Merces: Confidential Token Transfers via MPC and CoSNARKs

Daniel Escudero[2], Florian Lugstein[1], Christian Rechberger[1,2], Verena Schröppel[1,2] and Roman Walch[2]

[1] ISEC, Graz University of Technology
firstname.lastname@tugraz.at
[2] TACEO, Graz, Austria
papers@taceo.io

Abstract. Fungible tokens on public blockchains expose all balances and transfer amounts in the clear, which is incompatible with the financial privacy required by many real-world applications. We present Merces, a confidential token contract that hides user balances and transaction amounts while preserving on-chain verifiability. The core idea is to store secret shares of balances within a decentralized MPC network, while only commitments are published to a smart contract. Thereby, Merces is capable of translating any existing token (e.g., any ERC20 token) into a confidential version. Deposits, withdrawals, and transfers are computed privately within the MPC network, which generates a collaborative SNARK (CoSNARK) to prove the validity of each state transition. In particular, the proof ensures that on-chain commitments are updated consistently and that the sender has sufficient funds. In Merces, privacy holds as long as the MPC servers do not collude and follow the protocol, whereas integrity—that funds move only on their owner's instruction and are never created—holds even if all MPC servers are actively corrupt, since no state transition takes effect without a proof verified on-chain. In this paper we give a full formalization of our construction in the Universal Composability (UC) framework, provide rigorous security proofs, and describe a concrete instantiation using Groth16 over BN254 with Poseidon2-based commitments. We further provide a complete end-to-end implementation, accompanied by extensive benchmarks and discussion of a working demo: our system achieves over 300 transactions per second, including proof generation, while requiring only minimal client-side computation.

Keywords: MPC · ZKP · CoSNARKs · Confidential Token · UC

1 Introduction

Blockchain-based technologies enable financial autonomy by decentralizing the governance and control of assets. Platforms such as Ethereum enable decentralized peer-to-peer payments without relying on centralized intermediaries. On top of Ethereum, a diverse ecosystem of decentralized applications (dApps) and decentralized finance (DeFi) protocols has emerged, forming a rapidly evolving financial infrastructure.

Within this ecosystem, the ERC-20 token standard plays a fundamental role. It defines a common interface for tokens implemented as smart contracts on the Ethereum blockchain, thereby establishing a baseline for interoperable decentralized financial systems. ERC-20 tokens represent transferable assets whose ownership and transactions are recorded on-chain.

However, Ethereum and most existing blockchains operate on a fully transparent ledger. All transactions, including (pseudonymous) sender and receiver addresses, transferred

amounts, and account balances, are publicly accessible. This transparency introduces significant privacy concerns: despite pseudonymity, prior work has shown the feasibility of linkability and deanonymization attacks. As a result, sensitive information about a user's financial behavior can be inferred, enabling surveillance and profiling.

These limitations highlight the need for stronger data protection mechanisms in decentralized financial systems. One approach is the concept of confidential tokens. The central idea is to extend ERC-20 tokens with privacy-preserving features by introducing a "shielded" layer. Users can deposit, and thereby "shield", a portion of their tokens into this private domain, where future transactions conceal both account balances and transferred amounts. Then, users can transact their shielded tokens while keeping the balances and the transacted amounts private. At any point, users may also "unshield" their assets, returning them to the standard ERC-20 contract and restoring full transparency.

Just as the integrity of ERC-20 tokens is protected by the integrity of the underlying Ethereum blockchain, shielded assets should be equally dependent on Ethereum's robustness. In particular, whoever operates the shielded layer off-chain should be unable to steal or create tokens, even by deviating arbitrarily from the protocol. Furthermore, the usual expected properties of an ERC-20 should hold: only users with knowledge of a given secret key can initiate a shielded transaction for a given address, and transactions can only take place if the sender has sufficient balance. In addition to these properties, privacy requirements mandate that both account balances and transferred amounts remain confidential.

1.1 Our Contribution

We present Merces, a confidential token contract that wraps existing ERC-20 tokens and hides user balances and transfer amounts (sender and receiver addresses remain public). Concretely, the paper makes the following contributions.

- **Blockchain-rooted integrity.** Funds cannot be stolen or created even if the entire MPC network is actively corrupt and deviates arbitrarily from the protocol: integrity is enforced on-chain by ZK verification, so only privacy depends on the MPC servers behaving honestly and not colluding. This is stronger than other MPC-server designs (e.g., Eagle [ByCDF23]) where integrity itself rests on MPC non-collusion.

- **Practical deployability.** By batching many user intents under a single CoSNARK proof, we amortize on-chain pairing verification across the batch and reach gas costs that make actual on-chain deployment realistic. Concretely, on a layer-2, gas costs are in the order of low single-digit cents, whereas on Ethereum mainnet amortized costs per transaction are $< 1\,\$$. Prior account-model systems with comparable privacy goals (e.g., Zether [BAZB20]) have not crossed this bar. Our proof-of-concept achieves a throughput of up to 300 transactions per second, including proof generation.

- **Lightweight clients.** Users only need to post a small intent together with a Poseidon2-friendly ZK proof of share consistency ($\sim$135 ms on a consumer laptop); the heavyweight transfer-validity proofs are produced inside the MPC network.

- **Full formalization and security proof.** We formalize Merces in the UC framework and prove that our protocol realizes an ideal confidential-token functionality $\mathcal{F}_{\mathsf{CToken}}$. The model distinguishes passive from active corruption of the MPC servers: privacy is guaranteed as long as at most t servers are corrupt, all of them passively, while integrity—that funds only move on their owner's instruction—holds against any number of corruptions of either kind. We prove security against static corruptions, and explain how the proof extends to adaptive ones. Integrity against the MPC servers would be vacuous if proven only for passive corruptions, since passively corrupt servers follow

the protocol; this is why the active case is treated explicitly in both the functionality and the proof. Our construction is moreover agnostic to the underlying MPC: any instantiation realizing the MPC functionality $\mathcal{F}_{\mathsf{MPC}}$ can be used. Deployments can thus pick the trust model that suits them (honest- or dishonest-majority, semi-honest or malicious), and advances in MPC translate directly into improvements of Merces.

- **End-to-end system and deployment study.** We provide a complete implementation, deploy a live demo on the Arc and Base testnets, and present a detailed treatment of practical deployment considerations — gas costs, censorship resistance, fail-safes against MPC denial-of-service, regulator-compliance hooks, and the client–MPC communication design — that bridges the gap between an idealized protocol and a deployable system.

1.2 Related Work

Various approaches for confidential on-chain payments have been proposed, with notable differences in security, performance, and practical feasibility. In this section we briefly describe some of the most relevant works. For a more thorough comparison we refer to Section A in the appendix. For detailed surveys, we refer to [BCMR24], [AS22].

Trusted Execution Environments (TEEs), such as those used in Oasis[1] and Phala[2], offer confidential smart contracts by executing computations inside hardware-protected enclaves. Yet, TEEs rely on hardware trust assumptions and are vulnerable to side-channel attacks, with numerous demonstrated vulnerabilities[3].

Fully Homomorphic Encryption (FHE) is explored in several works. PriFHEte [MS25] proposes encrypted on-chain transactions for full anonymity, but is purely theoretical. SmartFHE [SWA23] extends this by enabling multi-user contracts using multi-key FHE, but the cost of their on-chain computations is likely to be impractical. Zether [BAZB20] uses cheaper and weaker additive homomorphic encryption. They provide a full implementation, but it still suffers from high gas costs. Indeed, a transfer in Zether requires > 7 million units of gas, compared to our solution which requires < 1 million, split between user and MPC nodes (amortized). cERC20 [GWKH25] delegates FHE computations off-chain to mitigate performance limitations. Despite these innovations, FHE's computational costs remain restrictive for scalable systems [CGGI20].

Further, gcVM [YLRDG26] uses garbled circuits and homomorphic encryption to provide public auditability and privacy. However, this public auditability feature is not implemented, and is expected to result in high computational costs.

Zcash [BCG+14] and Monero [Mon26] provide private transfers in the UTXO model, but are limited to transactions only and require standalone cryptocurrencies. Zexe [BCG+20] extends Zcash to enable general-purpose private computation, but requires expensive on-chain computations. Jigsaw [GGKS26] builds on Zexe and uses MPC servers to execute privacy-preserving contracts. However, the UTXO model leads to large growth in blockchain storage and issues with note-discovery.

Hawk [KMS+16] similarly uses commitments and ZKPs but requires a central coordinator, which compromises privacy. zkHawk [BCT21] and Hawkness [BT22] replace the coordinator with MPC between the clients, which solves the privacy issue, but puts heavy load onto the clients. Eagle [ByCDF23] uses off-chain MPC servers to enable general-purpose private computation, but their transaction integrity relies on the MPC non-collusion assumption, resulting in weaker integrity guarantees than our approach.

An overview of existing approaches applying privacy to token transfers, compared across key design properties, is shown in Table 1. Integrity distinguishes whether correctness is enforced directly by the blockchain or relies on optimistic assumptions with external

[1] https://oasis.net/
[2] https://phala.com/
[3] https://sgx.fail/

Table 1: Overview of existing work on confidential token transfers. Categorization by Integrity (Blockchain / Optimistic), Privacy Owner (Off-chain server MPC / MPC among Clients / Off-chain server (F)HE / Client / Trusted party), Token Strategy (General smart contract / Wraps existing token / New token), Transaction Model (Account-based / UTXO), Implementation (Full / Partial / None).

Paper	Integrity	Privacy Owner	Token Strat.	Trans. Model	Impl.
cERC20 [GWKH25]	Optimistic	FHE	Wraps	Account	Partial
Monero [Mon26]	Blockchain	Client	New	UTXO	Full
ZCash [BCG+14]	Blockchain	Client	New	UTXO	Full
Zexe [BCG+20]	Blockchain	Client	General	UTXO	Full
Jigsaw [GGKS26]	Blockchain	Server MPC	General	UTXO	Full
Eagle [ByCDF23]	Optimistic	Server MPC	General	UTXO	None
Hawk [KMS+16]	Blockchain	Trusted	General	UTXO	Full
zkHawk [BCT21]	Blockchain	Client MPC	General	UTXO	None
Hawkness [BT22]	Blockchain	Client MPC	General	UTXO	None
gcVM [YLRDG26]	Blockchain	Client	General	Account	Partial
SmartFHE [SWA23]	Blockchain	Client	General	Account	Partial
PriFHEte [MS25]	Blockchain	Client	Wraps	Account	None
Zether [BAZB20]	Blockchain	Client	General	Account	Full
Merces (ours)	Blockchain	Server MPC	General	Account	Full

verification. Privacy owner captures where private data is protected and processed, e.g., client-side, MPC among clients or servers, FHE-based outsourcing, or trusted parties. This property also strongly influences how well a system can extend to computations involving multiple data holders and how well properties such as delegated spending can be realized. Token strategy differentiates between systems that support general smart contracts, wrap existing tokens (e.g., ERC-20), or introduce new native tokens. Transaction model contrasts account-based designs with UTXO-based approaches. Finally, Implementation indicates the maturity of each proposal, ranging from fully implemented systems to partial prototypes or purely theoretical constructions.

Finally, we note that formalizing the notion of a smart-contract-enabled blockchain is not new, and has indeed been approached by multiple prior works. Kachina [KKK21b] presents a formal treatment, focusing on *private* smart contracts. SmartFHE [SWA23] also models this but in the game-based setting, which in contrast to UC-based simulation, offers weaker guarantees when it comes to composability. Finally, closer to us there is Eagle [ByCDF23], which presents a functionality for *public* smart contracts, which is similar to what we model here. Then they define a *private* smart contract functionality that builds on top of their public one. This is comparable to our approach, except we solely focus on confidential transfers.

1.3 Technical Overview

We describe Merces at a high level; the workflow is visualized in Figure 1. Three entities are involved: (i) users, who hold a private balance and can deposit, withdraw, and transfer tokens without leaking the transferred amount; (ii) a smart contract holding the pool of all deposited tokens together with commitments to user balances; and (iii) an MPC network holding secret shares of the actual balances and of the blinding values used to compute the on-chain commitments.

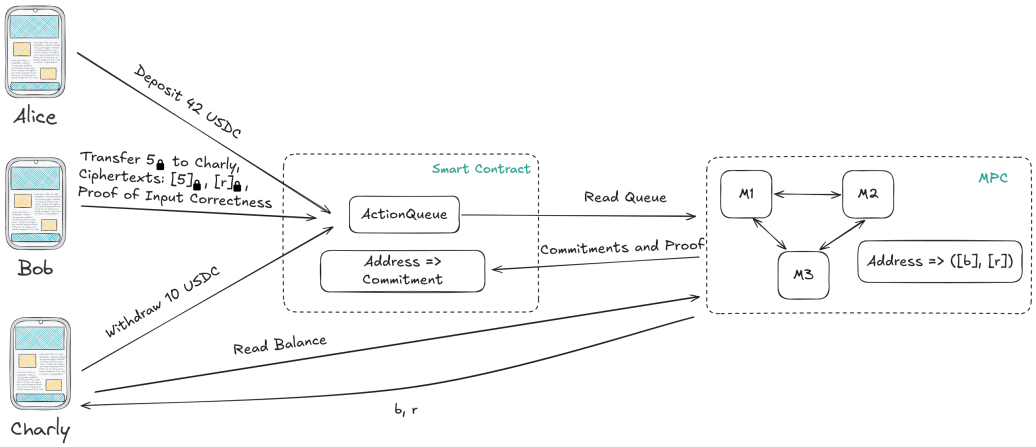


Figure 1: Overview of the MERCES workflow.

The smart contract maintains a hashmap (e.g., a mapping in Solidity)[4] from user addresses to balance commitments, which serves as the ground truth of user balances. Whenever a user initiates an action (deposit, withdraw, or transfer), the smart contract checks validity of the intent (e.g., a withdrawn amount must be non-negative) and registers it in a queue, which is continuously monitored by the MPC network. The MPC network processes the queue by updating the secret-shared balances of the involved parties, sampling fresh secret-shared blinding values, and computing the updated commitments. It then produces a zero-knowledge proof attesting that the commitments were updated correctly according to the registered user intent, and that the new balance of a sender in a transfer or withdraw is non-negative. When the smart contract receives the updated balance commitments from the MPC network, it verifies the zero-knowledge proof and, on success, updates the on-chain commitments of the involved users. Since no commitment changes without a proof the contract has verified, a misbehaving MPC network—even one in which every server is actively corrupt—cannot move or create funds: it can at most break privacy, by pooling or leaking the shares it holds, or stall the queue.

The MERCES smart contract holds the tokens corresponding to the sum of the secret-shared balances of all users. Whenever a user registers a deposit intent, it must transfer the corresponding tokens to the smart contract. Conversely, after the MPC network verifies the validity of a withdraw (i.e., the user had enough balance to fulfill it), the smart contract pays out the tokens to the intended recipient. Consequently, the amount cannot be hidden in a deposit or withdraw, since an actual token transfer happens on-chain. Confidential transfers, in contrast, hide the amount: the user posts a commitment of the intended amount on-chain and provides secret shares of the amount (and blinding value) to the MPC network. In our implementation, the user posts these secret shares, in encrypted form, to the smart contract, where each MPC party can only decrypt its designated share. We further require the client to post a zero-knowledge proof attesting that the committed amount is non-negative, secret-shared correctly, and encrypted under the MPC network's public key. This rules out malicious client requests at the contract level, which is essential for preventing censorship by the MPC network (see Section 6).

The description above reflects how MERCES is actually deployed. The formal protocol Π_{CToken} (Section 4) abstracts two deployment-level details: client-to-MPC communication is modeled directly via the MPC ideal functionality $\mathcal{F}_{\mathsf{MPC}}$ rather than routed through the smart contract as encrypted ciphertexts, and, accordingly, the client's NIZK only proves that the committed amount is in range, while the MPC network itself checks that

[4] https://docs.soliditylang.org

the on-chain amount commitment is consistent with the amount and blinding it receives via $\mathcal{F}_{\mathsf{MPC}}$. Neither deviation affects the security guarantees; the concrete realization — including the encrypted-share design, the client NIZK, batching, and gas-cost optimizations — is discussed in Section 6 and detailed in Section B.

1.4 Outline of the Paper

Section 2 presents the preliminaries necessary to understand the rest of the paper, including the UC framework and the ideal functionalities for MPC, smart contracts, zero-knowledge proofs, and commitments. Section 3 formalizes the ideal functionality $\mathcal{F}_{\mathsf{CToken}}$ for confidential token transfers, which captures the security guarantees we aim for. Section 4 presents Merces, our protocol realizing $\mathcal{F}_{\mathsf{CToken}}$, including the smart contract, MPC procedures, and ZK relations, and Section 5 proves its security against static corruptions and discusses the extension to adaptive ones. Section 6 discusses the concrete instantiation of Merces with 3-party replicated secret-sharing and Groth16 over BN254, together with benchmarks and gas-cost measurements. Section 6.5 describes a working end-to-end demo deployed on the Arc and Base testnets. Finally, Section 7 discusses how Merces generalizes beyond confidential transfers, and concludes with future work. The appendix contains an extended related-work discussion and the concrete practicalities of our implementation.

2 Preliminaries

This section introduces our model and the building blocks of Merces. Besides the blockchain itself, which we model as an ideal functionality, our protocol uses only standard primitives: secure multiparty computation, which only needs to withstand passive corruptions, NIZK arguments that are zero-knowledge and (weakly) simulation-extractable, and commitments with the properties of UC commitments.

2.1 Universal Composability Framework

We prove security within the universal composability (UC) framework, following the formulation of Canetti [Can01, Can20]. The framework models a protocol execution as a system of interactive Turing machines, parameterized by a security parameter κ.

Protocols, procedures, and functionalities. A *protocol* Π is specified as a collection of interactive Turing machines, one per party, that exchange messages over an authenticated network. An *ideal functionality* $\mathcal{F}$ is itself an interactive Turing machine that interacts with all parties and with a designated adversary; it captures the behavior we want Π to realize, including any unavoidable leakage to the adversary, and is treated as a trusted third party in the analysis. A protocol may invoke other functionalities $\mathcal{F}'$ as oracles — the so-called *$\mathcal{F}'$-hybrid model* — in which case parties send messages to $\mathcal{F}'$ instead of executing a subprotocol. Finally, similar to a functionality, a *procedure* π is a subroutine that a protocol may invoke, except that its behavior is not abstracted away and instead the whole protocol with the "unfolded procedure" is analyzed; this is comparable to a *macro* in programming languages.

