)$ to $\mathcal{S}$ and wait for answer $(\textsc{Proof}, \pi)$. Upon receiving the answer store (x, π) and send $(\textsc{Proof}, sid, \pi)$ to P.

Verification: On input $(\textsc{Verify}, sid, x, \pi)$ from $\mathcal{V}$, check whether (x, π) is stored. If not send $(\textsc{verify}, x, \pi)$ to $\mathcal{S}$ and wait for an answer $(\textsc{Witness}, w)$. Upon receiving the answer, check whether $(x, w) \in R$ and in that case, store (x, π). If (x, π) has been stored, return $(\textsc{Verification}, sid, 1)$ to $\mathcal{V}$, else return $(\textsc{Verification}, sid, 0)$.

Fig. 5: UC functionality $\mathcal{F}_{\mathsf{NIZK}}$ for Non-interactive Zero-Knowledge

MPC. Secure Multi-Party Computation (MPC) allows a set of mutually distrusting $\mathcal{P} = \{P_1, \ldots, P_n\}$ to compute any efficiently commutable function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ where each party P_i supplied private input x_i and received private output y_i. MPC guarantees that the only thing known to party P_i after the computation is x_i and y_i. Multiple security and computational models exist for this, but in this paper we will assume the arithmetic black box model, where computation is a directed acyclic graph of arithmetic operations over a finite field $\mathbb{F}$, where $|\mathbb{F}| = p \geq 2^s$. We assume the UC-security against a *static*, *active/malicious* adversary, who can corrupt up to $n-1$ parties and who may cause an abort at any point in the computation. We assume an MPC scheme, which is *reactive*, meaning that it is possible to compute $f(\cdot)$,and depending on the output, compute some other function $f'(\cdot)$ on the same input as $f(\cdot)$. This model can for example be realized by the SPDZ protocol [30]. For simplicity we will assume the bracket-notation, where the function to be computed is specified by arithmetic operations on hidden variables. Concretely we assume $[\cdot]$ expresses a value hidden in MPC and on which arithmetic computations can be carried out. I.e. $[x] \cdot [y] + [z]$, expresses the computation $x \cdot y + z$ of values $x, y, z \in \mathbb{F}$.

Outsourced MPC. Typically MPC in the setting we need require a non-constant amount of rounds of communication between *all* pairs of parties (depended on the function to compute). If we have many parties supplying input this can become prohibitively expensive. For this reason we introduce another model of MPC known as *outsourced MPC*. Jakobsen *et al.* [37] shows how to use information theoretic operation in conjunction with *any* MPC scheme as described above, to allow a large set of clients $\mathcal{C} = \{C_1, \ldots, C_m\}$ to supply private input to an MPC computation, executed by a small set of servers $\mathcal{P} = \{P_1, \ldots, P_n\}$, and receive private output. Crucially the clients only need to execute a few lightweight operations, bounded by their amount of inputs and outputs, and only need to communicate with the servers in a constant amount of rounds.

Insured MPC. It has been shown [26] that it is impossible to achieve *fairness* in MPC when more than $n/2$ of the parties are corrupted. By fairness we mean that if one party learns their output of the computation, so does the rest of the parties. This is a problem since the party learning the output may be malicious and thus maybe abort the protocol based on what they learned. Baum *et al.* [7] show how to incentivize the completion of an MPC protocol, in a public verifiable manner, through financial incentives enforced on a public ledger. Specifically they showed this is possible to do, based on any MPC scheme fitting the model discussed above. We combine this incentivized notion of MPC with the outsourced notion of MPC in the functionality $\mathcal{F}_{\mathsf{Ident}}$. Concretely this specifies an out-sourced MPC functionality where clients $\mathcal{C} = \{C_1, \ldots, C_m\}$ supply private input that is computed on in MPC by the servers $\mathcal{P} = \{P_1, \ldots, P_n\}$ and where the output of the computation is verifiably shared between the servers in such a manner that the shares can verified by an external verifier $\mathcal{V}$ after the completion of the protocol to identify any potential malicious behaviour. We refer to appendix B.1 for a detailed description of $\mathcal{F}_{\mathsf{Ident}}$ and its interaction with servers and clients.

B.1 Publicly Verifiable MPC Functionality $\mathcal{F}_{\mathsf{Ident}}$

We adopt $\mathcal{F}_{\mathsf{Ident}}$ from [7,9] but include the following extensions to its interface. Firstly, when realizing $\mathcal{F}_{\mathsf{Ident}}$ with a reactive MPC scheme such as [30,29], we can amend $\mathcal{F}_{\mathsf{Ident}}$ with a reactive interface as in $\mathcal{F}_{\mathsf{Online}}$ from [29], exposing arithmetic operations over secret values, each identified with a unique *vid*, which are *selectively* input or output to the parties.

In addition to a reactive interface, we permit clients to *securely input* values to $\mathcal{F}_{\mathsf{Ident}}$. This is realized with the secure client input protocol from [37], which permits MPC servers to verify the linear MAC of the client input inside a reactive MPC instance. We wrap this secure client input protocol inside $\mathcal{F}_{\mathsf{Ident}}$ (as in the security proof of [37]) to obtain an input interface which can be called by clients.

Theorem 3. *Functionality $\mathcal{F}_{\mathsf{Ident}}$ (in Figure 6) can be realized by a reactive secure computation scheme permitting linear operations for free and the secure client input protocol from [37].*

Proof. (Proof sketch) The original, non-interactive $\mathcal{F}_{\mathsf{Ident}}$ functionality from [7] can be extended with a reactive interface when realized with a reactive MPC scheme [30,29] with minor adaptations of the UC-proof in [7]. The secure client input protocol of [37] is information-theoretically secure, and can be instantiated with *any* MPC scheme with free linear operations, including [30,29], thereby realizing reactive and client input interfaces of $\mathcal{F}_{\mathsf{Ident}}$ in Figure 6.

Identifiable aborts during the output phase. We provide an overview of the execution of a generic protocol π in the $\mathcal{F}_{\mathsf{Ident}}$-hybrid setting, where π can either obtain the output of an MPC secure evaluation on private client inputs performed by an $\mathcal{F}_{\mathsf{Ident}}$ instance, or identify cheating parties.

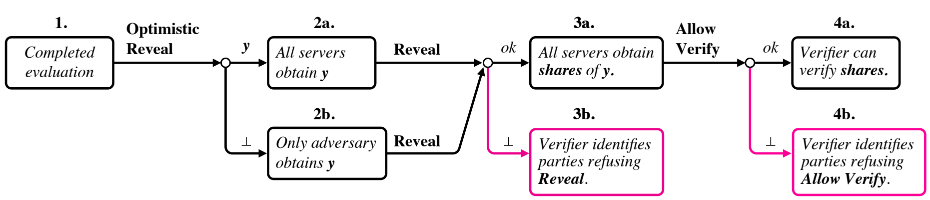


Fig. 7: Output phase of $\mathcal{F}_{\mathsf{Ident}}$ with verifiable output.

Following the secure evaluation on the client inputs, honest parties of π perform the following. Upon sending **Optimistic Reveal** to $\mathcal{F}_{\mathsf{ident}}$, honest parties

Functionality $\mathcal{F}_{\mathsf{Ident}}$

For each session, $\mathcal{F}_{\mathsf{Ident}}$ interacts with servers $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, clients $\mathcal{C} = \{\mathcal{C}_1, \dots, \mathcal{C}_m\}$ and also provides an interface to register external verifiers $\mathcal{V}$. $\mathcal{S}$ provides a set $I_{\mathcal{P}} \subset [n]$ of corrupt parties and $I_{\mathcal{C}} \subseteq [m]$ of corrupt clients. $\mathcal{F}_{\mathsf{Ident}}$ only interacts with $\mathcal{P}, \mathcal{C}, \mathcal{V}$ and $\mathcal{S}$[a] of the respective session *sid*.

Init: Upon first $(\textsc{Init}, sid)$ by all parties in $\mathcal{P}$ set $\texttt{rev}, \texttt{ver}, \texttt{ref} \leftarrow \emptyset$.
Input: Upon first $(\textsc{Input}, sid, x)$ by $\mathcal{C}_j$, forward $(\textsc{Input}, sid, \mathcal{C}_j)$ to $\mathcal{S}$. If $\mathcal{S}$ continues, $\mathcal{F}_{\mathsf{Ident}}$ samples vid, stores (vid, x) and returns $(\textsc{Input}, sid, \mathcal{C}_j, vid)$ to all $\mathcal{P}$.
Evaluate: Upon first $(\textsc{Eval}, sid, g, vid_1, ..., vid_p)$ by all parties $\mathcal{P}$, if $(vid_1, ..., vid_p)$ have been stored internally:
1. Compute $x_{p+1}, ..., x_{p+q} \leftarrow g(x_1, ..., x_p)$, sample $vid_{p+1}, ..., vid_{p+q}$.
2. Store $(vid_{p+1}, x_{p+1}), ..., (vid_{p+q}, x_{p+q})$, return $(vid_{p+1}, ..., vid_{p+q})$ to all parties.

Get Shares: Upon first $(\textsc{GetShare}, sid, vid)$ by $\mathcal{P}_i \in \mathcal{P}$ and if (vid, x) is stored:
1. For $\mathcal{P}_i \in I_{\mathcal{P}}$, let $\mathcal{S}$ provide $s_{vid}^{(i)} \in \mathbb{F}$. For $\mathcal{P}_i \in \overline{I_{\mathcal{P}}}$, let $s_{vid}^{(i)} \xleftarrow{\$} \mathbb{F}$ s.t. $x = \sum_{i \in [n]} s_{vid}^{(i)}$.
2. Return $(\textsc{Share}, sid, vid, s_{vid}^{(i)})$ to P_i.

Open with identifiable abort: All interfaces below are specific to a $(vid, \cdot)$.
Share: Upon first $(\textsc{Share}, sid, vid)$ by $\mathcal{P}_i \in \mathcal{P}$ and if (vid, x) is stored, sample shares as in **Get Shares** if not previously done and and store locally.
Optimistic Reveal: Upon $(\textsc{Optimist-Open}, sid, vid)$ by each honest $\mathcal{P}_i$ and if **Share** for (vid, x) was run, then send $(\textsc{Output}, sid, vid, x)$ to $\mathcal{S}$. If $\mathcal{S}$ continues, send $(\textsc{Output}, sid, vid, x)$ to each honest $\mathcal{P}_i$, otherwise send $(\textsc{Output}, sid, vid, \perp)$.
Reveal: Upon $(\textsc{Reveal}, sid, vid)$ by $\mathcal{P}_i$, if $i \notin \texttt{rev}[vid]$ send $(vid, i, s_{vid}^{(i)})$ to $\mathcal{S}$.
1. If $\mathcal{S}$ continues, set $\texttt{rev}[vid] \leftarrow \texttt{rev}[vid] \cup \{i\}$, send $(\textsc{Reveal}, sid, i, s_{vid}^{(i)})$ to all $\mathcal{P}$.
2. Else if $\mathcal{S}$ sends $(\textsc{Reveal-Not-Ok}, sid, vid, i, J)$ with $J \subseteq I_{\mathcal{P}}, J \neq \emptyset$, send $(\textsc{Reveal-Fail}, sid, vid, i)$ to all $\mathcal{P}$ and set $\texttt{ref}[vid] \leftarrow \texttt{ref}[vid] \cup J$.

Test Reveal: Upon $(\textsc{Test-Reveal}, sid, vid)$ from a party in $\mathcal{P} \cup \mathcal{V}$
1. If $\texttt{ref}[vid] \neq \emptyset$, return $(\textsc{Reveal-Fail}, sid, vid, \texttt{ref}[vid])$
2. Otherwise return $(\textsc{Reveal-Fail}, sid, vid, [n] \setminus \texttt{rev}[vid])$.

Allow Verify: Upon $(\textsc{Start-Verify}, sid, vid, i)$ from party $\mathcal{P}_i \in \mathcal{P}$ set $\texttt{ver}[vid] \leftarrow \texttt{ver}[vid] \cup \{i\}$. If $\texttt{ver}[vid] = [n]$ then deactivate all interfaces for *vid* except **Test Reveal** and **Verify**.
Verify: Upon $(\textsc{Verify}, sid, vid, z^{(1)}, \dots, z^{(n)})$ by $\mathcal{V}_i \in \mathcal{V}$ with $z^{(j)} \in \mathbb{F}$:
1. If $\texttt{ver}[vid] \neq [n]$ then return $(\textsc{Verify-Fail}, sid, vid, [n] \setminus \texttt{ver}[vid])$.
2. Else if $\texttt{ver}[vid] = [n]$ and $\texttt{rev}[vid] \neq [n]$, send to $\mathcal{V}_i$ what **Test Reveal** sends.
3. Else set $\texttt{ws} \leftarrow \{j \in [n] \mid z^{(j)} \neq s_{vid}^{(j)}\}$ and return $(\textsc{Open-Fail}, sid, vid, \texttt{ws})$.

[a] Throughout **Init**, **Input**, **Evaluate** and **(Get) Share**, $\mathcal{S}$ can at any point abort, upon which $\mathcal{F}_{\mathsf{Ident:rct}}$ sends $(\textsc{Abort}, \perp)$ to all parties and terminates.

Fig. 6: UC functionality $\mathcal{F}_{\mathsf{Ident}}$ for reactive MPC with Publicly Verifiable Output.

will either (2a) obtain the output $\mathbf{y} = (y_1, ..., y_m)$, where y_i denotes the output for client $C_i \in \mathcal{C}$, or (2b) only the adversary obtains $\mathbf{y}$. In either state (2a)/(2b), the honest parties of π can *always* reach a state of $\mathcal{F}_{\mathsf{Ident}}$ via **Reveal** and **Allow Verify**, in which either (4a) shares $\mathbf{s}^{(i)}$ for each server $P_i \in \mathcal{P}$ are received by all parties, that are verifiable by by $\mathcal{V}$ and from which $\mathbf{y} = \sum_{i\in[n]} \mathbf{s}^{(i)}$ can be reconstructed, or (3b)/(4b) the verifier $\mathcal{V}$ can identify cheating servers.

The interfaces of $\mathcal{F}_{\mathsf{Ident}}$ exposed to the public verifier $\mathcal{V}$ are central to the arbitration of an abort in protocol π. Here, a smart contract playing the role of public verifier $\mathcal{V}$ can identify the set of cheating servers therefore enforce a financial penalty agreed upon prior to the adversary learning $\mathbf{y}$.

C Ledger Functionalities

C.1 Public ledger functionality

In Figure 8 we describe the ideal functionality $\mathcal{F}_{\mathsf{Ledger}}$. It reflects a general public ledger, with the support for transfers of tokens through signatures, along with Turing complete smart contracts, modeled as arithmetic circuits over $\mathbb{F}$. It requires access to the global UC functionalities of $\mathcal{F}_{\mathsf{Clock}}$, for a notion of rounds, and $\mathcal{F}_{\mathsf{(T)Sig}}$ for signature validation.

Beyond its authenticated bulletin board functionality, on which it is based, $\mathcal{F}_{\mathsf{Ledger}}$ parses all the newly received, signed messages and updates its public state accordingly on the first activation of each $\mathcal{F}_{\mathsf{Clock}}$ round. In addition to authenticated messages, we define its public state to include a public ledger over a default token universe, maintaining balances associated with each signature verification key observed in the authenticated message list. Furthermore, $\mathcal{F}_{\mathsf{Ledger}}$ will maintain public state of *smart contracts instances*, each deployed with a transition function encoded as arithmetic circuits.

Interaction with UC functionalities. We permit smart contracts deployed to $\mathcal{F}_{\mathsf{Ledger}}$ to pass messages to external UC functionalities. This is required in order for a smart contract instance to evaluate the verification of proofs generated by a $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ instance or shares output by $\mathcal{F}_{\mathsf{Ident}}$. Interaction in the GUC model is permitted for global functionality $\mathcal{F}_{\mathsf{Ledger}}$ and other global UC functionalities, such as $\mathcal{F}_{\mathsf{Clock}}$. However, lifting the model of $\mathcal{F}_{\mathsf{Ident}}$ or $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ to global functionalities greatly complicates the definition of any functionality realized in the $\mathcal{F}_{\mathsf{Ident}}$, $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$-hybrid setting, as the simulator can no longer equivocate outputs from $\mathcal{F}_{\mathsf{Ident}}$ without simulating its internal state as a hybrid functionality, and extraction of a global $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ would imply a realization by less efficient constructions.

Although we do not model consensus details with $\mathcal{F}_{\mathsf{Ledger}}$, we argue that such a protocol must ultimately realized in the presence of an honest majority committee. Thus, we adopt this assumption with an honest-majority committee of dummy parties $\mathcal{Q} = \{Q_1, ..., Q_q\}$ interacting with $\mathcal{F}_{\mathsf{Ledger}}$, that *forward* verification calls between $\mathcal{F}_{\mathsf{Ledger}}$ and the environment $\mathcal{Z}$. Concretely, we permit the deployed contract circuits to feature *call-back* gates, which indicate an external functionality and message $(\mathcal{F}, \mathsf{m})$ that is forwarded to $\mathcal{Q}$ by $\mathcal{F}_{\mathsf{Ledger}}$.

- Upon receiving $(\textsc{ExtCall}, sid, \mathcal{F}, \mathsf{m})$ from $\mathcal{F}_{\mathsf{Ledger}}$, an honest party in $\mathcal{Q}$ then returns this message to $\mathcal{Z}$ and waits for a response.

Functionality $\mathcal{F}_{\mathsf{Ledger}}$

$\mathcal{F}_{\mathsf{Ledger}}$ interacts with global functionalities $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Sig}}$, $\mathcal{F}_{\mathsf{TSig}}$. It is parameterized by a token universe $\mathbb{T} = \{\tau_0, ..., \tau_t\}$ and maintains public ledger state $\mathcal{L} : \{0,1\}^* \rightarrow \mathbb{Z}^{|\mathbb{T}|}$, public contract states $\Gamma : \mathbb{Z} \rightarrow \{0,1\}^*$ and a contract counter $\mathsf{ctr} \in \mathbb{Z}$. $\mathcal{F}_{\mathsf{Ledger}}$ has an initially empty list $\mathcal{M}$ of messages posted to the authenticated bulletin board. Further, it interacts with committee parties $\mathcal{Q} = \{Q_1, ..., Q_q\}$.

Upon each activation $\mathcal{F}_{\mathsf{Ledger}}$ first sends a message $(\textsc{Read}, sid)$ to $\mathcal{F}_{\mathsf{Clock}}$. If ν has changed since the last call to $\mathcal{F}_{\mathsf{Clock}}$, then it parses all messages in $\mathcal{M}' \leftarrow \mathcal{M} \setminus \mathcal{M}_{\mathsf{read}}$ in listed order as follows:

Init: Upon parsing $((\textsc{Init}, sid, \mathcal{L}^{\mathsf{init}}), \mathsf{vk}) \in \mathcal{M}'$, set $\mathcal{L} \leftarrow \mathcal{L}^{\mathsf{init}}$, deactivate the parsing of $\textsc{Init}$ messages and activate parsing of $\textsc{Transfer}, \textsc{Deploy}, \textsc{Call}$ messages.

Transfer: Upon parsing $((\textsc{Transfer}, sid, \bar{\mathbf{v}}, \mathsf{vk}_{\mathsf{rcv}}), \mathsf{vk}) \in \mathcal{M}'$, assert $\mathcal{L}[\mathsf{vk}] \geq \bar{\mathbf{v}}$. Set $\mathcal{L}[\mathsf{vk}_{\mathsf{rcv}}] \leftarrow \mathcal{L}[\mathsf{vk}_{\mathsf{rcv}}] + \bar{\mathbf{v}}$ and $\mathcal{L}[\mathsf{vk}] \leftarrow \mathcal{L}[\mathsf{vk}] - \bar{\mathbf{v}}$.

Deploy Contract: Upon parsing $((\textsc{Deploy}, sid, \gamma, T), \mathsf{vk}) \in \mathcal{M}'$, parse initial state $\gamma \in \{0,1\}^*$ and T as an arithmetic circuit over $\mathbb{F}$ of maximum depth d_T. Update $\Gamma \leftarrow \Gamma \cup \{(\mathsf{cn} = \mathsf{ctr}, (\gamma, \bar{\mathbf{0}}, T))\}$ and contract id counter $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$.

Call Contract: Upon parsing $((\textsc{Call}, sid, (\mathsf{cn}, \mathsf{fn}, x, \bar{\mathbf{v}})), \mathsf{vk}) \in \mathcal{M}'$, assert $\mathcal{L}[\mathsf{vk}] \geq \bar{\mathbf{v}}$. Read $\mathcal{F}_{\mathsf{Clock}}$ round ν, obtain contract state, balances and circuit $(\gamma, \bar{\mathbf{w}}, T) \leftarrow \Gamma[\mathsf{cn}]$ and evaluate circuit T on inputs $(\nu \mid \gamma \mid \bar{\mathbf{w}} \mid \mathsf{cn}, \mathsf{fn}, x, \bar{\mathbf{v}} \mid \mathsf{vk})$, which outputs an encoding of a state transition ts, $\mathcal{L} \mid \gamma \mid \bar{\mathbf{w}} \rightarrow^{\mathsf{ts}} \mathcal{L}' \mid \gamma' \mid \bar{\mathbf{w}}'$, updating ledger, contract state and balance.

1. T is permitted "callback" gates which specify:
   a. Fragments of state $\mathcal{L} \mid \Gamma$ to read during evaluation of T.
   b. External functionality $\mathcal{F}$ and message m to send:
      - Forward $(\textsc{ExtCall}, sid, \mathcal{F}, \mathsf{m})$ to all $\mathcal{Q}$ who output this forwarded message.
      - Upon $(\textsc{CallBack}, sid, \mathsf{ret})$ from each $\mathcal{Q}$, write replies to call-back gate; the majority response from $\mathcal{Q}$ is read by the circuit upon continuation.
2. On completed evaluation of T, assert:
   a. Preservation of tokens: $\bar{\mathbf{w}} + \sum_{\mathsf{vk} \in \mathsf{dom}(\mathcal{L})} \mathcal{L}[\mathsf{vk}] = \bar{\mathbf{w}}' + \sum_{\mathsf{vk} \in \mathsf{dom}(\mathcal{L})} \mathcal{L}'[\mathsf{vk}]$
   b. No outflow from non-calling accts: $\forall \mathsf{vk}' \in \mathsf{dom}(\mathcal{L}), \mathsf{vk}' \neq \mathsf{vk} : \mathcal{L}'[\mathsf{vk}'] \geq \mathcal{L}[\mathsf{vk}']$.
3. Set $\mathcal{L} \leftarrow \mathcal{L}'$ and $\Gamma[\mathsf{cn}] \leftarrow (\gamma', \bar{\mathbf{w}}', T)$.

After parsing $\mathcal{M}'$, it sets $\mathcal{M}_{\mathsf{read}} \leftarrow \mathcal{M}$ and sends $(\textsc{Update}, sid)$ to $\mathcal{F}_{\mathsf{Clock}}$.

Post: Upon receiving $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma)$ from some entity contact the instance of $\mathcal{F}_{\mathsf{Sig}}$ or $\mathcal{F}_{\mathsf{TSig}}$ belonging to vk. If σ verifies for m and vk then send $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma)$ to $\mathcal{S}$ and append (m, vk) to the list $\mathcal{M}$.

Read: Upon receiving $(\textsc{Read}, sid)$ from some entity, return $\mathcal{M}$.