The environment and the two worlds. Security is defined in the simulation paradigm. A PPT adversary $\mathcal{A}$ controls a subset of *corrupt* parties (see below). The *environment* $\mathcal{Z}$ is an additional PPT machine that chooses inputs for all parties, observes their outputs, and interacts with $\mathcal{A}$ throughout the execution. Two execution experiments are defined:

- **Real-world execution** $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa)$: $\mathcal{Z}$ activates the parties of Π on its chosen inputs. Honest parties run Π, while $\mathcal{A}$ controls the corrupt parties and the network scheduling. At the end, $\mathcal{Z}$ outputs a single bit.
- **Ideal-world execution** $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa)$: the parties are replaced by *dummy parties* that forward their inputs from $\mathcal{Z}$ to $\mathcal{F}$ and forward $\mathcal{F}$'s outputs back. A PPT *simulator* $\mathcal{S}$ plays the role of $\mathcal{A}$, with access only to the leakage interface $\mathcal{F}$ exposes for corrupt parties. $\mathcal{Z}$ again outputs a single bit.

UC emulation. A protocol Π *UC-realizes* (or *instantiates*) the functionality $\mathcal{F}$ if for every PPT real-world adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that, for every PPT environment $\mathcal{Z}$,

$$\Big|\Pr[\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa)=1]-\Pr[\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa)=1]\Big|\leq \mathsf{negl}(\kappa).$$

Composition. The central technical feature of the framework is the *universal composition theorem*: if Π UC-realizes $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model and Π' UC-realizes $\mathcal{F}'$, then the composed protocol $\Pi^{\Pi'}$, obtained by replacing every invocation of $\mathcal{F}'$ in Π with an execution of Π', UC-realizes $\mathcal{F}$. This allows us to design and analyze each subprotocol against its own ideal abstraction and then assemble the full system without re-doing the proof.

Participants and corruptions. Throughout the paper, we work with two disjoint sets of participants: a set of *users* $\mathcal{U}$, who are the clients of the system, and a set of *MPC parties* $\mathcal{P}=\{P_1,\ldots,P_n\}$, who jointly maintain the secret-shared state and run the MPC subprotocols on the users' behalf. Users and MPC parties can send messages to each other over authenticated point-to-point channels: the adversary sees every message and may delay or drop it, but cannot modify it or inject messages on behalf of honest parties. Formally, this is the $\mathcal{F}_{\mathsf{AUTH}}$-hybrid model [Can01], which we leave implicit throughout; we simply write that a party sends a message to another. The environment $\mathcal{Z}$ may corrupt any subset of $\mathcal{U}\cup\mathcal{P}$. Corruptions are *static*: the corrupt parties, and the mode of each corrupt MPC party, are fixed at the beginning of the execution (Section 5.1 discusses adaptive corruptions). An MPC party is corrupt in one of two *modes*: *passively*, in which case it keeps following the protocol but its internal state and all messages it sends and receives are revealed to the adversary; or *actively*, in which case the adversary controls it fully and it may deviate arbitrarily from the protocol. Considering passive and active corruptions at the same time is standard in MPC [FHM98]. We may assume corrupt users are always active. The corruption mode matters only for the MPC parties, where it determines whether privacy is preserved (Section 2.2); integrity, as we shall see, never depends on it.

The designated user U_{merces}. One user plays a special role: a designated user $U_{\mathsf{merces}}\in\mathcal{U}$ deploys our contract and relays the results computed by the MPC parties to the chain. Since it acts on the MPC parties' behalf, we place it in their trust domain: we only consider environments in which U_{merces} deploys the correct contract, and which corrupt U_{merces} only if some MPC party is actively corrupt (see Section 4.1).

2.2 Secure Multiparty Computation

We model secure computation through a generic reactive functionality $\mathcal{F}_{\mathsf{MPC}}$, in the style of the arithmetic black box of [DN03], shared by a set of MPC parties $\mathcal{P}=\{P_1,\ldots,P_n\}$ and providing service to the users $\mathcal{U}$. As is common in MPC, we consider computation over a finite field $\mathbb{F}_p$, whose elements we think of as *signed* integers in $[-(p-1)/2,(p-1)/2]$.

$\mathcal{F}_{\mathsf{MPC}}$ maintains an internal store mapping handle IDs to values in $\mathbb{F}_p$; we write $[\![x]\!]$ to denote that x is held under some handle, suppressing the ID when irrelevant. Users may push secret values into the store via $(\textsc{PrivInput}, x)$, and the parties $\mathcal{P}$ may collectively invoke $(\textsc{Compute}, f, \ldots)$ to apply an arbitrary function[5] $f : \mathbb{F}_p^\ell \to \mathbb{F}_p^m$ to stored handles, $(\textsc{Rand}, \mathsf{hid})$ to sample fresh secret randomness, and $(\textsc{Output}, \mathsf{hid}, Q)$ to reconstruct a stored value, either towards the MPC parties ($Q = \mathcal{P}$) or privately towards a user ($Q = U$).

The functionality is parameterized by a privacy threshold $t < n$, and operates in one of two regimes, depending on the set of corrupt parties. If at most t MPC parties are corrupt, *all of them passively*, we say that *privacy holds*: $\mathcal{F}_{\mathsf{MPC}}$ is a normal arithmetic black box, where stored values remain hidden from the adversary—corrupt parties only see the values that the protocol explicitly opens—and all values are computed and opened correctly. If more than t parties are corrupt, or *at least one* party is actively corrupt, we say that *privacy is lost*: $\mathcal{F}_{\mathsf{MPC}}$ leaks every value it stores, and lets the adversary choose every value it outputs. In this regime "all bets are off" inside the MPC: the values it outputs need not bear any relation to the intended computation. Our integrity claims hold in this regime as well, since integrity is enforced on-chain and never rests on $\mathcal{F}_{\mathsf{MPC}}$. We present $\mathcal{F}_{\mathsf{MPC}}$ formally as Functionality 1.

Functionality 1: $\mathcal{F}_{\mathsf{MPC}}$

$\mathcal{F}_{\mathsf{MPC}}$ interacts with the MPC parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, the users $\mathcal{U}$ and an adversary, and is parameterized by a threshold $t < n$. It keeps a dictionary mapping *handle IDs* hid to values $x \in \mathbb{F}_p$, and we write $[\![x]\!]$ when x is stored (omitting the ID hid from the notation). It also keeps a flag leaked, which is 1 if more than t MPC parties are corrupt or at least one of them is actively corrupt, and 0 otherwise.

Privacy is lost if more than t parties are corrupt, or if a single party deviates from the protocol. In that case, the adversary sees every stored value and chooses every output.

Storage and computation.

- On input $(\textsc{PrivInput}, x)$ from a user $U \in \mathcal{U}$, store $[\![x]\!]$ and send the corresponding hid to the parties in $\mathcal{P}$.
- On input $(\textsc{Rand}, \mathsf{hid})$ from the parties in $\mathcal{P}$, sample $x \leftarrow_\$ \mathbb{F}_p$ and store $[\![x]\!]$ in ID hid.
- On input $(\textsc{Compute}, f, (\mathsf{iid}_i)_{i\in[\ell]}, (\mathsf{oid}_j)_{j\in[m]})$ from the parties in $\mathcal{P}$, where $f : \mathbb{F}_p^\ell \to \mathbb{F}_p^m$, fetch $([\![x_i]\!])_{i\in[\ell]}$ from the IDs $(\mathsf{iid}_i)_{i\in[\ell]}$, compute $(y_1, \ldots, y_m) = f(x_1, \ldots, x_\ell)$, and store $([\![y_j]\!])_{j\in[m]}$ in $(\mathsf{oid}_j)_{j\in[m]}$.

If $\mathsf{leaked} = 1$, every value stored in this way is also sent to the adversary.

Output. On input $(\textsc{Output}, \mathsf{hid}, Q)$ from the parties in $\mathcal{P}$, where Q is either $\mathcal{P}$ or a user $U \in \mathcal{U}$, fetch $[\![x]\!]$ from ID hid and send x to Q. If $\mathsf{leaked} = 1$, send instead a value x' chosen by the adversary.

Instantiation. If privacy holds, $\mathcal{F}_{\mathsf{MPC}}$ is the standard arithmetic black box [DN03], which protocols based on secret sharing realize against static passive corruptions of up to $t < n/2$ parties [BGW88, AFL+16]; if privacy is lost, $\mathcal{F}_{\mathsf{MPC}}$ gives the adversary full control. Hence, $\mathcal{F}_{\mathsf{MPC}}$ requires no security against active corruptions. Our implementation uses three-party replicated secret sharing [AFL+16, MR18] (Section 6).

Why we give up privacy entirely. Giving up on privacy entirely on a single active corruption is deliberately conservative; the actual loss is protocol-dependent and typically much more restricted than a complete leak: e.g., in the honest-majority setting it is

[5] The most expensive functions we evaluate are commitments and NIZK proofs. For the former, our instantiation uses MPC-friendly hash-based commitments, and for the latter a SNARK, namely Groth16; computing SNARK proofs in MPC is known as *collaborative SNARKs* [OB22], or *CoSNARKs*.

restricted to so-called *additive errors* [GIP15], though for dishonest majority the concrete leakage is harder to bound. For computing a proof, Garg et al. [GGJ+25] even show that, with an honest majority, existing passively secure protocols for Groth16 and other SNARKs are, with minor changes, already secure against active corruptions. Our procedures, however, also compute commitments, check the validity of witnesses, and open values along the way, and an additive error introduced there can leak information: e.g., an actively corrupt party could shift a balance before the solvency check, and learn from whether a proof is produced whether the balance exceeds some threshold. Rather than bounding such leakage, we treat any active corruption as a complete loss of privacy; integrity is unaffected.

Security with abort. For our formalization, we work in the model of *security with abort*: in contrast to notions such as fairness or guaranteed output delivery, the adversary may cause an abort at any point of the protocol execution. For $\mathcal{F}_{\mathsf{MPC}}$, this is formalized by letting the adversary send an abort command that halts the functionality; we do not write it in Functionality 1, and implicitly assume the adversary can issue it. Our other functionalities have no such command: $\mathcal{F}_{\mathsf{CRS}}$ is a setup functionality, $\mathcal{F}_{\mathsf{SmartContr}}$ models a public blockchain that no one can halt, and for our main functionality $\mathcal{F}_{\mathsf{CToken}}$ we explain the reason in Section 3.

2.3 Blockchains and Smart Contracts

Recall that our goal is to add privacy to an existing token on a smart-contract-enabled blockchain, with privacy ensured by the non-collusion of the MPC parties but integrity guaranteed even if all of them are corrupt. A prime example of such a blockchain is *Ethereum*,[6] an *account-based* blockchain in which a dictionary maps users to their balances (in ETH), and in which *smart contracts*—programs with their own state—can be deployed and executed to alter the state of the blockchain.

As is common in the literature [KMS+16, BMTZ17, ByCDF23], we model the underlying blockchain as an ideal functionality $\mathcal{F}_{\mathsf{SmartContr}}$, available to all users $\mathcal{U}$; its Read command is also available to the MPC parties $\mathcal{P}$, who hold no tokens. We identify users with blockchain addresses: $\mathcal{U}$ ranges over all addresses, and every address can hold and receive tokens. $\mathcal{F}_{\mathsf{SmartContr}}$ maintains an append-only ledger $\mathcal{L}$, a balance map $\mathsf{bal} : \mathcal{U} \cup \{\mathsf{cid}\} \to \mathbb{N}$ for users and deployed contracts, and a contract registry $\mathcal{C}$ mapping each contract identifier cid to a pair $(\Phi_{\mathsf{cid}}, \mathsf{st}_{\mathsf{cid}})$ of contract code and current state. Users interact with $\mathcal{F}_{\mathsf{SmartContr}}$ through four commands: $(\textsc{Deploy}, \mathsf{cid}, \Phi, \mathsf{st}_0)$ uploads a new contract; $(\textsc{Transfer}, R, v)$ relocates v native tokens to a recipient R; $(\textsc{Execute}, \mathsf{cid}, \tau, v)$ invokes a deployed contract on input τ, optionally attaching v tokens, and lets the contract update its state and (optionally) trigger an outgoing transfer; and $(\textsc{Read})$ returns the full public state to the caller. All accepted Deploy, Transfer and Execute commands are appended to $\mathcal{L}$, so contract code, contract state, and ledger history are visible to every party—this is the standard public-bulletin model of a smart-contract chain. We abstract away from transaction ordering on the chain: $\mathcal{F}_{\mathsf{SmartContr}}$ processes each command upon receipt. Since every state transition of our contract is checked against the current on-chain state, adversarial ordering could only delay or reorder requests, which our functionality lets the adversary do anyway. We present $\mathcal{F}_{\mathsf{SmartContr}}$ formally as Functionality 2.

[6] https://ethereum.org/

Functionality 2: $\mathcal{F}_{\mathsf{SmartContr}}$

$\mathcal{F}_{\mathsf{SmartContr}}$ interacts with a set of users $\mathcal{U}$, the MPC parties $\mathcal{P}$ (only through Read), and an adversary $\mathcal{A}$. It maintains:

- an append-only ledger $\mathcal{L}$, initially empty;
- a map $\mathcal{C}$ from smart contract identifiers cid to pairs $(\Phi_{\mathsf{cid}}, \mathsf{st}_{\mathsf{cid}})$, initially $\mathcal{C} = \emptyset$;
- a balance map $\mathsf{bal} : \mathcal{U} \cup \{\mathsf{cid} : \mathsf{cid} \in \mathcal{C}\} \to \mathbb{N}$, with initial values chosen by the environment.

Together, $(\mathcal{L}, \mathcal{C}, \mathsf{bal})$ constitute the global state of $\mathcal{F}_{\mathsf{SmartContr}}$.

On input $(\textsc{Deploy}, \mathsf{cid}, \Phi, \mathsf{st}_0)$ from a user $U \in \mathcal{U}$:

- If $\mathsf{cid} \in \mathcal{C}$, ignore the message.
- Otherwise, set $\mathcal{C}[\mathsf{cid}] \leftarrow (\Phi, \mathsf{st}_0)$, initialize $\mathsf{bal}[\mathsf{cid}] \leftarrow 0$, and append $(\textsc{Deploy}, \mathsf{cid}, \Phi, \mathsf{st}_0)$ to $\mathcal{L}$.

On input $(\textsc{Transfer}, R, v)$ from a user $U \in \mathcal{U}$, where $R \in \mathcal{U}$ and $v \in \mathbb{N}$:

- If $\mathsf{bal}[U] < v$, ignore the message.
- Otherwise, set $\mathsf{bal}[U] \leftarrow \mathsf{bal}[U] - v$ and $\mathsf{bal}[R] \leftarrow \mathsf{bal}[R] + v$, and append $(\textsc{Transfer}, U, R, v)$ to $\mathcal{L}$.

On input $(\textsc{Execute}, \mathsf{cid}, \tau, v)$ from a user $U \in \mathcal{U}$, where $v \in \mathbb{N}$:
Executing a contract also allows the user to *optionally* transfer funds to the contract.

- If $\mathsf{cid} \notin \mathcal{C}$, or if $0 < v$ and $\mathsf{bal}[U] < v$, ignore the message.
- Otherwise, let $(\Phi, \mathsf{st}) \leftarrow \mathcal{C}[\mathsf{cid}]$ and compute $b \leftarrow \Phi(\mathsf{st}, \tau, U, v)$.
- If $b = (\mathsf{accept}, \mathsf{st}', R, v')$ with $v' \le \mathsf{bal}[\mathsf{cid}] + v$:
  - Set $\mathsf{bal}[U] \leftarrow \mathsf{bal}[U] - v$ and $\mathsf{bal}[\mathsf{cid}] \leftarrow \mathsf{bal}[\mathsf{cid}] + v$.
  - If $v' > 0$: set $\mathsf{bal}[\mathsf{cid}] \leftarrow \mathsf{bal}[\mathsf{cid}] - v'$ and $\mathsf{bal}[R] \leftarrow \mathsf{bal}[R] + v'$.
    The contract can *optionally* transfer funds to another account.
  - Update $\mathcal{C}[\mathsf{cid}] \leftarrow (\Phi, \mathsf{st}')$ and append $(\textsc{Execute}, \mathsf{cid}, \tau, U, v, R, v')$ to $\mathcal{L}$.
- If $b = \mathsf{reject}$, ignore the message (no balance changes).

On input (Read) from a user $U \in \mathcal{U}$ or an MPC party $P_i \in \mathcal{P}$:

- Return $(\mathcal{L}, \mathcal{C}, \mathsf{bal})$ to the caller.

Amounts and balances. Amounts and balances are integers, which the MPC parties and the commitments hold as elements of $\mathbb{F}_p$, read as signed integers (Section 2.2); comparisons such as $v \le b$ are between these integers. To make sure that no computation wraps around modulo p, we bound both. First, every amount v that is deposited, withdrawn, or transferred confidentially satisfies $0 \le v < B$ for a fixed bound $B \ll p$. $\mathcal{F}_{\mathsf{SmartContr}}$ need not enforce this, whereas $\mathcal{F}_{\mathsf{CToken}}$ ignores requests with $v \ge B$ (Section 3), and our protocol enforces it with a direct check when the amount is public (deposits and withdrawals), and with a range proof by the user when it is private (confidential transfers; Section 4.4). Second, we assume that the total supply of the token is below $p/2$. Since private balances never become negative and are backed by the tokens held in our contract, they then always lie in $[0, p/2)$, and crediting a balance never wraps around. In our instantiation (Section 6), $B = 2^{80}$ and $p \approx 2^{254}$, whereas the total supply of, e.g., ETH is about 2^{87} wei.