Fig. 8: Functionality $\mathcal{F}_{\mathsf{Ledger}}$ for Public Ledger and Smart Contracts.

- Upon input $(\textsc{CallBack}, sid, \mathcal{F}, \mathsf{ret})$ from $\mathcal{Z}$ to the same party $Q \in \mathcal{Q}$, it forwards this message to $\mathcal{F}_{\mathsf{Ledger}}$, which writes the majority response to the call-back gate.

The utility of forwarding $(\mathcal{F}, \mathsf{m})$ to the environment via dummy parties $\mathcal{Q}$ becomes immediate in the $\mathcal{F}$, $\mathcal{F}_{\mathsf{Ledger}}$-hybrid setting: here, the parties in the roles of $\mathcal{Q}$, upon receiving $(\textsc{ExtCall}, sid, \mathcal{F}, \mathsf{m})$ from $\mathcal{F}_{\mathsf{Ledger}}$ will call hybrid functionality $\mathcal{F}$ with message m, and return the response to $\mathcal{F}_{\mathsf{Ledger}}$. If $\mathcal{Q}$ maintains an honest majority, we obtain correctness of the verification replies returned to $\mathcal{F}_{\mathsf{Ledger}}$. We emphasize that this is a necessary modelling artifact arising from the constraints of the GUC-framework: in actual realizations we argue the parties in $\mathcal{Q}$ are the same parties which jointly realize the underlying ledger functionality as mining or staking parties.

C.2 Confidential ledger functionality

In Fig. 9 we describe the confidential token ledger functionality, $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ we require in our main construction. $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. It assume access to the $\mathcal{F}_{\mathsf{Ledger}}$ functionality in Fig. 8.

D Protocols

We detail the various protocol realizations of our scheme and their supporting smart contract programs.

D.1 Protocol realizing $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

In Fig. 10 we show how to realize our confidential token functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ from Fig. 9 on any Turing complete ledger, with the help of the smart contract $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ of Fig. 11.

Theorem 1. *Protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ GUC-realizes functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ in the $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{Setup}}$, $\mathcal{F}_{\mathsf{Sig}}$-hybrid model against any PPT-adversary corrupting any minority of committee $\mathcal{Q}$.*

Proof (Proof of Theorem 1). We construct a simulator $\mathcal{S}$ that interacts with $\mathcal{A}$, hybrid functionalities $\mathcal{F}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{NIZK}}$ and global functionalities $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Sig}}$ such that $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}} \circ \mathcal{S} \approx \Pi_{\underline{\mathsf{C}}\mathsf{Ledger}} \circ \mathcal{A}$ for any PPT environment $\mathcal{Z}$.

Concretely, to create an interaction indistinguishable from a protocol transcript in the composed setting, we construct a simulator $\mathcal{S}$ that generates valid messages for global $\mathcal{F}_{\mathsf{Ledger}}$ from simulated honest client activations and extracts inputs from dishonest messages and forwards these to ideal functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. This ensures consistency of $\mathcal{A}$'s view of $\mathcal{F}_{\mathsf{Ledger}}$ with the state of $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ during the simulated protocol execution.

On an honest GenAcct input, $\mathcal{S}$ generates a fresh signature verification key for the honest client from $\mathcal{F}_{\mathsf{Sig}}$, which it stores. For any subsequent honest input to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ which is forwarded to $\mathcal{S}$, the simulator can generate and post verifying messages global functionality $\mathcal{F}_{\mathsf{Ledger}}$.

For honest InitLedger, Transfer inputs, generating verifying messages to post on $\mathcal{F}_{\mathsf{Ledger}}$ is trivial for $\mathcal{S}$, as it generates and stores signature verification keys for honest clients. On observing dishonest InitLedger, Transfer messages on $\mathcal{F}_{\mathsf{Ledger}}$ and asserting that they are accepted by $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, the simulator

Functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

$\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ interacts with parties $\mathcal{C} = \{C_1, ..., C_p\}$. It is parameterized with token universe $\mathbb{T}$ and max. balance $\bar{\mathbf{v}}_{\mathsf{max}} \in \mathbb{Z}^{|\mathbb{T}|}$ and maintains public ledgers $\mathcal{L}$ and $\mathcal{L}_{\mathsf{Conf}}$, where $\mathcal{L}_{\mathsf{Conf}}$ maps account keys to confidential coins with hidden balances. The functionality is registered at global $\mathcal{F}_{\mathsf{Clock}}$.

Public ledger states are updated at the beginning of each clock round. On each activation, $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ reads $\mathcal{F}_{\mathsf{Clock}}$ and if ν is increased since the last activation, sets $\mathcal{L} \leftarrow \mathcal{L}'$, $\mathcal{L}_{\mathsf{Conf}} \leftarrow \mathcal{L}'_{\mathsf{Conf}}$. Initially, only *GenAcct* and *Init* interfaces are activated.

GenAcct: Upon $(\textsc{GenAcct}, sid)$ from C, forward $(\textsc{GenAcct}, sid, C)$ to $\mathcal{S}$. Upon obtaining fresh vk from $\mathcal{S}$, set $\mathcal{K}[C] \leftarrow \mathcal{K}[C] \cup \{\mathsf{vk}\}$. Return $(\textsc{AcctKey}, sid, \mathsf{vk})$.

Init: Upon receiving $(\textsc{InitLedger}, sid, \mathcal{L}_{\mathsf{Init}}, \mathsf{vk})$ from any $C_i \in \mathcal{C}$, forward to $\mathcal{S}$. Set $\mathcal{L}' \leftarrow \mathcal{L}_{\mathsf{Init}}$ and deactivate *Init* and activate all other interfaces.

Transfer: Upon $(\textsc{Transfer}, \bar{\mathbf{v}}, \mathsf{vk}_{\mathsf{rcv}}, \mathsf{vk})$ from C,
- Assert $\mathsf{vk} \in \mathcal{K}[C]$ and forward message $(\textsc{Transfer}, \bar{\mathbf{v}}, \mathsf{vk}_{\mathsf{rcv}}, \mathsf{vk})$ to $\mathcal{S}$.
- If $\mathcal{L}'[\mathsf{vk}] \geq \bar{\mathbf{v}}$, set $\mathcal{L}'[\mathsf{vk}_{\mathsf{rcv}}] \leftarrow \mathcal{L}'[\mathsf{vk}_{\mathsf{rcv}}] + \bar{\mathbf{v}}$ and $\mathcal{L}'[\mathsf{vk}] \leftarrow \mathcal{L}'[\mathsf{vk}] - \bar{\mathbf{v}}$.

Mint: Upon $(\textsc{Mint}, sid, \bar{\mathbf{v}}, \mathsf{vk})$ from C,
1. Assert $\mathcal{L}'[\mathsf{vk}] \geq \bar{\mathbf{v}}$ and forward $(\textsc{Mint}, sid, \bar{\mathbf{v}}, \mathsf{vk})$ to $\mathcal{S}$ and wait for id from $\mathcal{S}$.
2. Set $\mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}] \cup \{\langle \mathsf{id}, \bar{\mathbf{v}} \rangle\}$ & $\mathcal{L}'[\mathsf{vk}] \leftarrow \mathcal{L}'[\mathsf{vk}] - \bar{\mathbf{v}}$, return $(\textsc{Minted}, sid, \mathsf{id})$.

Confidential Transfer: Upon $(\textsc{ConfTfr}, sid, \mathsf{vk}_{\mathsf{rcv}}, \mathsf{id}_1, \mathsf{id}_2, \bar{\mathbf{v}}'_1, \bar{\mathbf{v}}'_2)$ from C,
1. Assert $\exists \mathsf{vk} \in \mathcal{K}[C] : \langle \mathsf{id}_i, \bar{\mathbf{v}}_i \rangle \in \mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}]$ for $i \in \{1, 2\}$.
2. Assert $\bar{\mathbf{v}}_1 + \bar{\mathbf{v}}_2 = \bar{\mathbf{v}}'_1 + \bar{\mathbf{v}}'_2$ and $\bar{\mathbf{v}}_i \leq \bar{\mathbf{v}}_{\mathsf{max}}$ for $i \in \{1, 2\}$.
3. Forward $(\textsc{ConfTfr}, sid, \mathsf{vk}_{\mathsf{rcv}}, \mathsf{id}_1, \mathsf{id}_2)$ to $\mathcal{S}$, and wait for $(\mathsf{id}'_1, \mathsf{id}'_2)$.
4. For $i \in \{1, 2\}$:
   - Set $\mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}] \setminus \{\langle \mathsf{id}_1, \bar{\mathbf{v}}_1 \rangle, \langle \mathsf{id}_2, \bar{\mathbf{v}}_2 \rangle\} \cup \{\langle \mathsf{id}'_2, \bar{\mathbf{v}}'_2 \rangle\}$
   - Set $\mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}_{\mathsf{rcv}}] \leftarrow \mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}_{\mathsf{rcv}}] \cup \{\langle \mathsf{id}'_1, \bar{\mathbf{v}}'_1 \rangle\}$.
5. If $\mathsf{vk}_{\mathsf{rcv}} \in \mathcal{K}[C']$ s.t. $C' \in \mathcal{I}$, send $(\langle \mathsf{id}'_1, \bar{\mathbf{v}}'_1 \rangle)$ to $\mathcal{S}$. Return $(\textsc{Change}, sid, \mathsf{id}'_2)$.

Confidential Receive: Upon $(\textsc{ConfRcv}, sid)$ from C,
- Return $(\textsc{Received}, sid, (\mathsf{vk}_1, \langle \mathsf{id}_1, \bar{\mathbf{v}}_1 \rangle), ..., (\mathsf{vk}_l, \langle \mathsf{id}_l, \bar{\mathbf{v}}_l \rangle))$, containing coins sent to C since the last call, where $\mathsf{vk}_i \in \mathcal{K}[C]$ for $i \in [l]$

Redeem: Upon $(\textsc{ConfRdm}, sid, \mathsf{id})$ from C,
1. Assert $\exists \mathsf{vk} \in \mathcal{K}[C] : \langle \mathsf{id}, \bar{\mathbf{v}} \rangle \in \mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}]$.
2. Remove $\langle \mathsf{id}, \bar{\mathbf{v}} \rangle$ from $\mathcal{L}'_{\mathsf{Conf}}[\mathsf{vk}]$ and set $\mathcal{L}'[\mathsf{vk}] \leftarrow \mathcal{L}'[\mathsf{vk}] + \bar{\mathbf{v}}$.
3. Send $(\textsc{ConfRdm}, sid, \langle \mathsf{id}, \bar{\mathbf{v}} \rangle, \mathsf{vk})$ to $\mathcal{S}$.

GetLedger: Upon $(\textsc{GetLedger}, sid)$, compute sanitized $\mathcal{L}''_{\mathsf{Conf}}$ such that coin balances are removed from $\mathcal{L}_{\mathsf{Conf}}$. Return $\mathcal{L}$, $\mathcal{L}''_{\mathsf{Conf}}$.

Fig. 9: Functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ for Confidential Ledgers.

can extract all dishonest inputs to forward to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, as these messages are posted to $\mathcal{F}_{\mathsf{Ledger}}$ in cleartext.

On an honest Mint input, $\mathcal{S}$ must generate and send a verifying Call message to $\mathcal{F}_{\mathsf{Ledger}}$ with the minted amount $\bar{\mathbf{v}}$ which activates the deployed $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ contract instance to mint a fresh confidential token. Since $\mathcal{S}$ simulates protocol messages from honest clients, it can generate a valid commitment for $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

itself and store its opening $(\bar{\mathbf{v}}, r)$. With the commitment opening, it obtains a verifying NIZK via $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$ proving $\mathcal{R}(\bar{\mathbf{v}}, c; r) = \{c = \mathbf{g}^{\bar{\mathbf{v}}} h^r\}$. For a dishonest mint message observed on $\mathcal{F}_{\mathsf{Ledger}}$ by $\mathcal{S}$, the simulator trivially extracts inputs for $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$: both minted amount and minting account key in the Call message sent to activate minting in the $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ contract instance are observable in cleartext on $\mathcal{F}_{\mathsf{Ledger}}$.