Modeling choices. The tokens captured by $\mathcal{F}_{\mathsf{SmartContr}}$ are the blockchain's native currency (ETH in the case of Ethereum): they are part of the blockchain specification itself, and

$\mathcal{F}_{\mathsf{SmartContr}}$ directly exposes commands to transact on them. Alternatively, tokens can be built as smart contracts themselves, which in practice correspond to ERC20 tokens[7] and are a major application of Ethereum's expressiveness. These can be captured by a variant of $\mathcal{F}_{\mathsf{SmartContr}}$ with a "hardcoded" contract that keeps balances and enables transfers. We choose to formalize the native token for clarity, but our approach applies equally to tokens such as ERC20. Contracts may also hold a balance in $\mathcal{F}_{\mathsf{SmartContr}}$, and users can send tokens to a contract by attaching them to an EXECUTE call; this is the mechanism we will use to let users turn public tokens into private ones. Finally, $\mathcal{F}_{\mathsf{SmartContr}}$ abstracts away several aspects of real blockchains: it is a local functionality rather than a global one shared with other protocols, it processes each command immediately and gives every party the same view of the chain (no blocks, consensus or forks), it identifies users with addresses rather than signature keys, and it supports a single native token and no fees. We refer to [BMTZ17] for a composable treatment of consensus, and to [ByCDF23, Appendix C] for a global ledger functionality with smart contracts, which also includes a confidential token ledger in a setting reasonably close to ours.

2.4 Zero-Knowledge Proofs

We use non-interactive zero-knowledge (NIZK) arguments of knowledge in the common reference string (CRS) model, introduced by Blum, Feldman and Micali [BFM88]. A NIZK for a relation $\mathcal{R}$ consists of the tuple $(\mathtt{NIZK}_{\mathcal{R}}.\mathtt{keygen}, \mathtt{NIZK}_{\mathcal{R}}.\mathtt{prove}, \mathtt{NIZK}_{\mathcal{R}}.\mathtt{verify})$. At the core, we have a statement-witness pair $(x, w) \in \mathcal{R}$, where x is the statement and w is the witness. Let $\mathcal{L}_{\mathcal{R}} \in \mathsf{NP}$ be the language with the associated witness relation $\mathcal{R}$, that is, $x \in \mathcal{L}_{\mathcal{R}}$ if and only if there exists w such that $(x, w) \in \mathcal{R}$. These algorithms have the following syntax:

- $\mathsf{crs}_{\mathcal{R}} \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{keygen}(1^{\kappa})$: A key generation algorithm that returns a CRS $\mathsf{crs}_{\mathcal{R}}$ based on the security parameter κ.
- $\pi \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}}, x, w)$: A proof generation algorithm that produces a proof π from the statement x and the witness w given the CRS $\mathsf{crs}_{\mathcal{R}}$. If $(x, w) \notin \mathcal{R}$, the algorithm returns $\perp$.[8]
- $\mathsf{accept}/\mathsf{reject} \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}}, x, \pi)$: A verification algorithm that returns accept if the proof π is valid for the given CRS and statement x, and reject otherwise.

A NIZK argument of knowledge must satisfy the following properties:

- **Completeness**: honestly computed proofs for valid statement-witness pairs are always accepted. Formally, for any PPT adversary $\mathcal{A}$ that outputs $(x, w) \in \mathcal{R}$,

$$\Pr\left[\mathsf{crs}_{\mathcal{R}} \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{keygen}(1^{\kappa});\ (x, w) \leftarrow \mathcal{A}(\mathsf{crs}_{\mathcal{R}});\ \pi \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}}, x, w) : \mathtt{NIZK}_{\mathcal{R}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}}, x, \pi) = \mathsf{accept}\right] = 1.$$

- **Zero-knowledge** [BFM88, FLS90]: there exist PPT simulators $(S_{1,\mathcal{R}}, S_{2,\mathcal{R}})$, where $S_{1,\mathcal{R}}(1^{\kappa})$ outputs a simulated CRS $\mathsf{crs}_{\mathcal{R}}$ together with a trapdoor $\mathsf{td}_{\mathcal{R}}$, and $S_{2,\mathcal{R}}(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}, x)$ outputs a simulated proof π for x. Consider the following oracles, which both return $\perp$ on inputs $(x, w) \notin \mathcal{R}$:

[7] https://ethereum.org/developers/docs/standards/tokens/erc-20/

[8] The convention that $\mathtt{NIZK}_{\mathcal{R}}.\mathtt{prove}$ returns $\perp$ on inputs $(x, w) \notin \mathcal{R}$ is without loss of generality: when $\mathcal{R}$ is expressed as an arithmetic circuit or R1CS, it is realized by checking constraint satisfaction on the extended witness before invoking the cryptographic prover. In our CoSNARK instantiation, this check is performed in MPC during witness extension, and the MPC parties compute no proof when $(x, [\![w]\!]) \notin \mathcal{R}$. This protects the witness even when the statement turns out to be false, since a proof computed from an invalid witness may leak information about it [GGJ+25]; e.g., when a corrupt user requests a transfer exceeding its balance, the witness still contains the receiver's balance.

– $\mathcal{O}_P(x, w)$ returns $\pi \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}}, x, w)$;
– $\mathcal{O}_S(x, w)$ returns $\pi \leftarrow S_{2,\mathcal{R}}(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}, x)$, without using w.

We require that for every PPT adversary $\mathcal{A}$, which may query its oracle polynomially many times and adaptively,

$$\Big|\Pr\left[\,\mathsf{crs}_{\mathcal{R}} \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{keygen}(1^\kappa);\ \mathcal{A}^{\mathcal{O}_P}(\mathsf{crs}_{\mathcal{R}}) = 1\,\right] - \Pr\left[\,(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}) \leftarrow S_{1,\mathcal{R}}(1^\kappa);\ \mathcal{A}^{\mathcal{O}_S}(\mathsf{crs}_{\mathcal{R}}) = 1\,\right]\Big| \leq \mathsf{negl}(\kappa).$$

This is the standard notion of multi-theorem zero-knowledge, in which the adversary chooses its statements adaptively [FLS90]; Section 5.1 discusses the stronger notion needed under adaptive corruptions.

- **Weak simulation-extractability** [GM17, BKSV21]: for every PPT adversary $\mathcal{A}$ there is a PPT extractor $E_{\mathcal{A}}$, which receives the view of $\mathcal{A}$ (its input, random coins and oracle answers), such that

$$\Pr\left[(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}) \leftarrow S_{1,\mathcal{R}}(1^\kappa);\ (x, \pi) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{sim}}}(\mathsf{crs}_{\mathcal{R}});\ w \leftarrow E_{\mathcal{A}} : \mathtt{NIZK}_{\mathcal{R}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}}, x, \pi) = \mathsf{accept} \ \wedge\ x \notin Q \ \wedge\ (x, w) \notin \mathcal{R}\right] \leq \mathsf{negl}(\kappa),$$

where $\mathcal{O}_{\mathsf{sim}}(x)$ returns the proof output by $S_{2,\mathcal{R}}(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}, x)$, and Q is the set of statements queried to $\mathcal{O}_{\mathsf{sim}}$.[9] Without oracle queries, this is knowledge soundness, which in turn implies soundness. We need the stronger notion because in our simulation the adversary produces its proofs after seeing simulated ones, under a simulated CRS.[10] Finally, the extractor $E_{\mathcal{A}}$ is *non-black-box*: it uses the code and view of $\mathcal{A}$ (for Groth16, the algebraic representation of the group elements it outputs [BKSV21]), so a UC simulator, which interacts with the environment straight-line, cannot run it. This is not a problem, since we use simulation-extractability only to bound the probability of bad events in the analysis, never inside the simulator.

Instantiation. Groth16 [Gro16], the SNARK we use, has all these properties. It is perfectly complete and perfectly zero-knowledge, and it is weakly simulation-extractable in the algebraic group model [BKSV21]. More generally, simulation-extractability is available for SNARKs at little cost, either by design [GM17] or through generic compilers [ARS20].

Setup. In our protocols and proofs we make use of a setup functionality $\mathcal{F}_{\mathsf{CRS}}$ (Functionality 3) that distributes the CRSs of the NIZKs, as well as the public parameters of the commitment scheme below, to all parties. In our security proof, the simulator emulates this functionality internally, which lets it sample *trapdoored* CRSs and commitment parameters (with $S_{1,\mathcal{R}}$ above and $\mathsf{TdSetup}$ below). Such setup functionalities are standard: NIZK arguments require a CRS [BFM88], and UC commitments cannot be realized without some setup [CF01]. In practice, the CRS of Groth16 is generated by a multi-party ceremony, which is secure as long as one of its participants is honest [BGM17, KMSV21], and the parameters of the hash-based commitment we use amount to the choice of the hash function.

[9] The notion is *weak* because $\mathcal{A}$ may still turn a simulated proof into another proof for the same statement, e.g., by rerandomizing it; *strong* simulation-extractability [DDO+01, GM17], which builds on the simulation soundness of [Sah99], rules this out as well. We do not need it, since in our protocol the effect of a proof is determined by its statement, which the contract computes itself.

[10] Knowledge soundness says nothing about such an adversary: it could, e.g., maul a simulated proof, which the simulator may have produced for a statement it has no witness for, into a proof for a related statement. Nor can a reduction to knowledge soundness produce the simulated proofs, since it does not have the trapdoor.

Functionality 3: $\mathcal{F}_{\mathsf{CRS}}$

- On input $(\mathrm{CRS}, \mathcal{R})$ from the MPC parties, where $\mathcal{R}$ is a relation, sample $\mathsf{crs}_{\mathcal{R}} \leftarrow \mathtt{NIZK}_{\mathcal{R}}.\mathtt{keygen}(1^{\kappa})$ (only the first time), and send $\mathsf{crs}_{\mathcal{R}}$ to all MPC parties and users.
- On input (ComSetup) from the MPC parties, sample $\mathsf{pp} \leftarrow \mathtt{setup}(1^{\kappa})$ (only the first time), and send pp to all MPC parties and users.

2.5 Cryptographic Commitments

A cryptographic commitment scheme consists of the tuple $(\mathtt{setup}, \mathtt{commit}, \mathtt{open})$. At its core, we have a commitment $c \in C$ (commitment space) to a message $m \in \mathbb{F}_p$ using commitment randomness $r \in \mathbb{F}_p$, so that both can be held by $\mathcal{F}_{\mathsf{MPC}}$. The algorithms are defined as:

- $\mathsf{pp} \leftarrow \mathtt{setup}(1^{\kappa})$: A setup algorithm that returns *public parameters* pp based on the security parameter κ.
- $c \leftarrow \mathtt{commit}_{\mathsf{pp}}(m; r)$: A commitment algorithm that produces a commitment c to message m using commitment randomness r and the *public parameters* pp.
- $\mathsf{accept/reject} \leftarrow \mathtt{open}_{\mathsf{pp}}(c, m, r)$: An algorithm that verifies whether (m, r) is a valid opening of c under the *public parameters* pp.

We will use the notation $\mathtt{commit}(m; r)$ throughout the rest of the paper, leaving the public parameters pp implicit.

A cryptographic commitment must satisfy the following properties:

- **Correctness:** For all pp and all $m, r \in \mathbb{F}_p$, a valid commitment must verify under opening. In other words, if $c = \mathtt{commit}_{\mathsf{pp}}(m; r)$, then $\mathtt{open}_{\mathsf{pp}}(c, m, r) = \mathsf{accept}$.
- **Binding:** A computationally bounded adversary $\mathcal{A}$ can only find two distinct message-randomness pairs (m, r) and (m', r') with $m \neq m'$ that result in the same valid commitment c with negligible probability. Formally, for any probabilistic polynomial-time adversary $\mathcal{A}$, it holds that:

$$\Pr\Big[\mathsf{pp} \leftarrow \mathtt{setup}(1^{\kappa}),\ (c, m, r, m', r') \leftarrow \mathcal{A}(\mathsf{pp}) : \\ \mathtt{open}_{\mathsf{pp}}(c, m, r) = \mathsf{accept} \ \wedge\ \mathtt{open}_{\mathsf{pp}}(c, m', r') = \mathsf{accept} \ \wedge\ m \neq m'\Big] \leq \mathsf{negl}(\kappa).$$

- **Hiding:** A computationally bounded adversary $\mathcal{A}$ cannot distinguish between commitments to two chosen messages except with negligible advantage. Formally, for any probabilistic polynomial-time adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, it holds that:

$$\Pr\Big[\mathsf{pp} \leftarrow \mathtt{setup}(1^{\kappa}),\ (m_0, m_1, \mathsf{state}) \leftarrow \mathcal{A}_1(\mathsf{pp}),\ b \xleftarrow{\$} \{0,1\}, \\ r \xleftarrow{\$} \mathbb{F}_p,\ c \leftarrow \mathtt{commit}_{\mathsf{pp}}(m_b; r),\ \tilde{b} \leftarrow \mathcal{A}_2(\mathsf{state}, c) : b = \tilde{b}\Big] \leq \frac{1}{2} + \mathsf{negl}(\kappa).$$

In addition, we require the scheme to be *equivocable* and *extractable*. Our simulator needs both. If privacy holds, it posts commitments on behalf of honest parties without knowing their contents, e.g., to the new balance of the receiver of a transfer; if the receiver is corrupt, the simulator learns this balance only after the transfer has been executed, and must then open the commitment to it when the receiver queries its balance (equivocation).

Conversely, when a corrupt user registers a confidential transfer, the simulator must learn the committed amount right away, to hand it to $\mathcal{F}_{\mathsf{CToken}}$ (extraction). These are exactly the two abilities of the simulator of a UC commitment, which must extract the values committed to by corrupt parties, and equivocate its own commitments, without rewinding the environment [CF01, Section 1.2].[11] Formally, there are PPT algorithms $\mathsf{TdSetup}$, EqCom and EqOpen, and a deterministic polynomial-time algorithm Ext, such that $\mathsf{TdSetup}(1^\kappa)$ outputs public parameters pp, distributed as those output by $\mathtt{setup}(1^\kappa)$, together with a trapdoor td. Consider the following two oracles, which an adversary may query polynomially many times and adaptively:

- $\mathcal{O}_{\mathsf{Com}}$, on query $(\textsc{Commit}, m)$, samples $r \leftarrow_{\$} \mathbb{F}_p$, records (c, m, r) for $c = \mathtt{commit}(m; r)$, and returns c; on query $(\textsc{Open}, c)$ for a recorded (c, m, r), it returns r.
- $\mathcal{O}_{\mathsf{Eq}}$, on query $(\textsc{Commit}, m)$, computes $(c, \mathsf{aux}) \leftarrow \mathsf{EqCom}(\mathsf{td})$, records (c, m, aux), and returns c; on query $(\textsc{Open}, c)$ for a recorded (c, m, aux), it returns $r \leftarrow \mathsf{EqOpen}(\mathsf{td}, \mathsf{aux}, m)$, and marks c as opened to m.

Note that $\mathcal{O}_{\mathsf{Eq}}$ chooses each commitment independently of the message, which it only uses when opening. We require:

- **Equivocability:** for every PPT adversary $\mathcal{A}$,

$$\Big|\Pr\left[\mathsf{pp} \leftarrow \mathtt{setup}(1^\kappa) : \mathcal{A}^{\mathcal{O}_{\mathsf{Com}}}(\mathsf{pp}) = 1\right] - \Pr\left[(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{TdSetup}(1^\kappa) : \mathcal{A}^{\mathcal{O}_{\mathsf{Eq}}}(\mathsf{pp}) = 1\right]\Big| \leq \mathsf{negl}(\kappa).$$

- **Extractability:** for every PPT adversary $\mathcal{A}$, the following probability is negligible:

$$\Pr\left[(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{TdSetup}(1^\kappa);\ (c, m, r) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Eq}}}(\mathsf{pp}) :\ \mathtt{open}(c, m, r) = \mathsf{accept} \ \wedge\ \mathsf{Bad}\right],$$

  where Bad holds if either c was not returned by $\mathcal{O}_{\mathsf{Eq}}$ and $m \neq \mathsf{Ext}(\mathsf{td}, c)$, or c was returned by $\mathcal{O}_{\mathsf{Eq}}$ but not opened by it to m.

In words, commitments produced with the trapdoor can be opened to any message, but only by the trapdoor holder, and the message of any other commitment that the adversary can open can be read off it with the trapdoor. Equivocable commitments go back to [Bea96, DIO98]. Our game-based formulation is close to the mixed commitments of [DN02, Section 3], and its requirement that only the trapdoor holder can open the commitments produced with the trapdoor corresponds to simulation-sound trapdoor commitments [GMY03, MY04]. Note that extraction is *straight-line*: Ext only gets the trapdoor and the commitment, and neither rewinds the adversary nor uses its code. This is essential: unlike the extractor of the NIZK (Section 2.4), which we only use in the analysis, our simulator runs Ext during the simulation, while interacting with the environment.

Instantiation. These properties are easy to achieve: since they are exactly the abilities of its simulator, any non-interactive UC commitment scheme provides them, and the trapdoor only needs to be known to the simulator, which generates the setup. In the CRS model, such schemes exist under standard assumptions, e.g., by combining an encryption scheme, whose decryption key enables extraction, with a trapdoor commitment, whose trapdoor enables equivocation [CF01, Section 3.2]; see also [DN02] for efficient constructions. In the programmable random-oracle model, even the plain hash-based commitment

[11] We cannot use the ideal commitment functionality of [CF01] directly: in our protocol, commitments are strings that the contract stores, that appear in the statements of the NIZKs, and that are computed inside $\mathcal{F}_{\mathsf{MPC}}$ from secret-shared values, so we need a scheme with explicit algorithms.

$\texttt{commit}(m;r) = H(m\|r)$, which we use, is a UC commitment [HM04, CDG+18], so our requirements come at no additional cost. There, pp amounts to the choice of the hash function, and the simulator's control over the random oracle serves as the trapdoor: EqCom outputs a random c, EqOpen programs the oracle so that $H(m\|r) = c$ for a fresh r, and Ext looks up c among the adversary's queries to H, which the simulator observes. We discuss modeling the hash function of our implementation, Poseidon2 [GKS23], as a random oracle in Section 6.

3 Formalizing Confidential Token Transfers

We formalize confidential token transfers as the functionality $\mathcal{F}_{\mathsf{CToken}}$ below. The functionality works as $\mathcal{F}_{\mathsf{SmartContr}}$ — it keeps *public* balances of the native token, allows for smart contracts, and enables interaction with these — but additionally maintains an internal dictionary of balances that are intended to remain private. $\mathcal{F}_{\mathsf{CToken}}$ exposes three new functions:

- Deposit and Withdraw, which allow users to move back and forth between their public tokens and their "private" ones;
- ConfTransfer, which enables a user to send some private tokens to another user.

The ConfTransfer call works similarly to Transfer in $\mathcal{F}_{\mathsf{SmartContr}}$, but crucially, while the identities of sender and receiver are appended to the ledger, the transferred amount is *not* — this is what models the privacy requirement.

$\mathcal{F}_{\mathsf{CToken}}$ captures the following intuition: "even if *all* MPC parties are corrupt and deviate arbitrarily from the protocol, the adversary cannot modify balances or create invalid transactions". Private balances change only when a request registered by a user is executed, and an execution debits a private balance only if its owner requested it and has sufficient funds; this holds at every corruption level. The adversary's influence is thus limited to privacy and liveness. Privacy is lost in $\mathcal{F}_{\mathsf{CToken}}$ under the same conditions as in $\mathcal{F}_{\mathsf{MPC}}$ (Section 2.2): either more than t MPC parties are corrupt, or at least one of them is actively corrupt. Regarding liveness, requests do not take effect upon arrival: $\mathcal{F}_{\mathsf{CToken}}$ registers each request and lets the adversary decide when, and whether, it is executed. This reflects that requests are processed off-chain by the MPC parties, and that a corrupt network may refuse to process them, or process them in an order of its choosing.[12] Requests are either pending (submitted but not yet executed) or executed (processed, with the private balances updated accordingly). Finally, $\mathcal{F}_{\mathsf{CToken}}$ lets each user query its own private balance.[13]

If privacy holds, the adversary therefore learns only what is public on-chain—the identifiers of all requests, the senders and receivers of transfers, the amounts of deposits and withdrawals, and when requests are executed—together with the private balances of corrupt users and the times at which honest users query theirs; in particular, a corrupt receiver learns the amounts it receives, as the receiver of any payment would. If privacy is lost, $\mathcal{F}_{\mathsf{CToken}}$ reveals all amounts, and thereby all balances, but integrity is unaffected: every execution is still a request of the user whose balance it debits.

[12] We model this by letting $\mathcal{F}_{\mathsf{CToken}}$ keep a list of requests tagged by user/query-ID pairs. Each request carries an identifier qid chosen by the user, which cannot be reused, and under which the request is recorded on the ledger.

[13] Honest users rely on this to avoid spending more than they have, which for the sake of clarity in our formalism we assume they never do (see below); for corrupt users, it determines what the adversary learns about their balances, including the amounts they receive.

Functionality 4: $\mathcal{F}_{\mathsf{CToken}}$

$\mathcal{F}_{\mathsf{CToken}}$ works exactly as $\mathcal{F}_{\mathsf{SmartContr}}$: it interacts with the users $\mathcal{U}$ and an adversary $\mathcal{A}$, maintains the ledger $\mathcal{L}$, the contract map $\mathcal{C}$ and the balance map $\mathsf{bal} : \mathcal{U} \cup \{\mathsf{cid} : \mathsf{cid} \in \mathcal{C}\} \to \mathbb{N}$, and supports DEPLOY, TRANSFER, EXECUTE and READ with the same semantics. $\mathcal{F}_{\mathsf{CToken}}$ is parameterized by a threshold t, an amount bound B (Section 2.3), and a reserved contract identifier $\mathsf{cid}_{\mathsf{merces}}$, which cannot be deployed.

$\mathsf{cid}_{\mathsf{merces}}$ is the identifier under which our protocol deploys its contract. Since it is never deployed in $\mathcal{F}_{\mathsf{CToken}}$, EXECUTE commands referring to it are ignored.

In addition, $\mathcal{F}_{\mathsf{CToken}}$ keeps the following state:

- A *private balance* map $\mathsf{priv\text{-}bal} : \mathcal{U} \to \mathbb{N}$, initially 0 everywhere, which is kept internally.
- A map req from pairs (U, qid) to registered requests, each with a status in $\{\mathsf{pending}, \mathsf{executed}\}$.
- A flag leaked, which is 1 if more than t MPC parties are corrupt or at least one of them is actively corrupt, and 0 otherwise.

  Privacy is lost if more than t parties are corrupt, or if a single party deviates from the protocol. In that case, the adversary learns the amount of every transfer, and thereby every private balance.

Requests. In addition to the functions supported by $\mathcal{F}_{\mathsf{SmartContr}}$, $\mathcal{F}_{\mathsf{CToken}}$ also supports the following.

On input $(\text{DEPOSIT}, \mathsf{qid}, v)$ from a user $U \in \mathcal{U}$:

- If $\mathsf{req}[(U, \mathsf{qid})]$ is set, $v \geq B$, or $\mathsf{bal}[U] < v$, ignore the message.
- Otherwise, set $\mathsf{bal}[U] \leftarrow \mathsf{bal}[U] - v$ and $\mathsf{req}[(U, \mathsf{qid})] \leftarrow (\text{DEPOSIT}, v, \mathsf{pending})$, append $(\text{DEPOSIT}, U, \mathsf{qid}, v)$ to $\mathcal{L}$, and send it to the adversary.

  The tokens leave U's *public* balance immediately when the request is registered. The private credit instead awaits execution.

On input $(\text{WITHDRAW}, \mathsf{qid}, v)$ from a user $U \in \mathcal{U}$:

- If $\mathsf{req}[(U, \mathsf{qid})]$ is set or $v \geq B$, ignore the message.
- Otherwise, set $\mathsf{req}[(U, \mathsf{qid})] \leftarrow (\text{WITHDRAW}, v, \mathsf{pending})$, append $(\text{WITHDRAW}, U, \mathsf{qid}, v)$ to $\mathcal{L}$, and send it to the adversary.

On input $(\text{CONFTRANSFER}, \mathsf{qid}, R, v)$ from a user $U \in \mathcal{U}$, where $R \in \mathcal{U}$:

- If $\mathsf{req}[(U, \mathsf{qid})]$ is set, $R = U$, or $v \geq B$, ignore the message.
- Otherwise, set $\mathsf{req}[(U, \mathsf{qid})] \leftarrow (\text{CONFTRANSFER}, R, v, \mathsf{pending})$, append $(\text{CONFTRANSFER}, U, \mathsf{qid}, R)$ to $\mathcal{L}$, and send it to the adversary; if $\mathsf{leaked} = 1$, also send v.

  The amount v is neither appended to the ledger nor, if privacy holds, sent to the adversary; a corrupt receiver can learn it only through its own balance.

Execution. On input $(\text{PROCEED}, U, \mathsf{qid})$ from the adversary, where $\mathsf{req}[(U, \mathsf{qid})]$ is $\mathsf{pending}$:

- If the request debits an amount v from U (withdrawal or confidential transfer) and $\mathsf{priv\text{-}bal}[U] < v$, ignore the message.
- Otherwise, execute the request, and set its status to $\mathsf{executed}$:
  - $(\text{DEPOSIT}, v)$: set $\mathsf{priv\text{-}bal}[U] \leftarrow \mathsf{priv\text{-}bal}[U] + v$.
  - $(\text{WITHDRAW}, v)$: set $\mathsf{priv\text{-}bal}[U] \leftarrow \mathsf{priv\text{-}bal}[U] - v$ and $\mathsf{bal}[U] \leftarrow \mathsf{bal}[U] + v$.
  - $(\text{CONFTRANSFER}, R, v)$: set $\mathsf{priv\text{-}bal}[U] \leftarrow \mathsf{priv\text{-}bal}[U] - v$ and $\mathsf{priv\text{-}bal}[R] \leftarrow \mathsf{priv\text{-}bal}[R] + v$.

  Append $(\text{EXECUTED}, U, \mathsf{qid})$ to $\mathcal{L}$.

  Executions are the only way private balances change, and they never debit more than the available balance. Since honest users do not overdraw (see below), their requests are always solvent. A request that is never executed has no effect, except that the tokens of a pending deposit remain held.

Balance queries. On input (BALANCE) from a user $U \in \mathcal{U}$:

- If U is corrupt, send $\mathsf{priv\text{-}bal}[U]$ to the adversary.
- Otherwise, record a pending balance query of U and send (BALANCE, U) to the adversary.

On input (RESPOND, U) from the adversary, where U has a pending balance query: remove it and output $\mathsf{priv\text{-}bal}[U]$ to U. If $\mathsf{leaked} = 1$, the adversary may instead send (RESPOND, $U, \perp$), in which case U outputs $\perp$.

The adversary decides when a query is answered, and the answer is the balance at that time. If privacy is lost, the adversary may also make the answer $\perp$, but never a wrong balance.

Honest users do not overdraw. For simplicity we only consider environments in which honest users never request to move more private tokens than they have available. Concretely, an honest user U receives an input (WITHDRAW, qid, v) or (CONFTRANSFER, qid, R, v) only if v, plus the amounts of all pending withdrawals and confidential transfers of U, is at most $\mathsf{priv\text{-}bal}[U]$. Since only U's own requests debit $\mathsf{priv\text{-}bal}[U]$, all requests of U then remain solvent, in whichever order they are executed. The environment can evaluate this condition, since it chooses all amounts and sees on the ledger which requests are executed. Honest users can check their balance with BALANCE before submitting a request; how our deployment handles requests that turn out to be insolvent nonetheless is discussed in Section 6. Corrupt users may still submit insolvent requests, which are simply not executed.

Liveness and aborts. The adversary decides when, and whether, each request is executed, and likewise when balance queries are answered. This is unavoidable when requests are processed by an off-chain network that may refuse to serve them, and it is also why $\mathcal{F}_{\mathsf{CToken}}$, unlike $\mathcal{F}_{\mathsf{MPC}}$, has no global abort (cf. Section 2.2): halting $\mathcal{F}_{\mathsf{CToken}}$ altogether would also halt the public blockchain it models, whereas a corrupt network can only refuse to process some requests, or process them in an order of its choosing. A request that is never executed leaves all balances unchanged, with one exception: a pending deposit has already moved its tokens out of the user's public balance, so a network that refuses to process deposits freezes these tokens—they are neither stolen nor credited to anyone else. In practice, a network that can stall the system in this way may be unacceptable from a usability perspective. Formally, this could be addressed with notions from MPC with guaranteed output delivery, which we avoid for the sake of simplicity. Our deployment instead mitigates it with a fail-safe that lets users withdraw their committed balance directly from the contract, without the MPC servers, once the network has stopped processing requests for some time (Section 6); formalizing it would require a notion of time, which the UC model we use does not have, and we leave it informal.

4 Merces: Confidential Tokens

In this section we present our main protocol for confidential token transfers. We choose a top-down approach where we begin in Section 4.1 by presenting our protocol, Π_{CToken}, which instantiates Functionality $\mathcal{F}_{\mathsf{CToken}}$ in the $(\mathcal{F}_{\mathsf{SmartContr}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{CRS}})$-hybrid model. This makes use of several building blocks that are explained in subsequent sections. In Section 4.2 we present the smart contract code Φ_{merces} that gets deployed in the $\mathcal{F}_{\mathsf{SmartContr}}$ functionality. Also, Π_{CToken} uses some internal routines π_{Deposit}, π_{Withdraw} and $\pi_{\mathsf{ConfTransfer}}$, which are presented in Section 4.3. Finally, these procedures will produce NIZKs corresponding to certain statements, which are described in detail in Section 4.4.

4.1 Main Confidential Token Protocol

We present our protocol Π_{CToken} for confidential transactions. The contract Φ_{merces} from Section 4.2 is uploaded into $\mathcal{F}_{\mathsf{SmartContr}}$ via the DEPLOY command. In our model contract uploads must be done by a user, and for this we use the designated user $U_{\mathsf{merces}} \in \mathcal{U}$ of Section 2.1, which deploys Φ_{merces}. In practice, U_{merces} corresponds to the organization that deploys the MERCES contract and operates the MPC network. It holds no secret state: the contract code is public, and the calls it posts carry only public values, each of which the contract verifies. We assume U_{merces} deploys the correct contract; this is innocuous in practice, since deployed contract bytecode is publicly auditable.

In addition to deploying the contract, U_{merces} is also responsible of "representing" the MPC parties towards the chain $\mathcal{F}_{\mathsf{SmartContr}}$. Concretely, $\mathcal{F}_{\mathsf{SmartContr}}$ is designed so that only users can post calls to it, and we chose to model MPC parties separately from users. However, the MPC parties need to interact with the chain: e.g. reading pending requests, or posting the results of their computations. For this, though we could let each MPC node also play the role of a user, we favor simplicity and let U_{merces} act as a special user that interacts with both the MPC parties and $\mathcal{F}_{\mathsf{SmartContr}}$, relaying requests. For exameple... Finally, since U_{merces} is like a "front-end" to the MPC network, and since it handles no private data, we assume U_{merces} is (actively) corrupt only if some MPC party is actively corrupt. This is a restriction we will place in the environment when we write the proof.

DE: TODO finish this paragraph with an example of how we write U_{merces} relaying stuff in the text

Before we head in, note that each request carries a query ID qid, which is part of the user's input (as in $\mathcal{F}_{\mathsf{CToken}}$) and under which the contract records the request; a user never reuses a query ID. The user then sends qid to the MPC parties, which fetch the request from the contract. Our actual deployment is simpler (see Section 6.2): the MPC servers regularly scan the contract state and learn when a user posts a request.[14]

Protocol 1: Π_{CToken}

Setup.

- The MPC parties $\mathcal{P}$ call $\mathcal{F}_{\mathsf{CRS}}$ on input $(\mathrm{CRS}, \mathcal{R})$ for each $\mathcal{R} \in \{\mathcal{R}_{\mathsf{deposit}}, \mathcal{R}_{\mathsf{withdraw}}, \mathcal{R}_{\mathsf{transfer}}, \mathcal{R}_{\mathsf{user}}\}$ (see Section 4.4), and on input (COMSETUP). All parties receive the CRSs $\mathsf{crs}_{\mathcal{R}_{\mathsf{deposit}}}, \mathsf{crs}_{\mathcal{R}_{\mathsf{withdraw}}}, \mathsf{crs}_{\mathcal{R}_{\mathsf{transfer}}}, \mathsf{crs}_{\mathcal{R}_{\mathsf{user}}}$ and the public parameters pp of the commitment scheme, which we leave implicit from now on.
- U_{merces} calls $(\text{DEPLOY}, \mathsf{cid}_{\mathsf{merces}}, \Phi_{\mathsf{merces}}, \mathsf{st}_0 = \emptyset)$ on $\mathcal{F}_{\mathsf{SmartContr}}$ to deploy the contract Φ_{merces} from Section 4.2 with an initial empty state.
- The MPC parties $\mathcal{P}$ initialize two mappings $\mathsf{conf\text{-}bal} = \{U \to [\![0]\!]\}_{U \in \mathcal{U}}$ and $\mathsf{com\text{-}rand} = \{U \to [\![0]\!]\}_{U \in \mathcal{U}}$ in $\mathcal{F}_{\mathsf{MPC}}$.
  These match the default on-chain commitment $\mathtt{commit}(0; 0)$ of every user.

Public commands. The commands of $\mathcal{F}_{\mathsf{SmartContr}}$ are forwarded to $\mathcal{F}_{\mathsf{SmartContr}}$, with two exceptions. DEPLOY and EXECUTE commands referring to $\mathsf{cid}_{\mathsf{merces}}$ are ignored: users interact with Φ_{merces} only through the commands below. On input (READ), U calls READ on $\mathcal{F}_{\mathsf{SmartContr}}$ and outputs the result in the format of $\mathcal{F}_{\mathsf{CToken}}$, hiding Φ_{merces} and translating the calls to it into the corresponding ledger entries of $\mathcal{F}_{\mathsf{CToken}}$ (Remark 1).

Requests. On each of the inputs below, U ignores the input if it has already used qid, or if $v \geq B$.

If user U receives input $(\text{DEPOSIT}, \mathsf{qid}, v)$ from the environment:

1. U calls READ on $\mathcal{F}_{\mathsf{SmartContr}}$, and ignores the input if $\mathsf{bal}[U] < v$. Otherwise, it issues $(\text{EXECUTE}, \mathsf{cid}_{\mathsf{merces}}, \tau, v)$ to $\mathcal{F}_{\mathsf{SmartContr}}$, where $\tau = (\text{INITDEPOSIT}, \mathsf{qid})$.
   This transfers v tokens to the smart contract Φ_{merces}.

[14] Modeling this in the UC framework is cumbersome, since the model is event-driven with no notion of time; query IDs let the user "inform" the MPC parties when and where to look for a pending request.

2. U calls π_{Deposit} on input qid.
   The MPC parties $\mathcal{P}$ fetch the request, compute the new commitment and a proof, and post them through U_{merces} (Section 4.3.1).

If user U receives input $(\textsc{Withdraw}, \mathsf{qid}, v)$ from the environment:

1. U issues $(\textsc{Execute}, \mathsf{cid}_{\mathsf{merces}}, \tau, 0)$ to $\mathcal{F}_{\mathsf{SmartContr}}$, where $\tau = (\textsc{InitWithdraw}, \mathsf{qid}, v)$.
2. U calls π_{Withdraw} on input qid.
   As for deposits (Section 4.3.1).

If user U receives input $(\textsc{ConfTransfer}, \mathsf{qid}, R, v)$ from the environment, and $R \neq U$:

1. U samples $r_{\mathsf{transfer}} \leftarrow_{\$} \mathbb{F}_p$ and computes $\gamma_{\mathsf{transfer}} = \mathtt{commit}(v; r_{\mathsf{transfer}})$.
2. U computes $\pi_{\mathsf{user}} \leftarrow \mathtt{NIZK}_{\mathcal{R}_{\mathsf{user}}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}_{\mathsf{user}}}, x, w)$, where $x = (\gamma_{\mathsf{transfer}}, U)$ and $w = (v, r_{\mathsf{transfer}})$.
   This proves that $0 \leq v < B$ (Section 2.3). Since the statement contains U, the proof is bound to its sender: no other user can reuse it in a request of its own.
3. U issues $(\textsc{Execute}, \mathsf{cid}_{\mathsf{merces}}, \tau, 0)$ to $\mathcal{F}_{\mathsf{SmartContr}}$, where $\tau = (\textsc{InitTransfer}, \mathsf{qid}, R, \pi_{\mathsf{user}}, \gamma_{\mathsf{transfer}})$.
4. U calls $\pi_{\mathsf{ConfTransfer}}$ on input $(\mathsf{qid}, v, r_{\mathsf{transfer}})$.
   The MPC parties $\mathcal{P}$ receive shares of v and r_{transfer}, check them against $\gamma_{\mathsf{transfer}}$, compute the new commitments and a proof, and post them through U_{merces} (Section 4.3.2).

Balance queries. If user U receives input $(\textsc{Balance})$ from the environment, it calls π_{Balance} and outputs the result.
The MPC parties open U's balance and blinding randomness towards U, who checks them against its on-chain commitment (Section 4.3.3).