On an honest ConfTransfer input, $\mathcal{S}$ generates valid coin commitments and rangeproofs for a call activating ConfTrfr on the $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ contract instance deployed to $\mathcal{F}_{\mathsf{Ledger}}$. For an *honest sender and honest recipient*, $\mathcal{S}$ needs to generate output coin commitments that are consistent with the chosen input coins for the simulated protocol. Here, $\mathcal{S}$ always possesses the openings of the input coin commitments:

- *Coins previously received from an honest sender* were generated by $\mathcal{S}$ with arbitrary openings previously generated by $\mathcal{S}$: since $\mathcal{S}$ does not learn the transfer amount for confidential transfer between honest users, it generates output coins commitments with *arbitrary balances*, such that the product equality of input and output commitments holds: $\mathbf{g}^{\bar{\mathbf{v}}_1} h^{r_1} \mathbf{g}^{\bar{\mathbf{v}}_2} h^{r_2} = \mathbf{g}^{\bar{\mathbf{v}}'_1} h^{r'_1} \mathbf{g}^{\bar{\mathbf{v}}'_2} h^{r'_2}$. However, since simulated setup functionality $\mathcal{F}_{\mathsf{Setup}}$ samples $s \leftarrow\$ \mathbb{F}_p$ and outputs $h = g^s$, coins generated by $\mathcal{S}$ can later be *equivocated* to any value.
- *Coins previously received from a dishonest sender* feature openings sent directly to the receiving honest client simulated by $\mathcal{S}$ in the simulated protocol.

Thus, for an honest confidential transfer sending coins to another honest party, $\mathcal{S}$ generates output coin commitments with arbitrary chosen coin balances, stores their openings and obtains verifying rangeproofs via $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$. For an *honest sender and dishonest recipient*, $\mathcal{S}$ learns the transferred amount from $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, and can generate output coin commitments with correct balances and post these to $\mathcal{F}_{\mathsf{Ledger}}$ (with equivocation of the input coin commitments if necessary). Then, it forwards the coin openings as a simulated protocol message to the dishonest recipient.

Finally for a *dishonest sender and dishonest recipient* $\mathcal{S}$ can extract openings for all coins generated by the dishonest sender since they all have associated NIZK's obtained by sending valid coin openings to the simulated $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$ instance. Thus, the simulator can forward the transferred amounts to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. For a *dishonest sender and honest recipient*, the simulated honest recipient obtains transferred coin commitment opening as a protocol message, allowing $\mathcal{S}$ to forward this input to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$.

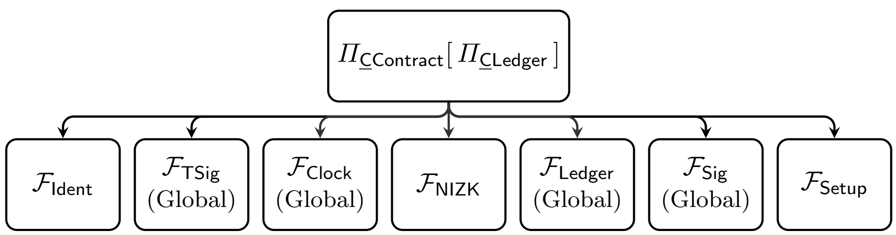
On an honest ConfReceive, the simulator must have previously provided inputs to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ for confidential transfers *initiated* by dishonest parties, as previously described. On a *dishonest confidential receive*, $\mathcal{S}$ will have previously sent the openings of the honestly sent coins to the dishonest recipient as a protocol message, in addition to having generated valid coins and rangeproofs observable on $\mathcal{F}_{\mathsf{Ledger}}$.

On an honest ConfRedeem, if the redeemed coin was originally sent by a dishonest user, $\mathcal{S}$ must have also received its opening as a protocol message , as it simulates the role of the honest user in the protocol execution. Otherwise, the redeemed coin must have been sent by an honest user, and can thus be equivocated by $\mathcal{S}$. Thus, with the equivocated coin opening, $\mathcal{S}$ can produce a verifying NIZK for the honest redeem action in the simulated protocol view. On a *dishonest redeem*, $\mathcal{S}$ observes the redeemed value publicly on $\mathcal{F}_{\mathsf{Ledger}}$, and can thus forward this input to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$.

As long as the majority of parties in $\mathcal{Q}$ are honest, verification responses from $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ are interpreted correctly by the call-back gate on $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. Thus, the public state of $\mathcal{L}_{\mathsf{Conf}}$ on $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ observed in the simulated protocol view is consistent with the confidential ledger maintained by $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$.

Finally, we note that the *updates* to ledger states induced by client activations are applied at the beginning of each $\mathcal{F}_{\mathsf{Clock}}$ round in both global $\mathcal{F}_{\mathsf{Ledger}}$ and ideal functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. □

D.2 Protocol realizing $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$



In Fig. 12 and 13 we show how to realize our privacy preserving smart contract functionality $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ from Fig. 2 on any Turing complete ledger, with the help of a smart contract with code of $\mathcal{X}_{\mathsf{Lock}}$ of Fig. 14 to manage confidential tokens and $\mathcal{X}_{\mathsf{Collateral}}$ of Fig. 15 to manage underlying collateral. Note that $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$ extends $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$, and similarly that contract $\mathcal{X}_{\mathsf{Lock}}$ extends $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$.

Theorem 2. *$\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}[\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}]$ realizes $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}[\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}]$ in the $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Ident}}$, $\mathcal{F}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{Setup}}$, $\mathcal{F}_{\mathsf{Sig}}$, $\mathcal{F}_{\mathsf{TSig}}$-hybrid model against any PPT-adversary corrupting at most $n-1$ of the n servers $\mathcal{P}$ statically and any minority of $\mathcal{Q}$.*

Proof. (Theorem 2) We construct a simulator $\mathcal{S}$ that interacts with $\mathcal{A}$, hybrid functionalities $\mathcal{F}_{\mathsf{Ident}}$, $\mathcal{F}_{\mathsf{NIZK}}$, and global functionalities $\mathcal{F}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{Sig}}$, $\mathcal{F}_{\mathsf{TSig}}$ such that $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}} \circ \mathcal{S} \approx \Pi_{\underline{\mathsf{C}}\mathsf{Ledger}} \circ \mathcal{A}$ for any PPT environment $\mathcal{Z}$.

Upon an honest Init, the simulator $\mathcal{S}$ simulates the roles of the honest parties in the simulated protocol execution, and jointly generates a threshold signature verification key with the dishonest parties via $\mathcal{F}_{\mathsf{TSig}}$. It simulates **GenAcct** and **InitLedger** as in $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$. As $\mathcal{S}$ generates the signature verification key for each honest server, it can call $\mathcal{F}_{\mathsf{Sig}}$ and generate verifying messages for the simulated honest server to post on global $\mathcal{F}_{\mathsf{Ledger}}$, observable by $\mathcal{A}$. Since $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$ extends $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$, $\mathcal{S}$ signs messages that initialize contracts $\mathcal{X}_{\mathsf{Lock}}[\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}]$ and $\mathcal{X}_{\mathsf{Collateral}}$, and can authorize collateral deposits to the contract instance $\mathcal{X}_{\mathsf{Collateral}}$ on $\mathcal{F}_{\mathsf{Ledger}}$.

Upon an honest Enroll, the simulator is forwarded the input client coin identifier and account verification key. $\mathcal{S}$ simulates the honest party by generating output masks for which it samples the random openings. Then it determines valid openings for the honest input coin:

- If the honest input coin was previously transferred by a dishonest party, simulator $\mathcal{S}$ can extract its opening from the NIZK range-proof generated via simulated hybrid $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$.
- If the honest input coin was previously transferred by an honest party, $\mathcal{S}$ must have generated the openings itself (See simulator of $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$).

In either case, the simulator sends valid openings of both honest input coins and mask commitments to hybrid $\mathcal{F}_{\mathsf{Ident}}$.

Subsequently, the simulator can simulate a consistent protocol execution of **Verify input** which only aborts if $\mathcal{A}$ provides inputs to simulated hybrid $\mathcal{F}_{\mathsf{Ident}}$ that are inconsistent with the input coin and mask commitments sent to $\mathcal{X}_{\mathsf{Lock}}$ on simulated $\mathcal{F}_{\mathsf{LedgerVM}}$. It simulates the batched sigma protocol to check input consistency in the simulated execution of **Verify input** in $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$. Inconsistency of inputs must arise from cheating by $\mathcal{A}$ and results in an abort. As shown in Section 3.1, the probability that the simulated protocol aborts due to inconsistent inputs whilst the ideal functionality continues is negligible in the group order of the Pedersen commitment scheme.

Upon an honest EVALUATE and its successful completion, the simulator will jointly sign eval via $\mathcal{F}_{\mathsf{TSig}}$ with the dishonest parties.

Upon an honest OPEN, the simulator first observes what $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ outputs, and then will modify the state of the simulated $\mathcal{F}_{\mathsf{Ident}}$ instance, such that the adversary in the simulated protocol observes the masked outputs consistent with what $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ outputs.

Upon an honest WITHDRAW, the simulator does nothing. At each $\mathcal{F}_{\mathsf{Clock}}$ round during the simulated protocol execution, if the adversary aborts, $\mathcal{S}$ will forward an abort to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$. If a dishonest party cheats during the **Open** phase of the simulated protocol execution it will be identified by the simulated $\mathcal{F}_{\mathsf{Ident}}$ instance, and its identity is forwarded to $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ by $\mathcal{S}$.

$\mathcal{S}$ simulates the honest parties of committee $\mathcal{Q}$ in the simulated protocol execution. As long as the majority of parties in $\mathcal{Q}$ are honest, verification responses from $\mathcal{F}_{\mathsf{Ident}}$ are interpreted correctly by the call-back gate on $\mathcal{X}_{\mathsf{Lock}}$, permitting the cheating parties in the simulated protocol execution to be correctly identified during an abort. □

E Confidential contract extensions

Multi-round confidential contracts. For an ongoing, multi-round confidential contract execution, we argue a simple extension to our $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$ model to permit clients can selectively participate in the **Enroll** phase of a round, or to *skip* a given round by ticking the $\mathcal{F}_{\mathsf{Clock}}$ after calling a **Skip Round** interface.

In the protocol realizing a multi-round confidential contract functionality, we propose an output budget for each client corresponding to the number of *unused*, pre-processed output masks: in each round, a client will receive a masked output which can be retrieved from the contract $\mathcal{X}_{\mathsf{Lock}}$ on $\mathcal{F}_{\mathsf{Ledger}}$, regardless whether it provides a new input and participates in **Enroll**: masked outputs for a specific client are generated in each round until its pre-processed output masks have been consumed. The evaluation of the confidential contract in each round is still evaluated over *all clients* and their secret state $\mathsf{st} = ([s_1], ..., [s_m])$, even if only a subset have provided fresh inputs for a round. A client **Skip** implies evaluating the contract circuit over default input values.

Each participation in an **Enroll** phase of a round permits a client to *restore* its depleted output budget, by generating masks in commitment form and inputting their openings to the MPC instance, which are subsequently verified for consistency in the **Verify Input** protocol phase. Each output mask can be associated with a *fee* paid to the servers executing the MPC: once all output budgets (and associated output masks) are consumed, the multi-round confidential contract can terminate safely.