Remark 1 (Reading the public state). Since $\mathcal{F}_{\mathsf{CToken}}$ has no contract for MERCES, outputting the raw state of $\mathcal{F}_{\mathsf{SmartContr}}$ would let the environment tell the two worlds apart. U therefore removes $\mathsf{cid}_{\mathsf{merces}}$ from the contract and balance maps and its deployment from the ledger, and translates the calls to Φ_{merces} on the ledger: the INITDEPOSIT, INITWITHDRAW and INITTRANSFER calls of U' for qid become $(\textsc{Deposit}, U', \mathsf{qid}, v)$, $(\textsc{Withdraw}, U', \mathsf{qid}, v)$ and $(\textsc{ConfTransfer}, U', \mathsf{qid}, R)$, respectively, where v is the deposited or withdrawn amount and R the receiver, and the closing call for (U', qid) becomes $(\textsc{Executed}, U', \mathsf{qid})$.

4.2 The Smart Contract

We describe the smart contract Φ_{merces} that U_{merces} deploys on $\mathcal{F}_{\mathsf{SmartContr}}$. Its public state consists of a commitment to each user's private balance and a queue of requests. Each request goes through two calls: the user registers it with an *initial call* (INITDEPOSIT, INITWITHDRAW or INITTRANSFER), and, once the MPC parties have processed it off-chain, U_{merces} completes it with the matching *closing call* (ENDDEPOSIT, ENDWITHDRAW or ENDTRANSFER), which carries the new commitments and a proof of their correctness. The contract verifies the proof, updates the commitments, and marks the request as done, so that its query ID cannot be reused. The code of Φ_{merces} has the bound B and the CRSs of the four relations of Section 4.4 hardcoded.

Contract: Φ_{merces}

State. Two dictionaries: $\mathsf{st.comm}$, mapping each user U to a commitment, and $\mathsf{st.queue}$, mapping each pair (U, qid) to a pending request or to DONE. Unset entries default to $\mathtt{commit}(0; 0)$ and $\perp$, respectively.

Code. Recall from $\mathcal{F}_{\mathsf{SmartContr}}$ that, when a user U issues $(\textsc{Execute}, \mathsf{cid}, \tau, v)$, the contract is invoked as $\Phi(\mathsf{st}, \tau, U, v)$. If any of the checks below fails, or the input is not of one of the forms below, Φ_{merces} outputs $b = \mathsf{reject}$.

On input $(\mathsf{st}, (\textsc{InitDeposit}, \mathsf{qid}), U, v)$:

1. Check that $\mathsf{st.queue}[(U, \mathsf{qid})] = \bot$ and $v < B$.
2. Set $\mathsf{st.queue}[(U, \mathsf{qid})] \leftarrow (\textsc{PendingDeposit}, v)$, and output $b = (\mathsf{accept}, \mathsf{st}, \bot, 0)$.
   The v tokens attached to the call stay in the contract.

On input $(\mathsf{st}, (\textsc{EndDeposit}, U, \mathsf{qid}, \pi, \gamma_{\mathsf{new}}), U_{\mathsf{merces}}, 0)$:

1. Check that $\mathsf{st.queue}[(U, \mathsf{qid})] = (\textsc{PendingDeposit}, v)$ for some v, and that $\mathtt{NIZK}_{\mathcal{R}_{\mathsf{deposit}}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}_{\mathsf{deposit}}}, x, \pi) = \mathsf{accept}$ for $x = (\mathsf{st.comm}[U], \gamma_{\mathsf{new}}, v)$.
2. Set $\mathsf{st.queue}[(U, \mathsf{qid})] \leftarrow \textsc{Done}$ and $\mathsf{st.comm}[U] \leftarrow \gamma_{\mathsf{new}}$, and output $b = (\mathsf{accept}, \mathsf{st}, \bot, 0)$.

On input $(\mathsf{st}, (\textsc{InitWithdraw}, \mathsf{qid}, v), U, 0)$:

1. Check that $\mathsf{st.queue}[(U, \mathsf{qid})] = \bot$ and $v < B$.
2. Set $\mathsf{st.queue}[(U, \mathsf{qid})] \leftarrow (\textsc{PendingWithdraw}, v)$, and output $b = (\mathsf{accept}, \mathsf{st}, \bot, 0)$.

On input $(\mathsf{st}, (\textsc{EndWithdraw}, U, \mathsf{qid}, \pi, \gamma_{\mathsf{new}}), U_{\mathsf{merces}}, 0)$:

1. Check that $\mathsf{st.queue}[(U, \mathsf{qid})] = (\textsc{PendingWithdraw}, v)$ for some v, and that $\mathtt{NIZK}_{\mathcal{R}_{\mathsf{withdraw}}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}_{\mathsf{withdraw}}}, x, \pi) = \mathsf{accept}$ for $x = (\mathsf{st.comm}[U], \gamma_{\mathsf{new}}, v)$.
2. Set $\mathsf{st.queue}[(U, \mathsf{qid})] \leftarrow \textsc{Done}$ and $\mathsf{st.comm}[U] \leftarrow \gamma_{\mathsf{new}}$, and output $b = (\mathsf{accept}, \mathsf{st}, U, v)$.
   This transfers v tokens from the contract to U.

On input $(\mathsf{st}, (\textsc{InitTransfer}, \mathsf{qid}, R, \pi_{\mathsf{user}}, \gamma_{\mathsf{transfer}}), U, 0)$:

1. Check that $\mathsf{st.queue}[(U, \mathsf{qid})] = \bot$, $R \in \mathcal{U} \backslash \{U\}$, and $\mathtt{NIZK}_{\mathcal{R}_{\mathsf{user}}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}_{\mathsf{user}}}, x, \pi_{\mathsf{user}}) = \mathsf{accept}$ for $x = (\gamma_{\mathsf{transfer}}, U)$.
   The statement contains the caller U, so no other user can reuse π_{user}.
2. Set $\mathsf{st.queue}[(U, \mathsf{qid})] \leftarrow (\textsc{PendingTransfer}, R, \gamma_{\mathsf{transfer}})$, and output $b = (\mathsf{accept}, \mathsf{st}, \bot, 0)$.

On input $(\mathsf{st}, (\textsc{EndTransfer}, U, \mathsf{qid}, \pi, \gamma^U_{\mathsf{new}}, \gamma^R_{\mathsf{new}}), U_{\mathsf{merces}}, 0)$:

1. Check that $\mathsf{st.queue}[(U, \mathsf{qid})] = (\textsc{PendingTransfer}, R, \gamma_{\mathsf{transfer}})$ for some R and $\gamma_{\mathsf{transfer}}$, and that $\mathtt{NIZK}_{\mathcal{R}_{\mathsf{transfer}}}.\mathtt{verify}(\mathsf{crs}_{\mathcal{R}_{\mathsf{transfer}}}, x, \pi) = \mathsf{accept}$ for $x = (\mathsf{st.comm}[U], \mathsf{st.comm}[R], \gamma^U_{\mathsf{new}}, \gamma^R_{\mathsf{new}}, \gamma_{\mathsf{transfer}})$.
2. Set $\mathsf{st.queue}[(U, \mathsf{qid})] \leftarrow \textsc{Done}$, $\mathsf{st.comm}[U] \leftarrow \gamma^U_{\mathsf{new}}$ and $\mathsf{st.comm}[R] \leftarrow \gamma^R_{\mathsf{new}}$, and output $b = (\mathsf{accept}, \mathsf{st}, \bot, 0)$.

4.3 MPC Procedures

Π_{CToken} delegates each request to one of three procedures, π_{Deposit}, π_{Withdraw} and $\pi_{\mathsf{ConfTransfer}}$, and balance queries to a fourth one, π_{Balance}. In the first three, the user notifies the MPC parties $\mathcal{P}$ of its request. The MPC parties keep the requests they are notified about in a first-in-first-out list, and process them one at a time: they fetch the request from Φ_{merces}, compute in MPC the new balances, their commitments and a proof, and close the request on-chain. We say that they are *processing* a request from the moment they fetch it until they close or drop it. Whenever a check fails or the prover outputs $\bot$, they *drop* the request: they stop processing it without changing conf-bal or com-rand, and the request remains pending on-chain. Since we assume honest users do not overdraw (Section 3), this only happens to requests of corrupt users, unless some MPC party is actively corrupt.

Closing a request. To close a request with a closing call τ and new values for some entries of conf-bal and com-rand, the MPC parties send τ to U_{merces}, who issues (EXECUTE, $\mathsf{cid}_{\mathsf{merces}}, \tau, 0$) to $\mathcal{F}_{\mathsf{SmartContr}}$ and notifies them. The MPC parties then call READ on $\mathcal{F}_{\mathsf{SmartContr}}$, and update the entries to the new values if Φ_{merces} marks the request as DONE; otherwise, they drop the request. Updating the shares only once the closing call has been accepted keeps them consistent with the on-chain commitments, and processing one request at a time ensures that the commitments a proof refers to are still on-chain when its closing call is posted.

4.3.1 MPC Procedures for Deposit and Withdrawal

In π_{Deposit}, the user does not send the amount v: the MPC parties read it from the request on-chain, where U attached the v tokens to its initial call. They add v to U's balance on shares, commit to the result with fresh randomness, and prove the update correct with respect to relation $\mathcal{R}_{\mathsf{deposit}}$. π_{Withdraw} is symmetric, subtracting v instead; relation $\mathcal{R}_{\mathsf{withdraw}}$ additionally enforces solvency ($v \leq b_{\mathsf{old}}$), so the prover outputs $\perp$ if U lacks the funds.

Procedure 2: π_{Deposit}

Takes as input qid from a user U.

1. U sends qid to the MPC parties $\mathcal{P}$, who add (U, qid) to their list of requests.
2. When processing (U, qid), the MPC parties:
   (a) Call READ on $\mathcal{F}_{\mathsf{SmartContr}}$ to fetch the state st of Φ_{merces}, and check that $\mathsf{st.queue}[(U, \mathsf{qid})] = (\text{PENDINGDEPOSIT}, v)$ for some v.
   (b) Use $\mathcal{F}_{\mathsf{MPC}}$ to sample $[\![r_{\mathsf{new}}]\!]$, set $[\![b_{\mathsf{new}}]\!] \leftarrow \mathsf{conf\text{-}bal}[U] + v$, and compute and open $\gamma_{\mathsf{new}} = \mathtt{commit}([\![b_{\mathsf{new}}]\!]; [\![r_{\mathsf{new}}]\!])$ and $\pi = \mathtt{NIZK}_{\mathcal{R}_{\mathsf{deposit}}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}_{\mathsf{deposit}}}, x, [\![w]\!])$, where $x = (\mathsf{st.comm}[U], \gamma_{\mathsf{new}}, v)$ and $[\![w]\!] = (\mathsf{conf\text{-}bal}[U], \mathsf{com\text{-}rand}[U], [\![r_{\mathsf{new}}]\!])$.
   (c) Close the request with $\tau = (\text{ENDDEPOSIT}, U, \mathsf{qid}, \pi, \gamma_{\mathsf{new}})$ and the new values $\mathsf{conf\text{-}bal}[U] \leftarrow [\![b_{\mathsf{new}}]\!]$ and $\mathsf{com\text{-}rand}[U] \leftarrow [\![r_{\mathsf{new}}]\!]$.

Procedure 3: π_{Withdraw}

Takes as input qid from a user U.

1. U sends qid to the MPC parties $\mathcal{P}$, who add (U, qid) to their list of requests.
2. When processing (U, qid), the MPC parties:
   (a) Call READ on $\mathcal{F}_{\mathsf{SmartContr}}$ to fetch the state st of Φ_{merces}, and check that $\mathsf{st.queue}[(U, \mathsf{qid})] = (\text{PENDINGWITHDRAW}, v)$ for some v.
   (b) Use $\mathcal{F}_{\mathsf{MPC}}$ to sample $[\![r_{\mathsf{new}}]\!]$, set $[\![b_{\mathsf{new}}]\!] \leftarrow \mathsf{conf\text{-}bal}[U] - v$, and compute and open $\gamma_{\mathsf{new}} = \mathtt{commit}([\![b_{\mathsf{new}}]\!]; [\![r_{\mathsf{new}}]\!])$ and $\pi = \mathtt{NIZK}_{\mathcal{R}_{\mathsf{withdraw}}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}_{\mathsf{withdraw}}}, x, [\![w]\!])$, where $x = (\mathsf{st.comm}[U], \gamma_{\mathsf{new}}, v)$ and $[\![w]\!] = (\mathsf{conf\text{-}bal}[U], \mathsf{com\text{-}rand}[U], [\![r_{\mathsf{new}}]\!])$.
   (c) Close the request with $\tau = (\text{ENDWITHDRAW}, U, \mathsf{qid}, \pi, \gamma_{\mathsf{new}})$ and the new values $\mathsf{conf\text{-}bal}[U] \leftarrow [\![b_{\mathsf{new}}]\!]$ and $\mathsf{com\text{-}rand}[U] \leftarrow [\![r_{\mathsf{new}}]\!]$.

4.3.2 MPC Procedure for Confidential Transfers

In $\pi_{\mathsf{ConfTransfer}}$, the amount is private: the user inputs v and the randomness r_{transfer} of its on-chain commitment $\gamma_{\mathsf{transfer}}$ to $\mathcal{F}_{\mathsf{MPC}}$. The MPC parties first recompute this commitment in MPC and compare it with the one on-chain, which ensures that the user input the amount it committed to in its initial call. They then debit v from the sender's balance and credit it to the receiver's, commit to both new balances, and prove the update correct

with respect to relation $\mathcal{R}_{\mathsf{transfer}}$, which also enforces the sender's solvency.

Procedure 4: $\pi_{\mathsf{ConfTransfer}}$

Takes as input $(\mathsf{qid}, v, r_{\mathsf{transfer}})$ from a user U.

1. U sends qid to the MPC parties $\mathcal{P}$, and $(\textsc{PrivInput}, v)$ and $(\textsc{PrivInput}, r_{\mathsf{transfer}})$ to $\mathcal{F}_{\mathsf{MPC}}$. Once they have received qid and the handles $[\![v]\!]$ and $[\![r_{\mathsf{transfer}}]\!]$, the MPC parties add (U, qid) to their list of requests.
2. When processing (U, qid), the MPC parties:
   (a) Call $\textsc{Read}$ on $\mathcal{F}_{\mathsf{SmartContr}}$ to fetch the state st of Φ_{merces}, and check that $\mathsf{st.queue}[(U, \mathsf{qid})] = (\textsc{PendingTransfer}, R, \gamma_{\mathsf{transfer}})$ for some R and $\gamma_{\mathsf{transfer}}$.
   (b) Use $\mathcal{F}_{\mathsf{MPC}}$ to compute and open $\mathtt{commit}([\![v]\!]\,;[\![r_{\mathsf{transfer}}]\!])$, and check that it equals $\gamma_{\mathsf{transfer}}$.
   (c) Use $\mathcal{F}_{\mathsf{MPC}}$ to sample $[\![r^U_{\mathsf{new}}]\!]$ and $[\![r^R_{\mathsf{new}}]\!]$, set $[\![b^U_{\mathsf{new}}]\!] \leftarrow \mathsf{conf\text{-}bal}[U] - [\![v]\!]$ and $[\![b^R_{\mathsf{new}}]\!] \leftarrow \mathsf{conf\text{-}bal}[R] + [\![v]\!]$, and compute and open $\gamma^U_{\mathsf{new}} = \mathtt{commit}([\![b^U_{\mathsf{new}}]\!]\,;[\![r^U_{\mathsf{new}}]\!])$, $\gamma^R_{\mathsf{new}} = \mathtt{commit}([\![b^R_{\mathsf{new}}]\!]\,;[\![r^R_{\mathsf{new}}]\!])$ and $\pi = \mathtt{NIZK}_{\mathcal{R}_{\mathsf{transfer}}}.\mathtt{prove}(\mathsf{crs}_{\mathcal{R}_{\mathsf{transfer}}}, x, [\![w]\!])$, where $x = (\mathsf{st.comm}[U], \mathsf{st.comm}[R], \gamma^U_{\mathsf{new}}, \gamma^R_{\mathsf{new}}, \gamma_{\mathsf{transfer}})$ and $[\![w]\!]$ consists of the old and new balances and randomness of U and R, together with $[\![v]\!]$ and $[\![r_{\mathsf{transfer}}]\!]$ (Section 4.4).
   (d) Close the request with $\tau = (\textsc{EndTransfer}, U, \mathsf{qid}, \pi, \gamma^U_{\mathsf{new}}, \gamma^R_{\mathsf{new}})$ and the new values $\mathsf{conf\text{-}bal}[U] \leftarrow [\![b^U_{\mathsf{new}}]\!]$, $\mathsf{conf\text{-}bal}[R] \leftarrow [\![b^R_{\mathsf{new}}]\!]$, $\mathsf{com\text{-}rand}[U] \leftarrow [\![r^U_{\mathsf{new}}]\!]$ and $\mathsf{com\text{-}rand}[R] \leftarrow [\![r^R_{\mathsf{new}}]\!]$.

4.3.3 MPC Procedure for Balance Queries

In π_{Balance}, the MPC parties open the user's secret-shared balance and blinding randomness towards it, and the user checks them against its on-chain commitment. The MPC parties answer only while they are not processing a request, so that the opened values match the on-chain commitment; otherwise, U could output $\perp$ even if no one misbehaved.

Procedure 5: π_{Balance}

Executed on behalf of a user U.

1. U sends $(\textsc{Balance})$ to the MPC parties $\mathcal{P}$.
2. As soon as they are not processing a request, the MPC parties $\mathcal{P}$ open $\mathsf{conf\text{-}bal}[U]$ and $\mathsf{com\text{-}rand}[U]$ towards U through $\mathcal{F}_{\mathsf{MPC}}$.
3. Upon receiving (b, r), U calls $\textsc{Read}$ on $\mathcal{F}_{\mathsf{SmartContr}}$ to fetch the state st of Φ_{merces}, and outputs b if $\mathtt{commit}(b; r) = \mathsf{st.comm}[U]$, and $\perp$ otherwise.
   A corrupt network can refuse to answer, but cannot report a wrong balance.