Mitigation of token minting. If all servers collude, they can jointly sign (via $\mathcal{F}_{\mathsf{TSig}}$) an arbitrary distribution of funds in the **Open** phase of $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$, resulting in output coins generated on $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$, with balances that exceed the value of underlying tokens managed by $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ and even the underlying ledger. This is not publicly detectable, even if $\mathcal{X}_{\mathsf{Lock}}$ implemented a product consistency check over input and output coins with correlated commitment randomness, $\prod_{j\in[m]} c_j = \prod_{j\in[m]} c_j^{\mathsf{out}}$, as hidden balances can exceed $\bar{\mathbf{v}}_{\mathsf{max}} = 2^l - 1$, and p additional units of each token type can be minted. Although the *incorrect redistribution* of confidential balances input to a contract round cannot be mitigated, we show an extension of $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$ that prevents the minting of tokens under full server corruption, at the cost of a constant-factor increase in communication complexity. Here, we adopt the bit decomposition approach as in Banerjee *et al.* [4], but greatly improve on the protocol efficiency by doing without any NIZK's and commitments generated inside the MPC circuit, which is strictly reserved for contract application logic.

In the **Enroll** phase, clients each generate bit a commitment pair $(c_0 = \mathsf{com}(b_0, s_0), c_1 = \mathsf{com}(b_1, s_1))$ for each bit position $k \in [l]$, such that $b_i = 0 \wedge b_{i-1} = 1$ for random $i \leftarrow\!\!\$\, \{0, 1\}$. Let π denote the bit permutation sampled by the client for the bit position $k \in [l]$, such that:

$$\pi(k) = \begin{cases} 0 & b_{k,0} = 0 \wedge b_{k,1} = 1 \\ 1 & b_{k,0} = 1 \wedge b_{k,1} = 0 \end{cases}$$

This permutation on the individual bits is later used to mask the bit-decomposed output. Commitment pairs are posted to $\mathcal{F}_{\mathsf{Ledger}}$, together with an efficient sigma proof that commitments are to bit values [35], incurring an additional communication complexity logarithmic in the size of the commitment group order.

During the **Enroll** phase, users input the opening to these bit commitment pair in the permuted order:

$$([\,\bar{v}\,], [\,\bar{r}\,], \{([\,b_{0,k}\,], [\,s_{0,k}\,], [\,b_{1,k}\,], [\,s_{1,k}\,])\}_{k\in[l]})$$

where $c_{\mathsf{in}} = (\bar{v}, \bar{r})$ is the opening to the confidential input coin, and each tuple $(b_{0,k}, s_{0,k}, b_{1,k}, s_{1,k})$ is the opening to the k'th bit commitment pair with permuted bit ordering. We adopt a well-formedness check on bits input to $\mathcal{F}_{\mathsf{Ident}}$ from [33]: servers assert for each $k \in [l]$ bit position, that one of $[\, b_{k,0} \,], [\, b_{k,1} \,]$ holds the value 1 and the other holds the value 0. Concretely, for bit pair $([\, b_0 \,], [\, b_1 \,])$, servers jointly sample and open $\alpha, \beta, \gamma, \leftarrow\$ \mathbb{F}$, and compute:

$$[\, t \,] = \alpha \cdot ([\, b_0 \,] \cdot [\, b_0 \,] - [\, b_0 \,]) + \beta \cdot ([\, b_1 \,] \cdot [\, b_1 \,] - [\, b_1 \,]) + \gamma \cdot ([\, b_0 \,] \cdot [\, b_1 \,])$$

$$[\, t' \,] = ([\, b_0 \,] + [\, b_1 \,])$$

Upon securely opening t and t', servers assert that $t = 0 \wedge t' = 1$. Consistency between all commitments and their openings input to $\mathcal{F}_{\mathsf{Ident}}$ are verified during **Verify input** phase by the servers.

Importantly, the financial output of a client is output in bit-decomposed form, where individual bits are permuted in the ordering as chosen by the clients.

$$(b'_1, ..., b'_l)$$

Let $(b_1, ..., b_l)$ denote the true bit-decomposition of a clients output balance. Then $b'_k = b_k$ if $\pi(k) = 0$ and is bit permuted otherwise, where π denotes the permutation chosen by the client in the enroll phase.

For contract $\mathcal{X}_{\mathsf{Lock}}$ to generate the client output coin from bit commitments submitted during **Enroll**, it computes.

$$c_{\mathsf{out}} = \prod_{k \in [l]} c_{k,i}^{2^k} \text{ where } i = b'_k \in \{0, 1\}$$

Here, note that b'_k is interpreted as selector for bit commitment pair $(c_{k,0}, c_{k,1})$ for bit position k. As both $b'_k = i$ and the bit message of $c_{k,i}$ are permuted by $\pi(k)$, the hidden balance of c_{out} is unmasked. Given the generation of output coins from l bit balance representations, confidential output balances are bounded by $2^l - 1 = \bar{v}_{\mathsf{max}}$.

It remains to prove consistency between input and output commitments to $\mathcal{X}_{\mathsf{Lock}}$ to ensure no token minting occurred. For this, servers compute the commitment randomness for each client output coin and the difference in commitment randomness between the input and output coins.

$$[\, s_{\mathsf{out}} \,] = \sum_{k \in [l]} 2^k \cdot [\, s_{k,i} \,] \text{ where } i = b'_k \in \{0, 1\} \qquad [\, \bar{r}_{\mathsf{diff}} \,] = \sum_{j \in [m]} [\, \bar{r}_{\mathsf{in},j} \,] - [\, s_{\mathsf{out},j} \,]$$

Servers locally compute $h^{\bar{r}^{(i)}_{\mathsf{diff}}}$ over the local share value of $[\, \bar{r}_{\mathsf{diff}} \,]$ and send it to all other servers. Each server then reconstructs $h^{\bar{r}_{\mathsf{diff}}}$ and verifies that sum of confidential input and output balances must be equal and that no tokens are minted (balance over-flow is mitigated by bounding output balances by $2^l - 1$).

$$\prod_{j \in [m]} c_{\mathsf{in},j} = h^{\bar{r}_{\mathsf{diff}}} \cdot \prod_{j \in [m]} c_{\mathsf{out},j}$$

Table 4: Complexity of the per user overhead by using the stand-alone token minting mitigation approach.

	Exponentiation	MPC mult.
User	$6 \cdot l$	0
Server	$8 \cdot l + 1$	$42.5 \cdot l + 15$
Comm. #$\mathbb{G}$elem.	$O(n \cdot l)$	$O(n^2 \cdot l)$

We outline the overhead of this approach to prevent malicious minting when all servers are corrupted in Tab. 4, when assuming that Schnorr proofs are added for each commitment to allow extraction (though not in UC) and when using the work of Reistad and Toft [55] to do the needed bit decomposition in MPC.

F Applications

In this section we briefly outline some interesting application which privacy preserving smart contracts can help facilitate, along with our scheme can be used to provide privacy preserving side-chains and how it can be extended to allow for privacy preserving cross-chain smart contracts.

F.1 Privacy preserving applications

Several general applications for privacy preserving smart contracts have already been suggested in previous works. We briefly outline some of these here.

Auctions Auctions of digital goods, or digital deeds linked to physical goods, can be constructed simpler and more efficiently than with non-privacy preserving smart contracts. Our solution could be used to implement first and second price auctions securely and privately. Concretely confidential tokens reflecting the maximum bid each user should be transferred to a privacy preserving smart contract along with the good for sale. The smart contract then compares the bids and transfers ownership of the good and handles the payment and refunding, according to code of the smart contract being executed in MPC.

Identity management Decentralized Identity (DID) management is the idea that, by using blockchains, users remain in charge over how their private attributes (certified by an appropriate authority) are used online. Multiple schemes for this has been suggested such as Sovrin [42] or CanDID [49]. However, these schemes generally only consider leveraging the blockchain for storing user's attribute information. However, using privacy preserving smart contracts would allow integration of user-certified attributes in both the web 2 and web 3 space. Concretely the users could give their hidden certified attributes as input the privacy preserving smart contract, which can validate them privately and use the content of these attributes to affect its business logic. For example the attributes can be used to decide the price of an NFT or to validate whether a user is privilege enough to execute certain commands of the contract.

Mixer Our structure can naturally be extended to allow for a mixing functionality. While several other technologies exist for this, we observe that doing this in MPC allows several advantages that can prevent the mixing to be used for money laundering. Concretely we could imagine that KYC (Know Your Customer) information linked to the users' blockchain address must be given and privately validated against deny-lists, to prevent criminals using this service. Even if deny-lists are not in use, linking to an actual identity could also be leveraged to allow a given user to only get privacy on the first x amount of tokens they mix, and after that, information on the token amount will become public.

F.2 Anonymous side-chain

Our solution could also be used to construct privacy preserving side-chains. When no server is trying to cheat, there is technically no need for the MPC servers to post anything related to the specific clients and their input to the blockchain, after the *evaluation* phase. Thus the MPC servers can alone realize a privacy preserving side-chain where they in MPC hold the opening information to the commitments of hidden tokens. Thus users can request transfers to other users in this side-chain, if the servers just use the MPC scheme to keep track of how many tokens each user has. At certain intervals, each user can then just decide to get paid back whatever they hold in the side-chain, by the execution of the *open* and *withdraw* phases. This can be used to enhance the anonymity of hidden transfers, since now *only* the MPC servers know the transaction graph, and yet they do not know the transaction amounts. An interesting observation with this case is also that the side-chain will be faster and cheaper to use than the underlying layer 1 blockchain, since it will only be managed by the MPC servers.

F.3 Cross-chain Exchange

In section we will discuss how to use the ideas of P2DEX [9] to make our scheme capable of doing confidential computation and transactions *across* multiple layer 1 blockchains.

Decentralized exchanges. When it comes to decentralized exchange, multiple approaches exist but generally fall into one of the following families:

P2P Two parties, each with tokens on a chain the other decide, agree on doing an exchange with a certain exchange rate. This is for example the approach used in hash-proofs [57]. This unfortunately requires multiple rounds of on-chain interaction, fees, not to mention the issue of having parties find each other.

Exchange chain A chain contains wrapped tokens pegged to their native counterparts through holding smart contracts on all the native chains. This allows to reduce the cross-chain exchange problem to an on-chain problem, assuming the problem of inter-chain communication has been solved. With an exchange chain in place there are multiple ways of facilitating exchanges, since now the problem is reduced same-chain exchange: **Order book**: In the order book approach all orders (e.g. limit orders) are written to the chain and then matched and carried out by a smart contract. Unfortunately this inherently front-running by miners. **AMM**: An AMM is basically a liquidity holding smart contract, allowing exchange between two different tokens. The smart contract then facilitates exchange between tokens of type A and B, with an exchange rate that ensure that the product of the amount of tokens in the contract, remains constant. Unfortunately AMMs are highly susceptible to front-running by miners, since orders and exchange rates will be known to miners before they get carried out.

While these approaches solve some issues related to decentralized exchange none of these are unfortunately a silver bullet for users who desire both ease of use, decentralization and front-running resistance [6].

P2DEX. P2DEX [9] is a different system for achieving cross chain exchange, although it can be considered a special case of the order book approach. It uses a set of outsourced MPC servers [28,37] who threshold control burner addresses, where the clients transfer the tokens they wish to exchange, to compute order matching based on private input of clients. The servers then use the threshold keys for these addresses to send money out of these burner addresses to the intended recipients.

Adding cross-chain functionality. Like our work, P2DEX also use a set of MPC servers to compute on client's private input. But unlike P2DEX we don't use burner addresses, but instead a holding smart contract Lock, administered by a *single* distributed signing key. But we note that the P2DEX approach will also work with the smart contract based approach. Thus by simply having Lock smart contracts instantiated on multiple blockchains, with different administration keys, these can form the same purpose as the burner addresses in P2DEX. Concretely this can be realized by simply having each client provide a confidential token commitment on each chain they expect to receive some tokens. Note that such a commitment can be of 0 tokens. The MPC servers will then validate all the confidential tokens given to Lock on each of the different chains, through the *verify input* phase. Then one, unified privacy preserving smart contract ConfContract will be executed, which will yield new commitments for each of the relevant clients on each chain. The clients can then use *withdraw* in Lock on each of the different chains to finish the computation and get their confidential tokens on the relevant chains.