4.4 The ZK Statements

The protocol uses four NIZK relations, three produced by the MPC parties (one per operation: $\mathcal{R}_{\mathsf{deposit}}$, $\mathcal{R}_{\mathsf{withdraw}}$, $\mathcal{R}_{\mathsf{transfer}}$) and one produced by the user at $\textsc{InitTransfer}$ time ($\mathcal{R}_{\mathsf{user}}$, a range proof on the committed transfer amount, bound to the sender). As in Section 2.3, amounts and balances are read as integers, so comparisons such as $v \leq b_{\mathsf{old}}$ are between integers.

Relation $\mathcal{R}_{\mathsf{deposit}}$. The new commitment results from adding the public deposit amount v to the value committed in the old commitment, refreshed with new randomness.

$$\mathcal{R}_{\mathsf{deposit}} = \left\{ \begin{pmatrix} x = (\gamma_{\mathsf{old}}, \gamma_{\mathsf{new}}, v), \\ w = (b_{\mathsf{old}}, r_{\mathsf{old}}, r_{\mathsf{new}}) \end{pmatrix} : \begin{array}{l} \gamma_{\mathsf{old}} = \mathtt{commit}(b_{\mathsf{old}}; r_{\mathsf{old}}) \\ \gamma_{\mathsf{new}} = \mathtt{commit}(b_{\mathsf{old}} + v; r_{\mathsf{new}}) \end{array} \right\}$$

Relation $\mathcal{R}_{\mathsf{withdraw}}$. Symmetric to $\mathcal{R}_{\mathsf{deposit}}$ but with subtraction, plus a solvency check $v \leq b_{\mathsf{old}}$: if U lacks the funds, the prover outputs $\perp$ and the MPC parties drop the request (Section 4.3).

$$\mathcal{R}_{\mathsf{withdraw}} = \left\{ \begin{pmatrix} x = (\gamma_{\mathsf{old}}, \gamma_{\mathsf{new}}, v), \\ w = (b_{\mathsf{old}}, r_{\mathsf{old}}, r_{\mathsf{new}}) \end{pmatrix} : \begin{array}{l} \gamma_{\mathsf{old}} = \mathtt{commit}(b_{\mathsf{old}}; r_{\mathsf{old}}) \\ \gamma_{\mathsf{new}} = \mathtt{commit}(b_{\mathsf{old}} - v; r_{\mathsf{new}}) \\ v \leq b_{\mathsf{old}} \end{array} \right\}$$

Relation $\mathcal{R}_{\mathsf{transfer}}$. The new commitments correctly debit the sender and credit the receiver by the secret amount v committed in $\gamma_{\mathsf{transfer}}$, with sender solvency.

$$\mathcal{R}_{\mathsf{transfer}} = \left\{ \begin{pmatrix} x = (\gamma^U_{\mathsf{old}}, \gamma^R_{\mathsf{old}}, \gamma^U_{\mathsf{new}}, \gamma^R_{\mathsf{new}}, \gamma_{\mathsf{transfer}}), \\ w = \begin{pmatrix} b = (b^U_{\mathsf{old}}, b^R_{\mathsf{old}}, b^U_{\mathsf{new}}, b^R_{\mathsf{new}}, v), \\ r = (r^U_{\mathsf{old}}, r^R_{\mathsf{old}}, r^U_{\mathsf{new}}, r^R_{\mathsf{new}}, r_{\mathsf{transfer}}) \end{pmatrix} \end{pmatrix} : \begin{array}{l} \{x[i] = \mathtt{commit}(b[i]; r[i])\}_{i=1}^{5} \\ v \leq b^U_{\mathsf{old}} \\ b^U_{\mathsf{new}} = b^U_{\mathsf{old}} - v \\ b^R_{\mathsf{new}} = b^R_{\mathsf{old}} + v \end{array} \right\}$$

Relation $\mathcal{R}_{\mathsf{user}}$. Produced by the user (not by MPC) and verified on-chain by Φ_{merces}. It attests that the user's amount commitment $\gamma_{\mathsf{transfer}}$ opens to some $v \in [0, B)$ (Section 2.3); the role this plays in the security argument is discussed in Section 5. The statement also contains the sender U, which the relation does not constrain: a proof for (γ, U') with $U' \neq U$ is a proof for a different statement, so by simulation-extractability only someone who knows an opening of γ can produce it. Since Φ_{merces} sets U to the caller of InitTransfer, no user can reuse the proof of another.[15]

$$\mathcal{R}_{\mathsf{user}} = \left\{ \big(x = (\gamma, U),\ w = (v, r)\big) \ : \begin{array}{l} 0 \leq v < B, \\ \gamma = \mathtt{commit}(v; r) \end{array} \right\}$$

5 Security Proof

As is common in UC, we prove security against a class of *environments*.[16] We consider environments that corrupt any users and *any* number of MPC parties, passively or actively. This is uncommon in MPC, where the adversary is normally bounded to at most t corruptions, beyond which protocols offer no guarantees. In our case, even if *all* MPC parties are actively corrupt, a meaningful functionality remains: the adversary learns all balances and amounts, and decides which requests are executed and when, but tokens still move only as their owners requested. Corruptions are static (Section 2.1); Section 5.1 explains how the proof extends to adaptive corruptions.

[15] In Groth16, a public input that appears in no constraint does not affect verification at all, so the circuit must include U in a dummy constraint, e.g., $s = U \cdot U$ for an auxiliary variable s.

[16] For instance, honest-majority protocols are secure against environments that corrupt a minority of the parties.

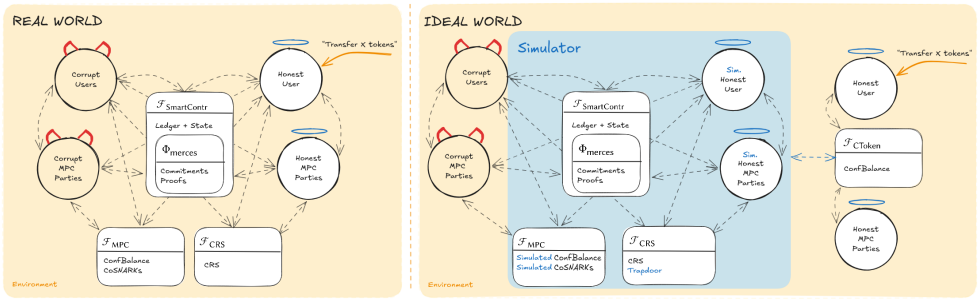


Figure 2: Overview of the UC model for MERCES. On the left, the *real world*, where the users and the MPC parties execute the protocol, using functionalities $\mathcal{F}_{\mathsf{MPC}}$, $\mathcal{F}_{\mathsf{SmartContr}}$ and $\mathcal{F}_{\mathsf{CRS}}$ as building blocks. On the right, the *ideal world*, where the corrupt users and the corrupt MPC parties still "think" they are running the protocol, but it is instead emulated by a *simulator* $\mathcal{S}$, which interacts with the ideal functionality $\mathcal{F}_{\mathsf{CToken}}$. Both worlds should be indistinguishable to the *environment* $\mathcal{Z}$, which controls the corrupt parties and sees the inputs and outputs of the honest parties.

Theorem 1. *Assume that the NIZK arguments are complete, zero-knowledge and weakly simulation-extractable (Section 2.4), and that the commitment scheme is equivocable and extractable (Section 2.5). Consider PPT environments that statically corrupt any users and any number of MPC parties, passively or actively, subject to the following restrictions:*

- U_{merces} *deploys the correct contract* Φ_{merces}*, and is corrupt only if some MPC party is actively corrupt (Section 2.1);*
- *honest users never request to move more private tokens than they have (Section 3);*
- *the total supply of the token is below* $p/2$ *(Section 2.3).*

Then Π_{CToken} *UC-realizes* $\mathcal{F}_{\mathsf{CToken}}$ *in the* $(\mathcal{F}_{\mathsf{SmartContr}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{CRS}})$*-hybrid model against such environments.*

Proof. We describe a simulator $\mathcal{S}$, and then show that no environment can distinguish the real execution from the ideal one with $\mathcal{S}$. As depicted in Figure 2, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{SmartContr}}$, $\mathcal{F}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{CRS}}$, as well as the honest users and MPC parties, towards the adversary, and interacts with $\mathcal{F}_{\mathsf{CToken}}$ on behalf of the corrupt users. Since corruptions are static, there are two cases, which $\mathcal{S}$ can tell apart from the start. Either *privacy holds*, i.e., at most t MPC parties are corrupt, all of them passively, in which case U_{merces} is honest by our restriction; or *privacy is lost*, i.e., the flag leaked of $\mathcal{F}_{\mathsf{CToken}}$ is 1.

The simulator. $\mathcal{S}$ proceeds as follows. Steps (1)–(4) are common to both cases, step (5) covers the case where privacy is lost, and steps (6)–(8) the case where privacy holds.

(1) *Setup.* $\mathcal{S}$ samples trapdoored CRSs $(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}) \leftarrow S_{1,\mathcal{R}}(1^{\kappa})$ for the four relations of Section 4.4, and trapdoored commitment parameters $(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{TdSetup}(1^{\kappa})$, and uses them in its emulation of $\mathcal{F}_{\mathsf{CRS}}$. The contract Φ_{merces} is deployed on the emulated $\mathcal{F}_{\mathsf{SmartContr}}$ as in the protocol.

(2) *Public state.* Apart from Φ_{merces}, $\mathcal{S}$ keeps its emulated $\mathcal{F}_{\mathsf{SmartContr}}$ in sync with $\mathcal{F}_{\mathsf{CToken}}$: it forwards the public commands of corrupt users to $\mathcal{F}_{\mathsf{CToken}}$, and reads the state of $\mathcal{F}_{\mathsf{CToken}}$ to learn about those of honest users. The contract Φ_{merces}, its balance and the calls to it only exist in $\mathcal{S}$'s emulation. $\mathcal{S}$ keeps them consistent with $\mathcal{F}_{\mathsf{CToken}}$ as described next, so that translating the emulated chain as honest users do (Remark 1) yields the state of $\mathcal{F}_{\mathsf{CToken}}$.

(3) *Requests of corrupt users.* When the emulated contract accepts an initial call of a corrupt user U, $\mathcal{S}$ submits the corresponding request to $\mathcal{F}_{\mathsf{CToken}}$ on behalf of U. For a transfer, it submits the amount $v = \mathsf{Ext}(\mathsf{td}, \gamma_{\mathsf{transfer}})$.[17]

(4) *Closing calls.* Whenever the emulated contract accepts a closing call for a request (U, qid), $\mathcal{S}$ sends $(\textsc{Proceed}, U, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{CToken}}$.

(5) *If privacy is lost.* $\mathcal{F}_{\mathsf{CToken}}$ sends $\mathcal{S}$ the amount of every transfer, so $\mathcal{S}$ knows all amounts and balances. It therefore runs the code of the honest parties on the actual values, exactly as in the real execution; its emulated $\mathcal{F}_{\mathsf{MPC}}$ sends every stored value to the adversary and lets the adversary choose every output, as $\mathcal{F}_{\mathsf{MPC}}$ does when privacy is lost. For a balance query of an honest user U, $\mathcal{S}$ sends $(\textsc{Respond}, U)$ to $\mathcal{F}_{\mathsf{CToken}}$ if the values that U receives open $\mathsf{st.comm}[U]$, and $(\textsc{Respond}, U, \bot)$ otherwise.

(6) *If privacy holds: requests of honest users.* When $\mathcal{F}_{\mathsf{CToken}}$ reports a new request of an honest user U, $\mathcal{S}$ emulates the initial call of U on its emulated $\mathcal{F}_{\mathsf{SmartContr}}$, and the messages of U to the MPC parties. $\mathcal{S}$ does not know the amount of a transfer: it posts an equivocal commitment $\gamma_{\mathsf{transfer}} \leftarrow \mathsf{EqCom}(\mathsf{td})$ and a simulated proof $\pi_{\mathsf{user}} \leftarrow S_{2,\mathcal{R}_{\mathsf{user}}}(\mathsf{crs}_{\mathcal{R}_{\mathsf{user}}}, \mathsf{td}_{\mathcal{R}_{\mathsf{user}}}, (\gamma_{\mathsf{transfer}}, U))$, and stores dummy values for v and r_{transfer} in its emulated $\mathcal{F}_{\mathsf{MPC}}$. Here and below, we leave implicit the auxiliary output of EqCom, which $\mathcal{S}$ keeps for EqOpen.

(7) *If privacy holds: processing.* $\mathcal{S}$ runs the code of the honest MPC parties and of U_{merces}, with dummy values in place of the secrets; the passively corrupt MPC parties only see handles and opened values. It decides whether a request is dropped as the MPC parties would (Section 4.3):

- the check of the queue entry only involves public values;
- for a transfer, the check of the amount commitment passes if the sender is honest; for a corrupt sender U, $\mathcal{S}$ knows the inputs (v', r') that U gave to $\mathcal{F}_{\mathsf{MPC}}$, and checks whether $\mathtt{commit}(v'; r') = \gamma_{\mathsf{transfer}}$, in which case v' is the amount that $\mathcal{S}$ submitted to $\mathcal{F}_{\mathsf{CToken}}$, by extractability;
- the solvency check passes if U is honest, since honest users do not overdraw; for a corrupt U, $\mathcal{S}$ obtains $\mathsf{priv\text{-}bal}[U]$ from $\mathcal{F}_{\mathsf{CToken}}$, through a $\textsc{Balance}$ query on behalf of U, and compares it with the amount of the request.

If the request is not dropped, $\mathcal{S}$ lets its emulated $\mathcal{F}_{\mathsf{MPC}}$ output equivocal commitments $\gamma_{\mathsf{new}} \leftarrow \mathsf{EqCom}(\mathsf{td})$ as the new commitments, and a simulated proof $\pi \leftarrow S_{2,\mathcal{R}}(\mathsf{crs}_{\mathcal{R}}, \mathsf{td}_{\mathcal{R}}, x)$ for the statement x of the procedure.

(8) *If privacy holds: balance queries.* When $\mathcal{F}_{\mathsf{CToken}}$ reports $(\textsc{Balance}, U)$ for an honest user U, $\mathcal{S}$ emulates the message of U to the MPC parties, and sends $(\textsc{Respond}, U)$ to $\mathcal{F}_{\mathsf{CToken}}$ once the MPC parties open the balance of U towards it. For a corrupt user U, $\mathcal{S}$ obtains $b = \mathsf{priv\text{-}bal}[U]$ from $\mathcal{F}_{\mathsf{CToken}}$, and opens the current commitment $\mathsf{st.comm}[U]$ to b with EqOpen. This is the only place where $\mathcal{S}$ opens an equivocal commitment.

Consistency of the emulated chain. For a commitment c on the emulated chain, let $\mathsf{val}(c)$ denote its content: if $\mathcal{S}$ produced c, the amount or balance in $\mathcal{F}_{\mathsf{CToken}}$ that c stands for; otherwise, $\mathsf{Ext}(\mathsf{td}, c)$. We say that a statement is *valid* if the contents of its commitments, together with its public values, satisfy the constraints that its relation places on amounts and balances (Section 4.4); e.g., (γ, U) is valid for $\mathcal{R}_{\mathsf{user}}$ if $0 \leq \mathsf{val}(\gamma) < B$. The adversary chooses the proofs in the initial calls of corrupt users and, if privacy is lost, those in closing calls. Consider the following two conditions:

[17] If $\mathcal{S}$ produced $\gamma_{\mathsf{transfer}}$ itself, it submits the amount that $\gamma_{\mathsf{transfer}}$ stands for. $\mathcal{S}$ knows this amount unless $\gamma_{\mathsf{transfer}}$ is an equivocal commitment that it has not opened, which condition (ii) below rules out.

(i) every statement for which the emulated contract accepts a proof chosen by the adversary is valid;

(ii) the emulated contract accepts no initial call of a corrupt user that contains an equivocal commitment that $\mathcal{S}$ has not opened.

We claim that, as long as (i) and (ii) hold, so do the following:

(a) $\mathcal{F}_{\mathsf{CToken}}$ registers exactly the requests whose initial calls the emulated contract accepts, and executes exactly those whose closing calls it accepts;

(b) $\mathsf{val}(\mathsf{st.comm}[U]) = \mathsf{priv\text{-}bal}[U]$ for every user U, and $\mathsf{val}(\gamma_{\mathsf{transfer}})$ is the amount of every pending transfer in $\mathcal{F}_{\mathsf{CToken}}$.

Both hold initially, and can only be affected when the emulated contract accepts a call.

- *Initial calls.* For honest users, (a) and (b) hold by construction. For corrupt users, the contract and $\mathcal{F}_{\mathsf{SmartContr}}$ check the same conditions as $\mathcal{F}_{\mathsf{CToken}}$ (an unused query ID, $v < B$, a receiver $R \neq U$ and, for deposits, a sufficient public balance), except that for transfers, the contract verifies π_{user} for the statement $(\gamma_{\mathsf{transfer}}, U)$ instead of checking $v < B$. By (i), $0 \leq \mathsf{val}(\gamma_{\mathsf{transfer}}) < B$, and by (ii), this is the amount that $\mathcal{S}$ submits.

- *Closing calls if privacy holds.* These are posted by the honest U_{merces}, for requests that $\mathcal{S}$ did not drop. The contract accepts them, since requests are processed one at a time, and $\mathcal{F}_{\mathsf{CToken}}$ executes them, since they are solvent. The new commitments are produced by $\mathcal{S}$ and stand for the balances after the execution, so (b) is maintained.

- *Closing calls if privacy is lost.* Their commitments and proofs are chosen by the adversary. By (i), their statements are valid. For a withdrawal or transfer, the amount v of the request, which for a transfer is $\mathsf{val}(\gamma_{\mathsf{transfer}})$ by (b), thus satisfies $v \leq \mathsf{val}(\mathsf{st.comm}[U]) = \mathsf{priv\text{-}bal}[U]$, so $\mathcal{F}_{\mathsf{CToken}}$ executes the request. Moreover, the new commitments contain the balances after the execution, which do not wrap around modulo p by the bound on the total supply, so (b) is maintained.

Finally, by (b) and extractability, whenever the values (b, r) that an honest user U receives in a balance query open $\mathsf{st.comm}[U]$, we have $b = \mathsf{priv\text{-}bal}[U]$, which is what $\mathcal{F}_{\mathsf{CToken}}$ outputs to U.

Indistinguishability. We go from the real execution to the ideal one through the following hybrids.

H_0: The real execution.

H_1: As H_0, but the commitment parameters are sampled with $\mathsf{TdSetup}$ and, if privacy holds, the commitments of honest users and the new commitments computed inside $\mathcal{F}_{\mathsf{MPC}}$ are computed with EqCom, and opened with EqOpen when their openings are needed, i.e., as witnesses of proofs and in balance queries. This is indistinguishable from H_0 by equivocability.

H_2: As H_1, but the CRSs are sampled with $S_{1,\mathcal{R}}$ and, if privacy holds, the proofs computed by honest users and inside $\mathcal{F}_{\mathsf{MPC}}$ are computed with $S_{2,\mathcal{R}}$. This is indistinguishable from H_1 by zero-knowledge, since all witnesses are known in H_1. Commitments now only need to be opened in the balance queries of corrupt users.

H_3: The ideal execution with $\mathcal{S}$. Compared to H_2, the outputs of honest users come from $\mathcal{F}_{\mathsf{CToken}}$ and, if privacy holds, $\mathcal{S}$ does not know the amounts and balances of honest users. By the claim, the outputs of honest users are the same as in H_2. Moreover, in H_2 the view of the adversary depends on these amounts and balances only through the decisions to drop requests and the answers to the balance queries of corrupt users. $\mathcal{S}$ computes both from what it learns from $\mathcal{F}_{\mathsf{CToken}}$, with the same result as in H_2 by the claim. Hence H_2 and H_3 are identical unless (i) or (ii) is violated, or the adversary opens a commitment c to a value other than $\mathsf{val}(c)$, which happens with negligible probability by extractability.

Conditions (i) and (ii). It remains to show that (i) and (ii) are violated with negligible probability in H_2, where both conditions make sense with the actual chain in place of the emulated one and the honest parties in place of $\mathcal{S}$. Suppose otherwise, and consider the first proof chosen by the adversary that the contract accepts in violation of (i) or (ii), for a statement x; a reduction can guess it among the polynomially many such proofs. No simulated proof was computed for x. Indeed, if privacy is lost, there are no simulated proofs at all. If privacy holds, the proof is a proof π_{user} in the initial call of a corrupt user U', and its statement (γ, U') differs from the statements (γ, U) of honest users U, for which the simulated proofs for $\mathcal{R}_{\mathsf{user}}$ were computed; this is where we use that π_{user} is bound to its sender. Weak simulation-extractability therefore yields a witness for x, which contains an opening of every commitment in x.[18] By extractability, none of these commitments is an equivocal commitment that has not been opened, and each commitment c is opened to $\mathsf{val}(c)$. Hence the call does not violate (ii), and x is valid, so it does not violate (i) either: a contradiction. Hence H_2 and H_3 are indistinguishable, and so are the real and ideal executions. □

5.1 Adaptive Corruptions

We sketch how Theorem 1 extends to adaptive corruptions without erasures. There, the environment may corrupt users and MPC parties at any point of the execution, a passively corrupt MPC party may later be corrupted actively, and upon corrupting a party the adversary learns its entire state, including the randomness it used so far. The protocol is unchanged; the model, the assumptions and the analysis change as follows.

Functionalities. $\mathcal{F}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{CToken}}$ accept corruptions at any time, and set $\mathsf{leaked} = 1$ as soon as more than t MPC parties are corrupt or one of them is actively corrupt. At that point, $\mathcal{F}_{\mathsf{MPC}}$ sends all the values it stores to the adversary, and $\mathcal{F}_{\mathsf{CToken}}$ sends all private balances and the amounts of all transfers registered so far: privacy is lost retroactively, since the MPC parties still hold shares of these values. When a user is corrupted, $\mathcal{F}_{\mathsf{CToken}}$ sends its inputs and outputs so far to the adversary. The restriction on U_{merces} becomes that it is corrupted only once some MPC party is actively corrupt. The resulting guarantee is meaningful for a long-running system: an MPC network that is compromised at some point reveals the past, but can neither forge transfers from then on nor undo earlier ones.

Simulator. While privacy holds, $\mathcal{S}$ proceeds as in steps (6)–(8) of the proof of Theorem 1, and once privacy is lost, as in step (5). In addition, whenever a corruption reveals values that $\mathcal{S}$ did not know, it must *explain* what it has shown the adversary so far. When a user U is corrupted while privacy holds, $\mathcal{S}$ learns the amounts of its transfers, opens their commitments accordingly with EqOpen, as well as the balance commitments that U

[18] The extractor must not obtain the trapdoor td of the commitment scheme: the adversary from which it extracts receives the commitment parameters, and the commitments and openings produced with td, from the outside.

checked in its balance queries, and produces prover randomness for the simulated proofs π_{user} of U. When privacy is lost, it does the same for all its commitments and for all proofs computed inside $\mathcal{F}_{\mathsf{MPC}}$, which stores their randomness.

Assumptions. Equivocability as defined in Section 2.5 already covers these openings. Zero-knowledge must be strengthened to *zero-knowledge with honest prover state reconstruction* [FLS90, GOS06]: there is a third simulator algorithm $S_{3,\mathcal{R}}$ which, given the state of $S_{2,\mathcal{R}}$ and a witness w for the statement, outputs prover randomness that explains the simulated proof. Formally, the two oracles of Section 2.4 must remain indistinguishable when $\mathcal{O}_P$ also returns the randomness it used, and $\mathcal{O}_S$ the output of $S_{3,\mathcal{R}}$. Groth16 [Gro16] has this property: its simulator knows the full trapdoor, and can therefore compute the prover randomness that explains a simulated proof once it is given the witness. Finally, the protocol that realizes $\mathcal{F}_{\mathsf{MPC}}$ must be secure against adaptive passive corruptions. This holds for information-theoretic protocols such as [BGW88] over secure channels [CFGN96], but is not immediate for protocols that derive their randomness from shared keys of a pseudorandom function, such as [AFL+16], since the simulator would have to explain these keys.

Analysis. The hybrids are as in the proof of Theorem 1, with the randomness of simulated proofs reconstructed by $S_{3,\mathcal{R}}$ when it leaks. However, weak simulation-extractability can no longer be applied to H_2 directly: the adversary may now learn the reconstructed randomness of simulated proofs, which the simulation oracle of Section 2.4 does not provide. Conditions (i) and (ii) are instead shown separately, using that an honest proof can be computed without the trapdoor, by anyone who knows the witness. For (i), we consider H_2 with all proofs of honest parties computed honestly again, which is indistinguishable from H_2 by zero-knowledge. There, no proof is simulated, so weak simulation-extractability applies, and together with extractability it shows as in that proof that (i) is violated with negligible probability. This carries over to H_2, since a violation can be detected efficiently given td, so it would otherwise distinguish the two executions. For (ii), consider the first violation in H_2: a corrupt user U' posts an equivocal commitment γ of an honest party that has not been opened, with an accepting proof for (γ, U'). A reduction guesses γ and computes all proofs π_{user} of honest users honestly again, except the one posted with γ, if any; by zero-knowledge, this changes the probability of the violation only negligibly. Now at most one proof for $\mathcal{R}_{\mathsf{user}}$ is simulated, for a statement (γ, U) with $U \neq U'$, and its randomness is not reconstructed before the violation: at that point, U is still honest and privacy still holds, as otherwise γ would have been opened. Hence weak simulation-extractability applies, and yields an opening of γ, which contradicts extractability.

5.1.1 UC-Secure NIZKs

Our analysis relies on the non-black-box extractor of the NIZK arguments, and thereby on knowledge assumptions (for Groth16, on the algebraic group model). This would be avoided if the NIZK arguments UC-realized the ideal NIZK functionality [GOS06], which in particular requires a straight-line extractor. The simulator would then obtain a witness for every proof of the adversary during the simulation, so that conditions (i) and (ii) would follow from the properties of the commitment scheme alone, also under adaptive corruptions, without the separate arguments of Section 5.1. However, Groth16 has no straight-line extractor: a proof from which a trapdoor reveals the witness cannot be shorter than the witness. Known transformations therefore add an encryption of the witness to the proof [KZM+15, BKSV21], which would increase the cost of proving in MPC and of verifying on-chain, whereas weak simulation-extractability lets us use Groth16 as it is.

Alternatively, succinct proofs can be composed in a variant of the UC framework that accounts for knowledge assumptions [KKK21a].

6 Implementation and Benchmarks

We discuss the concrete instantiation of MERCES, the corresponding design choices, and the performance of our end-to-end implementation. We further address practical-deployment aspects not captured by $\mathcal{F}_{\mathsf{CToken}}$ — gas costs, DoS countermeasures, user–MPC communication, compliance, and others.

6.1 Building Blocks Instantiations

- **MPC.** In our implementation we use semi-honest 3-party replicated secret sharing [MR18] as the MPC protocol and instantiate all commitments as hash-based commitments using the Poseidon2 [GKS23] hash function.
- **Commitments.** We use a hash-based commitment scheme $(m; r) \mapsto H(m\|r)$.[19] Poseidon2 is chosen for being both ZK- and reasonably MPC-friendly: it (and its predecessor Poseidon [GKR+21]) follow the Hades [GLR+20] design, originally proposed for MPC-friendly symmetric ciphers.
- **NIZK.** We use Groth16 [Gro16], still the standard proof system on most blockchains due to its small proof sizes and gas-efficient verifiers, and known to be highly MPC-friendly [OB22]. For evaluating Groth16 in MPC we use CoGroth16[20], which lets us write the proof circuits in Circom[21] and evaluate them efficiently under 3-party replicated secret sharing.

6.2 The Actual Smart Contract and Protocol

We discuss the changes from the formal Φ_{merces} of Section 4.2 to the contract we actually deploy; the source code accompanies this submission. Π_{CToken} and the procedures π_{Deposit}, π_{Withdraw}, $\pi_{\mathsf{ConfTransfer}}$ also adapt to practical considerations. Section B in the appendix gives the detailed workflow.

6.2.1 Workflow for ConfTransfer

A user U initiates a deposit, withdraw, or transfer by posting an "intent" in Φ_{merces}. For deposits and withdraws this includes the *public* amount; for a confidential transfer, Φ_{merces} requires a commitment $\gamma_{\mathsf{new}} = \mathtt{commit}(v; r_{\mathsf{transfer}})$ of the transfer amount together with a ZK proof that the underlying value is bounded, and the user must separately secret-share $[\![v]\!]$ and $[\![r_{\mathsf{transfer}}]\!]$ to the MPC servers.

A malicious U can send *inconsistent* shares (e.g., shares that are not replicated correctly). $\mathcal{F}_{\mathsf{MPC}}$ abstracts this away, and a concrete instantiation could run an input-check protocol; we opt for a different approach. Instead of sending shares directly to the servers, U posts, together with $\gamma_{\mathsf{transfer}}$, *encryptions* of the shares under the MPC servers' public keys, and the ZK statement is enhanced to prove that these encryptions contain valid shares consistent with the commitment. This not only solves the share-consistency issue, it also:

[19] It is natural to consider *homomorphic commitments* (*e.g.* Pedersen commitments) as the baseline of MERCES. These are more MPC-friendly, and, presumably, this could make the MPC protocol to compute the commitments easier. However, this is misleading. We also need the commitments to be *simultaneously* ZK-friendly, which is less trivial. We discuss this in detail in Section C in the Appendix.

[20] https://github.com/TaceoLabs/co-snarks

[21] https://docs.circom.io/

- Removes the step from $\pi_{\mathsf{ConfTransfer}}$ where the MPC servers need to recompute and open the commitment to cross-check with the on-chain commitment, and
- Removes the need of U to send messages directly to the MPC servers.

The second aspect is crucial for an actual deployment. Routing the user's input through the smart contract, rather than via direct connections to each server, has several practical benefits. Clients do not need to discover or maintain endpoints for every MPC node, and the servers' network locations need not be publicly exposed, reducing the attack surface for targeted DoS. The on-chain queue serves as a single, publicly auditable source of truth: a user cannot equivocate by sending different shares to different servers, and the MPC network cannot silently drop a request, since every input is recorded on-chain together with the wallet signature that authenticates it. This also makes the interaction asynchronous — users need not be online while the MPC network processes their request — and authentication piggybacks on the wallet signature already required to post the transaction, removing the need for a separate authentication layer between users and MPC servers.

6.2.2 Workflow for Deposit and Withdraw

This stays close to Φ_{merces}. For depositing, the user posts the intent and the call itself transfers the amount to the contract; the MPC servers fetch from the queue, execute π_{Deposit}, and post the new commitments and proofs. For withdrawing, the user posts the intent and the public amount; the public-token transfer happens only after π_{Withdraw} posts valid commitments and proofs on-chain.

Regarding query IDs, $\pi_{\mathsf{ConfTransfer}}$, π_{Deposit}, π_{Withdraw} have the user sample qid, tag the on-chain intent with it, and send qid to the MPC servers. In practice the MPC servers simply scan the contract state for new requests; we chose the qid formulation because the UC framework is event-driven and modeling "parties regularly inspect the state" is awkward.

Remark 2 (Fail-safe for DoS by MPC network). As discussed, the MPC network can neither misuse funds (due to ZK proofs) nor censor individual users (enforced by in-order processing). It can only infringe privacy by breaking non-collusion, or deny service by refusing to process intents. The first is inherent to MPC; the second we address here. Since each user's balance is represented on-chain as a commitment, the smart contract can be extended with a fail-safe: a user supplying its balance and blinding value (used to create the on-chain commitment) can withdraw funds directly. This path should only activate if the MPC network has refused to process the queue for some time frame, e.g., 1 day, to avoid inconsistencies between the secret-shared MPC state and the on-chain commitments when the MPC network is honest.

For the fail-safe to work, the user must track its balance and blinding value, reading them from the MPC network periodically. To prevent an attacker from transferring tokens to a user just before MPC goes offline (locking the recipient out, since they do not know the new randomness), we restrict redemption to the last commitment created *by the user itself* — i.e., a deposit, withdraw, or transfer where it was the sender — so the user can query the randomness from the MPC network immediately after the transaction is processed.

Remark 3 (On reading balances and compliance). To read balances, the MPC network reconstructs the user's balance and blinding shares to the user (e.g., as encryptions under the user's public key). The user then recomputes the balance commitment locally and compares against its on-chain counterpart — no ZK proof required. Due to this simplicity, read-intents need not be registered on-chain; the user can query the MPC network directly, authenticating with a wallet-signed request (with a timestamp or nullifier nonce for replay prevention) — a step otherwise implicit when posting an intent on-chain.

Compliance via regulator decryption requests works similarly. For transparency, regulators post their decryption intent on-chain, with their wallet whitelisted in the smart contract. The MPC network responds with encryptions of the requested balance and blinding values under the regulator's public key; since the regulator can directly compare the opened value to the on-chain commitment, no ZK proof is required.

6.2.3 Processing Transactions in Batches

A crucial optimization in MERCES is letting the MPC parties wait for several transactions in the queue before processing them. Once a tweakable batch threshold is reached, the MPC servers process them together and produce a *single proof* attesting to the validity of all transactions at once. Since Groth16 has constant proof size, the on-chain verification work remains essentially constant, regardless of the batch size. Computational costs on smart contracts translate to monetary costs (*gas*); batching thus yields very low gas costs, which is what makes MERCES suitable for real-world deployments. Section B.2 in the appendix details the batching, and Section 6.4 reports the resulting monetary costs.

6.3 MPC Benchmarks

We evaluate a proof-of-concept implementation of our MPC protocol with batched transaction processing.[22] Our benchmarks measure the runtime and throughput for batches of 50 transactions. Our benchmarks give one AWS instance to each of the 3 parties and we consider two AWS configurations: (i) *m7a.4xlarge* providing 16 CPU cores to each party and 12.5 Gbit/s network bandwidth between them, and (ii) *m7a.24xlarge* providing 96 CPU cores and 37.5 Gbit/s network bandwidth. We measure the multi-threaded runtime of the following protocol components:

- No Proofs: Non-verifiable MPC execution of the transaction batch without proof generation.
- WitExt: Witness extension phase.
- CoGroth16: Distributed Groth16 proof generation.
- Full: Total runtime including both witness extension and proof generation.

For each configuration, we report both the total latency per batch and the resulting transaction throughput. Furthermore, we give performance numbers with and without using the two-hash compression proposed in [KVW26] to save gas costs. The benchmarks for the runtime and the corresponding throughput of transactions per second are given in Table 2.

In summary, a batch of 50 transactions takes 163–357 ms (depending on hardware and two-hash compression) including the ZK proof; a non-verifiable variant takes $\sim 10\,ms$. This gives a throughput of 250–300 transactions per second with proof, and roughly 6000 without on the stronger machine.

The bulk of the gap between the two setups lies in the CoGroth16 prover: under 3-party replicated sharing, Groth16 sends only $\approx 1\,kB$ in 2 rounds, so performance is CPU-bound on the MSM (multi-scalar multiplication) evaluations, which parallelize well. Witness extension, in contrast, runs many parallel Poseidon2 commitments with > 60 communication rounds and low CPU utilization in between, so its small runtime gap comes mainly from the faster network.

[22] The source code can be found at https://github.com/TaceoLabs/merces1-paper.

Table 2: Runtime (RT) in *ms* and transactions per second (TPS) for all the steps involved in processing a batch of 50 transfers on the MPC network.

Machine	No Proofs		WitExt		CoGroth16		Full	
	RT	TPS	RT	TPS	RT	TPS	RT	TPS
No Compression								
m7a.4xlarge	11.3	4424.8	47.0	1063.8	238.7	209.5	285.7	175.0
m7a.24xlarge	8.0	6250.0	40.1	1246.9	123.0	406.5	163.1	306.6
Two-Hash Compression [KVW26]								
m7a.4xlarge	11.3	4424.8	50.1	998.0	307.4	162.7	357.5	139.9
m7a.24xlarge	8.0	6250.0	45.3	1103.8	149.3	334.9	194.6	256.9

6.4 Gas Costs and Number of R1CS Constraints

We report the cost of the ZK circuits (in R1CS constraints) and the Solidity gas costs. Table 3 gives the number of constraints for the client transfer proof and for the batched MPC proof (batch size 50), with and without the two-hash compression of [KVW26].

Table 3: Number of R1CS constraints of the involved proofs. Batch size for the MPC proof is 50.

	No Compression	Two-Hash Compression [KVW26]
Client Transfer	8 896	9 465
Batched MPC Proof	79 850	91 249

Table 4 reports gas for the four main Solidity functions (*deposit*, *withdraw*, *transfer* for the client, and *processMPC* for the MPC network) on the anvil[23] test environment, with *processMPC* processing a batch of 50 transfers. We give costs in Gwei at a gas price of 0.005 Gwei[24] (Base, an Ethereum L2) and in $ at an ETH price of 2250 $[25]. Ethereum mainnet gas is roughly 70× more expensive at time of writing.[26] Numbers are given with and without two-hash compression.