This approach could of course also be combined with the mixer idea above, allowing for cross-chain mixers with selective levels of privacy depending on the amounts mixed by a given user.

Doing cross-chain exchange on hidden tokens also has the advantage of allowing parties, with very large amounts of tokens, to carry out an exchange in a slow and continues manner, thus preventing sudden exchange fluctuations. In fact, our system could enforce an upper bound on the amount of tokens to exchange in one round, and automatically split up large orders so they get completed over multiple rounds, instead of just one.

Security. The overall security of this approach follows from P2DEX, although we will also argue that intuitively there is nothing non-trivial to simulation if we adjust our ideal functionalities and protocol to follow this approach. Basically where there is formal cryptography to be proven is in the integration between the different ideal functionalities, in particular when ensuring consistency between the input to outsourced MPC and the commitments transferred to the holding contract. However, using our scheme across multiple chains make no difference in this. The only modelling difference is simply that the ideal blockchain functionalities can no be considered to "wrap" different instances of the same functionality (thus reflecting multiple chains). Such a wrap inherently does not affect secure insofar that it does not contain any logical loopholes.

F.4 Future work

While we construct and prove UC-secure a scheme for decentralized privacy preserving smart contracts, we believe there are multiple paths for future work to explore. An immediate interesting path is to implement and benchmark the system for some of the applications we have discussed. For example, a better formalization of the cross-chain approach, along with an investigation of MPC friendly algorithms for fair matching of exchange orders could allow the realization of a highly secure and private decentralized exchange. For such applications it also becomes important to investigate, the logic of how to use the collateral to punish malicious parties in case they cheat. In particular such that no rational party will end up with a skewed or perverse incentives. In relation to this, it would be interesting to investigate how to integrate Pedersen commitments with MPC in an efficient way, *without* requiring the MPC computation domain to be the same as the Pedersen message space. This could have a great affect on the efficiency if the MPC computation domain is significantly smaller than 256 bit. Currently we require the sharing the opening information of commitments to happen in a P2P manner, off-chain, when transferring hidden tokens. It would be interesting to investigate how to implement hidden tokens in a way that does not require client-to-client communication when doing transfers, while working with the rest our protocol. In relation to this, other ways the overall usability of our system could also be improved is constructing a protocol leveraging other existing results to allow stateless clients. For example through some notion of password authenticated distributed secret sharing [18]. In continuation of this, investigating how to prevent the use of user-supplied masks for each round of execution private smart contract computation, would also give a great impact on the usability of our solution.

Protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$

$\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ is run by clients $\mathcal{C}$ and committee $\mathcal{Q}$. The protocol runs in the presence of $\mathcal{F}_{\mathsf{Ledger}}, \mathcal{F}^{R}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{Sig}}$ instances. Initially, only accept inputs GenAcct and InitLedger.

GenAcct: Upon $(\textsc{GenAcct}, sid)$, obtain fresh vk from $\mathcal{F}_{\mathsf{Sig}}$. Set key store to $\mathcal{K} \leftarrow \mathcal{K} \cup \{\mathsf{vk}\}$, and return $(\textsc{NewAcct}, sid, \mathsf{vk})$.

InitLedger: Upon $(\textsc{InitLedger}, sid, \mathcal{L}_{\mathsf{Init}}, \mathsf{vk})$, client C parses $\mathcal{L}_{\mathsf{Init}}$ as a map from a set of signature keys to token balances $\mathbb{G} \mapsto (\mathbb{T} \mapsto \mathbb{Z})$ and asserts $\mathsf{vk} \in \mathcal{K}$.

1. C initializes $\mathcal{F}_{\mathsf{Ledger}}$ with $\mathcal{Q}$ and signs $m = (\textsc{Init}, sid, \mathcal{L}_{\mathsf{Init}})$ via $\mathcal{F}_{\mathsf{Sig}}$ with key vk. Send $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma_{\mathsf{vk}}(m))$ to $\mathcal{F}_{\mathsf{Ledger}}$.
2. C compiles $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ to initial contract state and circuit (γ, T). Sign $m = (\textsc{Deploy}, sid, \gamma, T, \mathsf{vk})$ via $\mathcal{F}_{\mathsf{Sig}}$ with vk, and send $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma_{\mathsf{vk}}(m))$ to $\mathcal{F}_{\mathsf{Ledger}}$. Ignore further InitLedger inputs and accept all other inputs.

Transfer: Upon $(\textsc{Transfer}, \bar{\mathbf{v}}, \mathsf{vk}_{\mathsf{rcv}}, \mathsf{vk})$, obtain $\mathcal{L}$ from GetLedger procedure. Assert $\mathsf{vk} \in \mathcal{K}$ and $\mathcal{L}[\mathsf{vk}] \geq \bar{\mathbf{v}}$. Sign $m = (\textsc{Transfer}, sid, \bar{\mathbf{v}}, \mathsf{vk}_{\mathsf{rcv}})$ via $\mathcal{F}_{\mathsf{Sig}}$ with key vk and send $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma_{\mathsf{vk}}(m))$ to $\mathcal{F}_{\mathsf{Ledger}}$.

Mint: On $(\textsc{Mint}, sid, \bar{\mathbf{v}}, \mathsf{vk})$, client C,

1. Assert $\mathsf{vk} \in \mathcal{K}$, obtain state $\mathcal{L}, \Gamma$ from $\mathcal{F}_{\mathsf{Ledger}}$ and assert $\mathcal{L}[\mathsf{vk}] \geq \mathbf{v}$.
2. Sample $r \leftarrow\!\$\, \mathbb{F}$, compute $c \leftarrow \mathsf{com}(\bar{\mathbf{v}}, r)$ and obtain string π from $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ where $\mathcal{R}(c, \bar{\mathbf{v}}; r) = \{c = \mathsf{com}(\bar{\mathbf{v}}, r)\}$.
3. Sign $(\textsc{Call}, sid, (\mathsf{cn}, f(\textsc{Mint}), (c, \pi), \bar{\mathbf{v}}))$ via $\mathcal{F}_{\mathsf{Sig}}$ with vk and post to $\mathcal{F}_{\mathsf{Ledger}}$.
4. Set wallet $\mathcal{W}[\mathsf{vk}] \leftarrow \mathcal{W}[\mathsf{vk}] \cup \{\langle \mathsf{id} = c, (\bar{\mathbf{v}}, r)\rangle\}$ and return $(\textsc{Minted}, sid, \mathsf{id})$.

ConfTransfer: On $(\textsc{ConfTrfr}, sid, \mathcal{C}_{\mathsf{rcv}}, \mathsf{vk}_{\mathsf{rcv}}, \{\mathsf{id}_i, \bar{\mathbf{v}}'_i\}_{i\in\{1,2\}})$, client C:

1. Assert $\exists \mathsf{vk}_{\mathsf{src}} \in \mathsf{dom}(\mathcal{W}) : (\mathsf{id}_i, (\bar{\mathbf{v}}_i, r_i)) \in \mathcal{W}[\mathsf{vk}_{\mathsf{src}}]$, $\bar{\mathbf{v}}_1 + \bar{\mathbf{v}}_2 = \bar{\mathbf{v}}'_1 + \bar{\mathbf{v}}'_2$, $\bar{\mathbf{v}}'_i \leq \bar{\mathbf{v}}_{\mathsf{max}}$.
2. For $i \in \{1, 2\}$, sample $r'_i \leftarrow\!\$\, \mathbb{F}$ such that $\sum_{i\in\{1,2\}} r'_i = \sum_{i\in\{1,2\}} r_i$ and compute $c'_i = \mathsf{com}(\bar{\mathbf{v}}'_i, r'_i)$ and π_i via $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ that proves $\mathcal{R}(c'_i; \bar{\mathbf{v}}'_i, r'_i) = \{\bar{\mathbf{v}}'_i \leq \bar{\mathbf{v}}_{\mathsf{max}}\}$
3. Sign and post $(\textsc{Call}, sid, (\mathsf{cn}, f(\textsc{ConfTransfer}), x, 0^t), \mathsf{vk}_{\mathsf{src}})$ to $\mathcal{F}_{\mathsf{Ledger}}$, where $x = (\{c_i, c'_i, \pi_i\}_{i\in\{1,2\}}, \mathsf{vk}_{\mathsf{rcv}})$.
4. Send $(\textsc{ConfTrfr}, sid, \bar{\mathbf{v}}'_1, r'_1, \mathsf{vk}_{\mathsf{src}})$ to $\mathcal{C}_{\mathsf{rcv}}$, which stores it.
5. Set $\mathcal{W}[\mathsf{vk}_{\mathsf{src}}] \leftarrow \mathcal{W}[\mathsf{vk}_{\mathsf{src}}] \cup (\mathsf{id}'_2 = c'_2, (\bar{\mathbf{v}}'_2, r'_2))$ and return $(\textsc{Change}, sid, \mathsf{id}'_2)$.

ConfReceive: On $(\textsc{ConfReceive}, sid)$ client C:

1. For $\mathsf{vk} \in \mathsf{dom}(\mathcal{W})$, retrieve $\{(\bar{\mathbf{v}}_i, r_i, \mathsf{vk}_i)\}_{i\in[l]}$ received from clients since the last ConfReceive input and $\mathcal{L}_{\mathsf{Conf}}$ from $\mathcal{F}_{\mathsf{Ledger}}$.
2. For $(\bar{\mathbf{v}}, r, \mathsf{vk}) \in \{(\bar{\mathbf{v}}, r, \mathsf{vk}_i)\}_{i\in[l]}$, compute $c = \mathsf{com}(\bar{\mathbf{v}}, r)$ and assert $c \in \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}]$.
   - If satisfied, add $(\mathsf{id} = c, (\bar{\mathbf{v}}, r))$ to $\mathcal{W}[\mathsf{vk}]$.
3. Returns $(\textsc{Received}, (\mathsf{vk}_1, \langle \mathsf{id}_1, \bar{\mathbf{v}}_1\rangle), ..., (\mathsf{vk}_l, \langle \mathsf{id}'_l, \bar{\mathbf{v}}'_l\rangle))$ for l' received transfers.

Redeem: On $(\textsc{ConfRdm}, sid, \mathsf{id})$ client C,

1. If $\exists(\mathsf{vk}, \bar{\mathbf{v}}, r) : (\mathsf{id}, (\bar{\mathbf{v}}, r)) \in \mathcal{W}[\mathsf{vk}]$, where $\mathsf{id} = \mathsf{com}(\bar{\mathbf{v}}, r)$.
2. Compute π via $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ which proves $\mathcal{R}(c, \bar{\mathbf{v}}; r) = \{c = \mathsf{com}(\bar{\mathbf{v}}, r)\}$.
3. Sign and post $(\textsc{Call}, sid, (\mathsf{cn}, f(\textsc{Redeem}), (\bar{\mathbf{v}}, c, \pi), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$.

GetLedger: Upon $(\textsc{GetLedger}, sid)$, client C obtains $(\mathcal{L}, \Gamma)$ and contract id cn from $\mathcal{F}_{\mathsf{Ledger}}$, reads $(\gamma, \mathbf{w}, T) \leftarrow \Gamma[\mathsf{cn}]$, and parses γ as $(\mathcal{L}_{\mathsf{Conf}}, \bar{\mathbf{m}})$. C outputs $(\textsc{Ledger}, sid, \mathcal{L}, \mathcal{L}_{\mathsf{Conf}})$.

ExtCall: Upon $(\textsc{ExtCall}, sid, \mathcal{F}, \mathsf{m})$ received from $\mathcal{F}_{\mathsf{Ledger}}$, party $Q \in \mathcal{Q}$ forwards m to hybrid instance $\mathcal{F}$ and waits. Upon response ret from $\mathcal{F}$, party Q forwards $(\textsc{CallBack}, sid, \mathsf{ret})$ to $\mathcal{F}_{\mathsf{Ledger}}$.

Fig. 10: Protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$ UC-securely realizing $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

Program $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

On input $(\nu \,|\, \gamma \,|\, \bar{\mathbf{w}} \,|\, \mathsf{cn}, \mathsf{fn}, x, \bar{\mathbf{v}} \,|\, \mathsf{vk})$, parses function selector fn and execute function routine with input string $x \in \{0,1\}^*$, parsed according to function descriptions below. Further, parse contract state γ as $\mathcal{L}_{\mathsf{Conf}}$, where $\mathcal{L}_{\mathsf{Conf}} : \mathbb{G}^{\mathsf{vrk}} \rightarrow \{\mathbb{G}^{\mathsf{com}}, ...\}$. $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ is parameterized with committee $\mathcal{Q}$.

Mint: parse x as (c, π) where $c \in \mathbb{G}$ and $\pi \in \{0,1\}^*$.
1. Send call to $\mathcal{Q}$ with msg for $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ to verify that π proves $\mathcal{R}(\bar{\mathbf{v}}, c; r) = \{c = \mathbf{g}^{\bar{\mathbf{v}}} h^r\}$.
2. Set $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \cup \{c\}$, $\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} + \bar{\mathbf{v}}$ and $\mathcal{L}[\mathsf{vk}] \leftarrow \mathcal{L}[\mathsf{vk}] - \bar{\mathbf{v}}$.
3. Output updated $(\mathcal{L}, \gamma' = \mathcal{L}_{\mathsf{Conf}}, \bar{\mathbf{w}})$.

ConfTransfer: parse x as $(c_1, c_2, c'_1, c'_2, \pi_1, \pi_2, \mathsf{vk}_{\mathsf{rcv}})$ where $c_i, c'_i \in \mathbb{G}$, $\pi_i \in \{0,1\}^*$ for $i \in \{1,2\}$ and $\mathsf{vk}_{\mathsf{rcv}} \in \mathbb{G}$.
1. Assert $\{c_1, c_2\} \in \mathcal{L}^{\mathsf{Conf}}[\mathsf{vk}]$ and that $c_1 \cdot c_2 = c'_1 \cdot c'_2$ holds.
2. Send call to $\mathcal{Q}$ with msg for $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ to verify $\forall i \in \{1,2\}$: π_i proves $\mathcal{R}(c'_i; \bar{\mathbf{v}}'_i, r'_i) = \{\bar{\mathbf{v}}'_i \leq \bar{\mathbf{v}}_{\mathsf{max}} \wedge c'_i = g^{\bar{\mathbf{v}}'_i} h^{r'_i}\}$.
3. Set $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \backslash \{c_1, c_2\} \cup \{c'_2\}$ and $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_{\mathsf{rcv}}] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_{\mathsf{rcv}}] \cup \{c'_1\}$.
4. Output updated $(\mathcal{L}, \gamma' = \mathcal{L}_{\mathsf{Conf}}, \bar{\mathbf{w}})$.

Redeem: parse x as $(\bar{\mathbf{v}}, c, \pi)$, $c \in \mathbb{G}$ and $\pi \in \{0,1\}^*$.
1. Assert $c \in \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}]$ and $\bar{\mathbf{w}} \geq \bar{\mathbf{v}}$.
2. Send call to $\mathcal{Q}$ with msg for $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ to verify π proves $\mathcal{R}(\bar{\mathbf{v}}, c; r) = \{c = \mathbf{g}^{\bar{\mathbf{v}}} h^r\}$.
3. Set $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \backslash \{c\}$, $\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} - \bar{\mathbf{v}}$.
4. Output updated $(\mathcal{L}, \gamma' = \mathcal{L}_{\mathsf{Conf}}, \bar{\mathbf{w}})$.

Fig. 11: The smart contract code $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ for confidential tokens.

I/II: Protocol $\mathit{\Pi}_{\underline{\mathsf{C}}\mathsf{Contract}}$, extends $\mathit{\Pi}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

All clients and servers are registered with $\mathcal{F}_{\mathsf{Clock}}$.

Init: On $(\textsc{Init}, sid, g)$ server $P \in \mathcal{P}$,

1. Runs **GenAcct** in $\mathit{\Pi}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ to generate signature verification key vk, sends to $\mathcal{P}$.
2. Runs **InitLedger** in $\mathit{\Pi}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ to initialize $\mathcal{F}_{\mathsf{Ledger}}$; here, $P \in \mathcal{P}$ obtains fresh vk.
3. Jointly samples key $\mathsf{vk}_{\mathsf{TSig}}$ via $\mathcal{F}_{\mathsf{TSig}}$ with $\mathcal{P}$.
4. Deploys $\mathcal{X}_{\mathsf{Lock}}[\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}]$ and $\mathcal{X}_{\mathsf{Collateral}}$ to $\mathcal{F}_{\mathsf{Ledger}}$.
   a. Obtains contract instance id's $\mathsf{cn}_{\mathsf{Lock}} = \mathsf{cn}_{\underline{\mathsf{C}}\mathsf{Ledger}}$ and $\mathsf{cn}_{\mathsf{Coll}}$ from $\mathcal{F}_{\mathsf{Ledger}}$.
   b. Signs and sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Init}), (\mathsf{vk}_{\mathsf{TSig}}), 0^{|\mathbb{T}|}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$.
   c. Sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Coll}}, f(\textsc{Deposit}), (\mathsf{vk}_{\mathsf{TSig}}), \bar{\mathbf{v}}_{\mathsf{Coll}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$.
5. Initializes $\mathcal{F}_{\mathsf{Ident}}$, asserts circuit depth of $\mathsf{depth}(g) \leq d_T$ and stores it.
6. Updates $\mathcal{F}_{\mathsf{Clock}}$.

Enroll: Upon input $(\textsc{Enroll}, sid, x, \mathsf{id}, \mathsf{vk})$, client $C \in \mathcal{C}$:

1. Asserts $\exists(\mathsf{id}, (\bar{\mathbf{v}}, \bar{r})) \in \mathcal{W}[\mathsf{vk}]$ and $\mathsf{cn}_{\mathsf{Lock}}$, $\mathsf{cn}_{\mathsf{Coll}}$ are in `enroll`/`coll`.
2. Generate output masks:
   a. Samples and stores $\hat{y}, \hat{\mathbf{w}} = (\hat{w}_1, ..., \hat{w}_{|\mathbb{T}|}), \hat{r} \leftarrow\$ \mathbb{F}$.
   b. Computes and stores $\hat{c} \leftarrow \mathsf{com}(\hat{\mathbf{w}}, \hat{s})$.
3. Sends client input and output masks $(x, (\bar{\mathbf{v}}, \bar{r}), (\hat{\mathbf{w}}, \hat{s}))$ to $\mathcal{F}_{\mathsf{Ident}}$.
4. Sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Enroll}), (c = \mathsf{com}(\bar{\mathbf{v}}, \bar{r}), \hat{c}), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$.
5. Removes $(\mathsf{id}, (\bar{\mathbf{v}}, \bar{r}))$ from $\mathcal{W}[\mathsf{vk}]$ and updates $\mathcal{F}_{\mathsf{Clock}}$.

Verify input: Upon input $(\textsc{Execute}, sid)$, if $\mathcal{F}_{\mathsf{Clock}}$ has progressed since last activation and $\mathsf{cn}_{\mathsf{Lock}}$, $\mathsf{cn}_{\mathsf{Coll}}$ are in `enrolled` and `coll` respectively, server $P_i \in \mathcal{P}$ performs:

1. P_i obtains client input coins and masks $\{(c_1, \hat{c}_1), ..., (c_m, \hat{c}_m)\}$ from $\mathcal{F}_{\mathsf{Ledger}}$.
2. For verification of client inputs $\{(\bar{\mathbf{v}}_j, \bar{r}_j, c_{j,})\}_{j \in [m]}$, P_i performs:
   a. Servers interact with $\mathcal{F}_{\mathsf{Ident}}$ and call following interfaces:
   - **Evaluate**: $[\,\bar{\mathbf{a}}\,], [\,\bar{b}\,], [\,\gamma\,] \leftarrow \mathsf{rand}()$[a]
   - **Open** $\gamma \leftarrow [\,\gamma\,]$.
   - **Get Shares**: $\bar{\mathbf{a}}^{(i)} = (\bar{a}_1^{(i)}, ..., \bar{a}_{|\mathbb{T}|}^{(i)}), \bar{b}^{(i)}, \{\bar{\mathbf{v}}_j^{(i)} = (\bar{v}_{j,1}^{(i)}, ..., \bar{v}_{j,|\mathbb{T}|}^{(i)}), \bar{r}_j^{(i)}\}_{j \in [m]}$

   a. Local computation of the following and sends resulting shares to all $\mathcal{P}$:
   - $\bar{c}_{\mathbf{a},b}^{(i)} \leftarrow \mathsf{com}(\bar{\mathbf{a}}^{(i)}, \bar{b}^{(i)})$, $\bar{v}_t^{(i)\prime} \leftarrow \bar{a}_t^{(i)} + \sum_{j \in [m]} \bar{v}_{j,t}^{(i)} (\gamma^{(i)})^j$ for $t \in [|\mathbb{T}|]$
   - $\bar{r}^{(i)\prime} \leftarrow \bar{b}^{(i)} + \sum_{j \in [m]} \bar{r}_j^{(i)} (\gamma^{(i)})^j$

   b. Each P_i reconstructs $(\bar{\mathbf{v}}' = \bar{v}_1', ..., \bar{v}_{|\mathbb{T}|}', \bar{r}')$, from shares and
   - Asserts: $\prod_{i \in [n]} \bar{c}_{\mathbf{a},b}^{(i)} \cdot \prod_{j \in [m]} (c_{j,\mathsf{in}})^{\gamma^j} = \mathbf{g}^{\bar{\mathbf{v}}'} h^{\bar{r}'}$
3. Servers repeats for the batch verification of client masks $\{(\hat{\mathbf{w}}_j, \hat{s}_j, \hat{c}_j)\}_{j \in [m]}$.
4. Server P_i updates $\mathcal{F}_{\mathsf{Clock}}$.

[a] e.g. XOR circuit evaluated on random inputs.

Fig. 12: Part 1 - Protocol $\mathit{\Pi}_{\underline{\mathsf{C}}\mathsf{Contract}}$ UC-securely realizing $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$.

II/II: Protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$, extends $\Pi_{\underline{\mathsf{C}}\mathsf{Ledger}}$

Evaluate: After *verify input* and if $\mathcal{F}_{\mathsf{Clock}}$ has progressed, $\mathsf{cn}_{\mathsf{Lock}}$ is in `enrolled`, each server $P_i \in \mathcal{P}$ performs:

1. It interacts with following interfaces of $\mathcal{F}_{\mathsf{Ident}}$.
   a. Runs **Evaluate** on circuit g with secret client inputs $\{(x_j, \bar{\mathbf{v}}_j)\}_{j\in[m]}$.
   b. Runs **Evaluate** to apply masks over output gate values of circuit g:
      - For each $j \in [m]$: $([\, y'_j\,], [\,\bar{\mathbf{w}}'_j\,]) \leftarrow ([\, y_j\,], [\,\bar{\mathbf{w}}_j\,]) + ([\,\hat{y}_j\,], [\,\hat{\mathbf{w}}_j\,])$
   c. Runs **Share** for obtain shares of $\{([\, y'_j\,], [\,\bar{\mathbf{w}}'_j\,])\}_{j\in[m]}$ from $\mathcal{F}_{\mathsf{Ident}}$.
2. Jointly signs $\sigma_{\mathsf{vk}_{\mathsf{TSig}}}(\mathsf{eval})$ with $\mathcal{P}$ via $\mathcal{F}_{\mathsf{TSig}}$: if $\mathcal{F}_{\mathsf{TSig}}$ aborts, runs **abort**.
3. Sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Lock}), (\sigma_{\mathsf{vk}_{\mathsf{TSig}}}(\mathsf{eval})), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$, updates $\mathcal{F}_{\mathsf{Clock}}$.

Open: Upon running *evaluate* and $\mathcal{F}_{\mathsf{Clock}}$ has progressed, each server $P_i \in \mathcal{P}$:

1. Runs **Optimistic Reveal** in $\mathcal{F}_{\mathsf{Ident}}$ for masked outputs $\mathsf{mout} = \{(y'_j, \bar{\mathbf{w}}'_j)\}_{j\in[m]}$.
2. Jointly signs $\mathsf{sig} = \sigma_{\mathsf{vk}_{\mathsf{TSig}}}(\mathsf{mout})$ via $\mathcal{F}_{\mathsf{TSig}}$: if *abort* is returned, run **abort**.
3. Sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Settle}), (\mathsf{mout}, \mathsf{sig}), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$, updates $\mathcal{F}_{\mathsf{Clock}}$.

Withdraw: Upon $(\textsc{Withdraw}, sid)$, each client $C_j \in \mathcal{C}$ performs:

1. Retrieves all masked outputs $\{(y'_{j,1}, \bar{\mathbf{w}}'_{j,1}), ..., (y'_{j,l}, \bar{\mathbf{w}}'_{j,l})\}$ added to $\mathsf{cn}_{\mathsf{Lock}}$ on $\mathcal{F}_{\mathsf{Ledger}}$ since last *withdraw* activation.
2. For each retrieved masked output, reads corresponding mask values $(\hat{y}_j, \hat{\mathbf{w}}_j, \hat{s}_j)$ stored locally and computes $y_j = y'_j - \hat{y}_j$, $\bar{\mathbf{w}}_j \leftarrow (\bar{\mathbf{w}}'_j - \hat{\mathbf{w}}_j)$.
   - Samples id'_j and set $\mathcal{W}[\mathsf{vk}_j] \leftarrow \mathcal{W}[\mathsf{vk}_j] \cup \{(\mathsf{id}'_j, (\bar{\mathbf{w}}_j, -\hat{s}_j))\}$.
3. Returns $(y_{j,1}, \langle \mathsf{id}'_{j,1}, \bar{\mathbf{w}}_{j,1}\rangle), ..., (y_{j,l}, \langle \mathsf{id}'_{j,l}, \bar{\mathbf{w}}_{j,l}\rangle)$.

Abort: Upon receiving $(\textsc{Abort}, sid)$, each $P_i \in \mathcal{P}$ ticks $\mathcal{F}_{\mathsf{Clock}}$:

1. If $\mathsf{cn}_{\mathsf{Lock}}$ is in state `enroll` or `enrolled`,
   signs and sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Abort}), \{0, \bar{\mathbf{0}}\}_{j\in[m]}), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$.
2. Else if $\mathsf{cn}_{\mathsf{Lock}}$ is in state `lock`, run **Reveal** and **Allow Verify** with $\mathcal{F}_{\mathsf{Ident}}$.
   - If $\mathcal{F}_{\mathsf{Ident}}$ returns cheating servers $\mathcal{J}$
     signs and sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Abort}), \{0, \bar{\mathbf{0}}\}_{j\in[m]}), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$.
   - Else server P_i obtains $\{y_j^{(i)}, \bar{\mathbf{w}}_j^{(i)}\}_{j\in[m]}$ from **Reveal**, signs and
     sends $(\textsc{Call}, sid, (\mathsf{cn}_{\mathsf{Lock}}, f(\textsc{Abort}), \{y_j^{(i)}, \bar{\mathbf{w}}_j^{(i)}\}_{j\in[m]}), \bar{\mathbf{0}}), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{Ledger}}$
3. Updates $\mathcal{F}_{\mathsf{Clock}}$ and *terminates.*

Fig. 13: Part 2 - Protocol $\Pi_{\underline{\mathsf{C}}\mathsf{Contract}}$ UC-securely realizing $\mathcal{F}_{\underline{\mathsf{C}}\mathsf{Contract}}$.

Program $\mathcal{X}_{\mathsf{Lock}}$, extends $\mathcal{X}_{\underline{\mathsf{C}}\mathsf{Ledger}}$

On input $(\nu \,|\, \gamma \,|\, \bar{\mathbf{w}} \,|\, \mathsf{cn}, \mathsf{fn}, x, \bar{\mathbf{v}}, \mathsf{vk})$, $\mathcal{X}_{\mathsf{Lock}}$ parses γ as $(\mathcal{L}_{\mathsf{Conf}}, \mathcal{L}_{\mathsf{Lock}}, \mathcal{M}, \mathcal{J}, \mathsf{st})$, where $\mathcal{L}_{\mathsf{Lock}}$ is a ledger of locked coins, $\mathcal{M}$ is a map of account verification keys to coin commitment masks, $\mathcal{J}$ is set of cheating servers and $\mathsf{st} \in \{\texttt{enroll}, \texttt{enroll}\underline{\texttt{ed}}, \texttt{lock}\}$ captures the phase the confidential coin lock is currently in.

Init lock: parse x as $\mathsf{vk}_{\mathsf{TSig}}$, set state to `enroll`.

Enroll: parse x as $(c, \hat{c}) \in \mathbb{G}^{\mathsf{com}} \times \mathbb{G}^{\mathsf{com}}$.
1. Assert state is `enroll` and $c \in \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}]$. Set $\mathcal{L}_{\mathsf{Lock}}[\mathsf{vk}] \leftarrow \mathcal{L}_{\mathsf{Lock}}[\mathsf{vk}] \cup \{c\}$, $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \setminus \{c\}$ and $\mathcal{M}[\mathsf{vk}] \leftarrow \hat{c}$.
2. If round ν has progressed since last state transition to `enroll`, set st to $\texttt{enroll}\underline{\texttt{ed}}$. Return updated $(\mathcal{L}, \gamma' = (\mathcal{L}_{\mathsf{Conf}}, \mathcal{L}_{\mathsf{Lock}}, \mathcal{M}, \mathcal{J}, \mathsf{st}), \bar{\mathrm{w}})$.

Evaluated: parse x as $\sigma_{\mathsf{vk}_{\mathsf{TSig}}}(\mathsf{eval}) \in \{0,1\}^*$.
1. Assert state st is $\texttt{enroll}\underline{\texttt{ed}}$, verify $\sigma_{\mathsf{vk}_{\mathsf{TSig}}}(\mathsf{eval})$ via $\mathcal{F}_{\mathsf{TSig}}$.
2. If no *abort* is returned and two $\mathcal{F}_{\mathsf{Clock}}$ rounds (observed via ν) have progressed since last state transition to $\texttt{enroll}\underline{\texttt{ed}}$: set st to `lock`.
3. Return updated $(\mathcal{L}, \gamma' = (\mathcal{L}_{\mathsf{Conf}}, \mathcal{L}_{\mathsf{Lock}}, \mathcal{M}, \mathcal{J}, \mathsf{st}), \bar{\mathrm{w}})$.

Settle: parse x as $\sigma_{\mathsf{vk}_{\mathsf{Lock}}}(\{y_j, \bar{\mathbf{v}}_j\}_{j \in [m]})$.
1. Assert state is `lock`, verify $\sigma_{\mathsf{vk}_{\mathsf{Lock}}}(\{y_j, \bar{\mathbf{v}}_j\}_{j \in [m]})$ via $\mathcal{F}_{\mathsf{TSig}}$.
2. If no *abort* is returned and $\mathcal{F}_{\mathsf{Clock}}$ round has progressed, run $\texttt{payout}(\{\bar{\mathbf{v}}_j\}_{j \in [m]})$.

Abort: parse x as $\{y_j^{(i)}, \bar{\mathbf{v}}_j^{(i)}\}_{j \in [m]}$.
1. If state is `enroll` or $\texttt{enroll}\underline{\texttt{ed}}$, run `reimburse`.
2. Else if state is `lock`,
   - Send (Test-Reveal, *sid*) and (Verify, *sid*, $\{y_j^{(i)}, \bar{\mathbf{v}}_j^{(i)}\}_{j \in [m]}$) to $\mathcal{F}_{\mathsf{Ident}}$.
   - If no *abort* is returned and $\mathcal{F}_{\mathsf{Clock}}$ round has progressed, reconstruct $\{\bar{\mathbf{v}}_j\}_{j \in [m]}$ when all shares received from $\mathcal{P}$ and run $\texttt{payout}(\{\bar{\mathbf{v}}_j\}_{j \in [m]})$.
   - Else if *abort* is returned from $\mathcal{F}_{\mathsf{Ident}}$ or shares are missing, record cheating servers $\mathcal{J}$ and run $\texttt{reimburse}(\mathcal{J})$.

$\texttt{payout}(\{\bar{\mathbf{v}}_j\}_{j \in [m]})$: for each C_j's output $\bar{\mathbf{v}}_j \in \{\bar{\mathbf{v}}_j\}_{j \in [m]}$,
1. $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_j] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}_j] \cup \{c_j \cdot \hat{c}_j^{-1}\}$, where $c_j = \mathbf{g}^{\bar{\mathbf{v}}_j} h^0$, $\hat{c}_j = \mathcal{M}[\mathsf{vk}_j]$. $\mathcal{L}_{\mathsf{Lock}} \leftarrow \emptyset$, $\mathcal{M} \leftarrow \emptyset$. Return updated $(\mathcal{L}, \gamma' = (\mathcal{L}_{\mathsf{Conf}}, \mathcal{L}_{\mathsf{Lock}}, \mathcal{M}, \mathcal{J}, \texttt{enroll}), \bar{\mathrm{w}})$.

$\texttt{reimburse}(\mathcal{J})$: for $\mathsf{vk} \in \mathsf{dom}(\mathcal{L}_{\mathsf{Lock}})$,
1. Set $\mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \leftarrow \mathcal{L}_{\mathsf{Conf}}[\mathsf{vk}] \cup \mathcal{L}_{\mathsf{Lock}}[\mathsf{vk}]$.
2. Set $\mathcal{L}_{\mathsf{Lock}} \leftarrow \emptyset$, $\mathcal{M} \leftarrow \emptyset$. Return updated $(\mathcal{L}, \gamma' = (\mathcal{L}_{\mathsf{Conf}}, \mathcal{L}_{\mathsf{Lock}}, \mathcal{M}, \mathcal{J}, \texttt{enroll}), \bar{\mathrm{w}})$.

Fig. 14: The smart contract code $\mathcal{X}_{\mathsf{Lock}}$ for extended confidential token functionality.

Program $\mathcal{X}_{\mathsf{Collateral}}$

Parameterized by signature verification keys $\{\mathsf{vk}_1, ..., \mathsf{vk}_n\}$ associated with servers $\mathcal{P} = \{P_1, ..., P_n\}$, contract identifier $\mathsf{cn}_{\mathsf{Lock}}$ and collateral threshold $\bar{\mathbf{v}}_{\mathsf{coll}}$.

Deposit collateral:
1. Assert local state is `deposit`, $\mathsf{cn}_{\mathsf{Lock}}$ state is `enroll`, and $\bar{\mathbf{v}}_{\mathsf{in}} \geq \bar{\mathbf{v}}_{\mathsf{coll}}$.
2. If collateral received by accounts associated with vk for $i \in [n]$, set state to `coll`.

Round activation: If $\mathcal{F}_{\mathsf{Clock}}$ round has progressed since update to `coll`.
1. If $\mathsf{cn}_{\mathsf{Lock}}$ is in `deposit`, return collateral and set state to `deposit`.
2. Else if $\mathsf{cn}_{\mathsf{Lock}}$ is in `abort` with cheating $\mathcal{J} \subseteq \mathcal{P}$, distribute $\mathcal{J}$'s collateral to $\mathcal{C}'$. Return collateral of honest servers. Set state to `deposit`.

Fig. 15: The smart contract code $\mathcal{X}_{\mathsf{Collateral}}$.