Table 4: Gas consumed for the on-chain operations and layer-2 gas cost assuming a price of 0.005 Gwei and 2250 $ per ETH. Process MPC processes a full batch of 50 transfers.

	No Compression			Two-Hash Compression [KVW26]		
	Gas	Gas cost [Gwei]	Gas cost [$]	Gas	Gas cost [Gwei]	Gas cost [$]
Deposit	96 994	484.97	0.001	96 994	484.97	0.001
Withdraw	74 546	372.73	0.001	74 546	372.73	0.001
Transfer	946 982	4 734.91	0.011	873 716	4 368.58	0.010
Process MPC	3 815 182	19 075.91	0.043	1 125 935	5 629.68	0.013

Client deposit/withdraw intents are cheap ($< 100k$ gas each); a transfer intent costs $870k$–$950k$ depending on compression, the extra cost stemming from storing the ciphertexts of the secret shares on-chain alongside the Groth16 proof verification. *Process MPC* is comparably expensive at 3.8M gas due to the Groth16 verifier's 300 public inputs; two-hash compression cuts this to 1.1M at the cost of 12k more R1CS constraints and ≈ 50 fewer

[23] https://www.getfoundry.sh/anvil
[24] From https://basescan.org/gastracker for Base mainnet at time of writing.
[25] Approximate price at time of writing; ETH price is highly volatile.
[26] See https://etherscan.io/gastracker; 0.333 Gwei at time of writing.

TPS on the MPC network, trading gas for throughput. In all cases, costs for registering a transfer intent and processing 50 transactions are in single-digit cents.

6.5 End-to-End System

We deployed MERCES as a fully integrated end-to-end demo on the Arc testnet[27] and the Base testnet[28]. The demo is meant to stress throughput: 200 wallets periodically post up to 500 transfer intents into the action queue, and the MPC nodes process them as in Section 6.3. The MPC nodes run in LAN, sharing an AMD EPYC 9454P 48-core CPU. These nodes handle requests for multiple MERCES deployments, so the throughput is lower than in Table 2.[29]

Figure 3 shows the dashboard of a running MERCES deployment with live network metrics. The *Transaction* section visualizes the current number of transaction requests processed by the MPC network per second. You can observe noticeable spikes whenever the predefined wallets submit bursts of transfer intents.

The *network utilization rate* shows the active transaction throughput relative to the maximum possible throughput, while the *Utilization MPC nodes* shows the relative CPU load, as well as the network's upload and download bandwidth for all three MPC nodes.

The *Average transaction latency* describes the average time between transaction issuance and the MPC network processing the request, publishing the new balance commitment on-chain, and verifying the ZK-proof. While pure MPC computation is fast (observe the *Average coSNARK generation time* metric), the latency experienced by a user is comparably high (e.g., 8.4 *s* as depicted in the screenshot). This discrepancy arises from the block time of the underlying blockchain. Indeed, the whole interaction requires two on-chain transactions, one for the client intent and one for the MPC network.

The *Total transaction* metric reports the cumulative number of confidential transfers successfully processed: roughly 5 million to date, demonstrating the system's stability under realistic operation.
The detailed processing steps and latencies of the actively running system from the point of view of the MPC network can be observed in the log view, as shown in Figure 5. Since our demo is deployed on a real blockchain (on testnet), the on-chain transactions can be viewed with the block explorer of the corresponding chain. Figure 4a shows a real transaction posted by our demo on the Arc testnet that calls the *processMPC* function of our MERCES contract. Figure 4b shows a screenshot of the public decoded input data of the transaction. This inputs include the number of actually processed user intents (4 in this example), the new sender and receiver commitment, the *valid* flag for each input, as well as the ZK proof attesting for the correct state-transition. The *beta* input is an artifact of the two-hash compression which is activated in this demo. Notice that the fourth processed transaction was marked to be invalid by the MPC network indicating that a sender did not have enough balance – a valid ZK proof was generated nonetheless. Regarding commitments, notice that the first one (the new sender commitment for the first processed transaction) is set to 0, indicating that a deposit was processed. The new receiver commitment, on the other hand, is a random looking 254 bit string, as expected. Since data posted to the blockchain is public, anyone can view this data. Due to our use of cryptographic commitments and ZK proofs, none of this data reveals any information about the underlying user balances and transaction amounts, making our token transfer system confidential.

[27] https://testnet.arcscan.app/
[28] https://sepolia.basescan.org/
[29] This setup was chosen for cost reasons, as the demo is intended to run for a longer period of time.

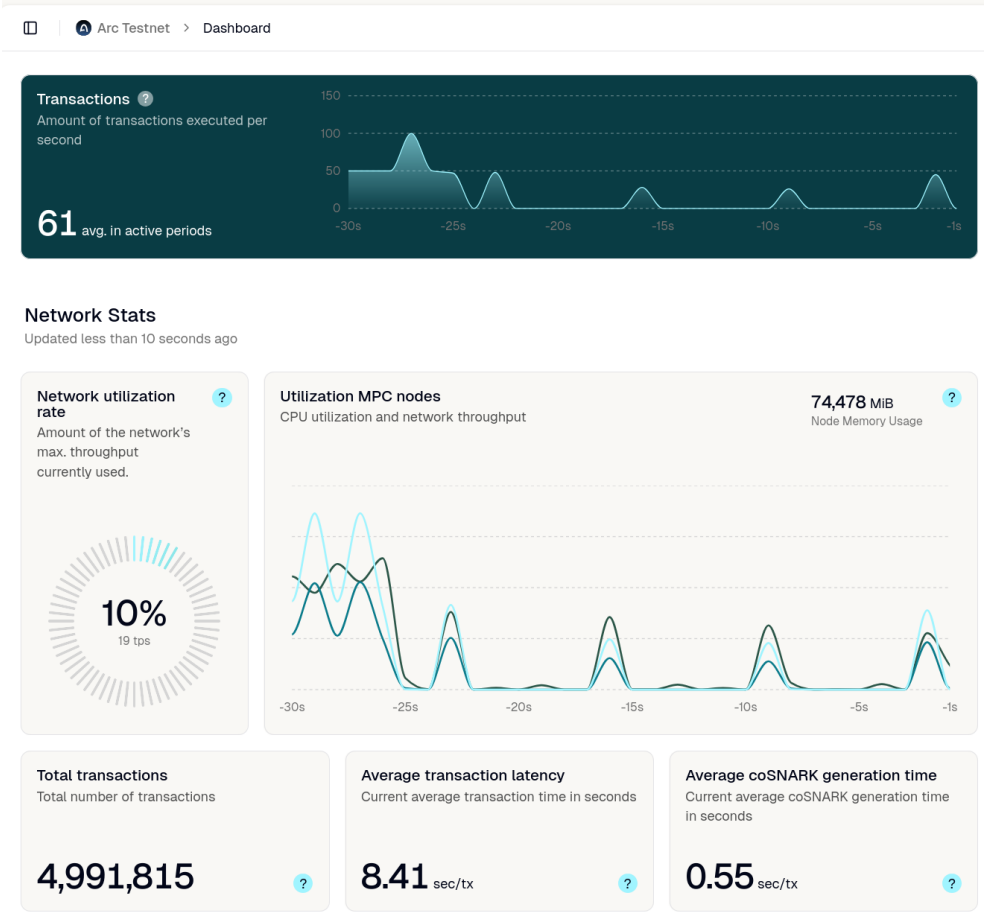


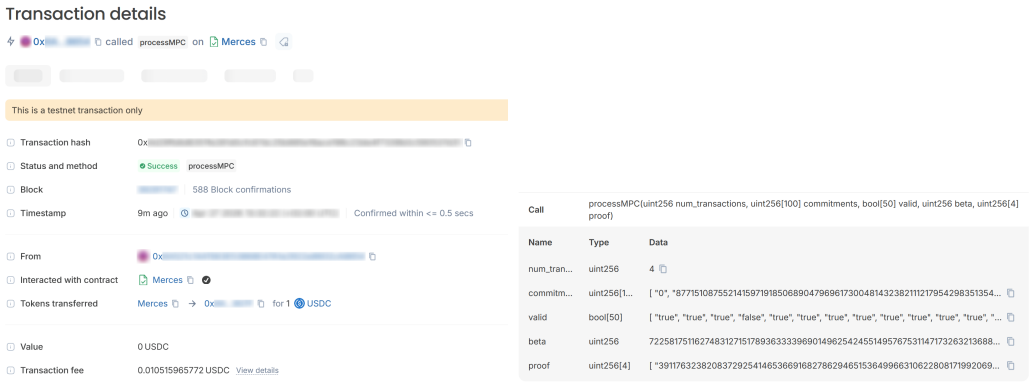
Figure 3: Merces demo — dashboard showing recent average transaction per second, network utilization, utilization of MPC nodes, total transactions, average transaction latency, and average coSNARK generation time.

7 On Extensions and Future Work

Merces can be generalized as follows. The MPC network computes one of many public functions $f_i(\cdot)$ on private inputs $\vec{x}$, where a smart contract holds a verifier $v_{f_i}(\cdot)$ for each function f_i, as well as a commitment to the inputs $C_{\vec{x}} \leftarrow \texttt{commit}(\vec{x})$. When the MPC network computes the output of one of these functions $\vec{y} \leftarrow f_i(\vec{x})$, it also computes a commitment to the output $C_{\vec{y}} \leftarrow \texttt{commit}(\vec{y})$, as well as a ZK proof $\pi_{f_i} \leftarrow \texttt{proof}(f_i; \vec{x}, C_{\vec{x}}, C_{\vec{y}})$ attesting to correct computation. It then posts π_{f_i} and $C_{\vec{y}}$ on-chain, where a smart contract verifies the proof $\texttt{valid} \leftarrow v_{f_i}(\pi_{f_i}, C_{\vec{x}}, C_{\vec{y}})$, and on success, accepts $C_{\vec{y}}$ as the commitment to the new output.

$f_i(\cdot)$ can be any function on private data; the smart contract can, in addition to the verifiers $v_{f_i}(\cdot)$, compute a public function $g_i(\cdot)$ on the public on-chain data. Merces instantiates this framework: user balances are commitments on-chain, the actual data is secret-shared on the MPC network, $f_i(\cdot)$ updates the balances, and $g_i(\cdot)$ is, e.g., a token transfer during a withdraw.

This naturally extends confidential transfers to further computations on private data. For example, private swaps: the contract holds, per user, multiple balance commitments for different denominations; $f_i(\cdot)$ deducts from one (e.g., USDC) and adds to another



(a) Block explorer view of an on-chain transaction calling the *processMPC* function of our Merces contract.

(b) Input data of a *processMPC* call to our Merces contract. The use of commitments and ZK proofs makes our token transfer system confidential.

Figure 4: Merces demo overview showing MPC processing and corresponding on-chain transactions.

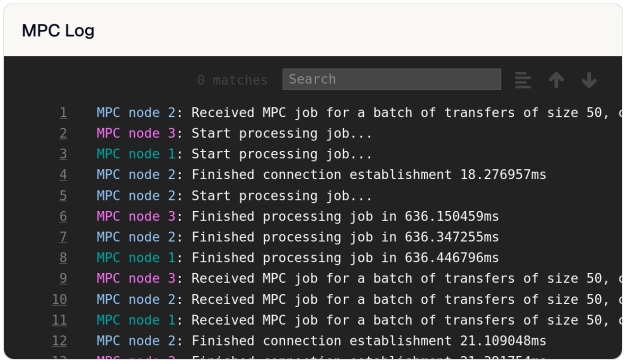


Figure 5: Merces demo — MPC log showing jobs processed by the MPC network.

(e.g., ETH), while $g_i(\cdot)$ performs the on-chain swap. If one denomination is a share of a yield-generating pool (e.g., Aave[30] or Morpho[31]), this gives private yield, and similarly private order books (dark pools).

As long as the circuit is efficient enough in ZK and MPC, Merces has no inherent extension limits. Other directions: implementing the standard ERC20 *approve* (storing and updating allowances analogously to balances) for smoother integration with existing applications; and serving as a confidential backend for the x402 protocol[32], where a server requests a payment before serving the response. The latter essentially requires a *transferWithAuthorization* functionality (cf. ERC-3009[33]) that enqueues a transfer intent on a valid user signature, where the signature is the one the client produces in x402.

Finally, replacing the secret-shared balance storage with an oblivious RAM-based structure [GO96] extends Merces to fully private transfers, hiding sender and receiver in addition to balances and amounts. Concrete performance of such an extension is left to future work.

[30] https://aave.com/
[31] https://morpho.org/
[32] https://www.x402.org/
[33] https://eips.ethereum.org/EIPS/eip-3009

References

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

[AKMQ23] JP Aumasson, Dmitry Khovratovich, Bart Mennink, and Porçu Quine. SAFE: Sponge API for field elements. Cryptology ePrint Archive, Report 2023/522, 2023.

[Ale26] Aleo. Aleo. https://aleo.org, 2026. Accessed: 2026-03-09.

[ARS20] Behzad Abdolmaleki, Sebastian Ramacher, and Daniel Slamanig. Lift-and-shift: Obtaining simulation extractable subversion and updatable SNARKs generically. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1987–2005. ACM Press, November 2020.

[AS22] Ghada Almashaqbeh and Ravital Solomon. SoK: Privacy-preserving computing in the blockchain era. In *2022 IEEE European Symposium on Security and Privacy*, pages 124–139. IEEE Computer Society Press, June 2022.

[Azt26] Aztec. Aztec. https://aztec.network/, 2026. Accessed: 2026-03-09.

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 423–443. Springer, Cham, February 2020.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.

[BCMR24] Foteini Baldimtsi, Kostas Kryptos Chalkias, Varun Madathil, and Arnab Roy. SoK: Privacy-preserving transactions in blockchains. Cryptology ePrint Archive, Report 2024/1959, 2024.

[BCT21] Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkHawk: Practical private smart contracts from MPC-based Hawk. In *IEEE International Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS 2021)*, pages 245–248. IEEE, 2021.

[Bea96] Donald Beaver. Adaptive zero knowledge and computational equivocation (extended abstract). In *28th ACM STOC*, pages 629–638. ACM Press, May 1996.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *20th ACM STOC*, pages 103–112. ACM Press, May 1988.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[BKSV21] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of groth's zk-SNARK. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part I*, volume 12674 of *LNCS*, pages 457–475. Springer, Berlin, Heidelberg, March 2021.

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

[BT22] Aritra Banerjee and Hitesh Tewari. Multiverse of HawkNess: A universally-composable MPC-based Hawk variant. *Cryptography*, 6(3):39, 2022.

[ByCDF23] Carsten Baum, James Hsin yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023, Part I*, volume 13950 of *LNCS*, pages 270–288. Springer, Cham, May 2023.

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

[Can20] Ran Canetti. Universally composable security. *Journal of the ACM*, 67(5):28:1–28:94, 2020.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Cham, April / May 2018.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Berlin, Heidelberg, August 2001.

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *28th ACM STOC*, pages 639–648. ACM Press, May 1996.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[DDO+01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 566–598. Springer, Berlin, Heidelberg, August 2001.

[DIO98] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Non-interactive and non-malleable commitment. In *30th ACM STOC*, pages 141–150. ACM Press, May 1998.

[DN02] Ivan Damgård and Jesper Buus Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 581–596. Springer, Berlin, Heidelberg, August 2002.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, Berlin, Heidelberg, August 2003.

[FHM98] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multi-party computation (extended abstract). In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 121–136. Springer, Berlin, Heidelberg, August 1998.

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In *31st FOCS*, pages 308–317. IEEE Computer Society Press, October 1990.

[GGJ+25] Sanjam Garg, Aarushi Goel, Abhishek Jain, Bhaskar Roberts, and Sruthi Sekar. Malicious security in collaborative zk-SNARKs: More than meets the eye. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part VII*, volume 16006 of *LNCS*, pages 396–428. Springer, Cham, August 2025.

[GGKS26] Sanjam Garg, Aarushi Goel, Dimitris Kolonelos, and Rohit Sinha. Jigsaw: Doubly private smart contracts. In *2026 IEEE Symposium on Security and Privacy*, pages 2288–2306. IEEE Computer Society Press, May 2026.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multiparty computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Berlin, Heidelberg, August 2015.

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021.

[GKS23] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *AFRICACRYPT 23*, volume 14064 of *LNCS*, pages 177–203. Springer, Cham, July 2023.

[GLR+20] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a generalization of substitution-permutation networks: The HADES design strategy. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 674–704. Springer, Cham, May 2020.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Cham, August 2017.

[GMY03] Juan A. Garay, Philip D. MacKenzie, and Ke Yang. Strengthening zero-knowledge protocols using signatures. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 177–194. Springer, Berlin, Heidelberg, May 2003.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for NP. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 339–358. Springer, Berlin, Heidelberg, May / June 2006.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016.

[GWKH25] Reimi Gai, Kaili Wang, Darren Kong, and Jacob Hirshman. Confidential ERC20 framework using fully homomorphic encryption (FHE). Whitepaper, https://github.com/Inco-fhevm/confidential-erc20-framework/blob/main/whitepaper.pdf, 2025.

[HM04] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 58–76. Springer, Berlin, Heidelberg, February 2004.

[KKK21a] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Composition with knowledge assumptions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 364–393, Virtual Event, August 2021. Springer, Cham.

[KKK21b] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. KACHINA - foundations of private smart contracts. In Ralf Küsters and Dave Naumann, editors, *CSF 2021 Computer Security Foundations Symposium*, pages 1–16. IEEE Computer Society Press, 2021.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016.

[KMSV21] Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 98–127. Springer, Cham, December 2021.

[KVW26] Dmitry Khovratovich, Mikhail Vladimirov, and Benedikt Wagner. Data matching in unequal worlds and applications to smart contracts. *CiC*, 2(4):33, 2026.

[KZM+15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015.

[Mon26] Monero Project. Monero. https://www.getmonero.org, 2026. Accessed: 2026-03-09.

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[MS25] Varun Madathil and Alessandra Scafuro. PriFHEte: Achieving full-privacy in account-based cryptocurrencies is possible. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part VII*, volume 16251 of *LNCS*, pages 199–229. Springer, Singapore, December 2025.

[MY04] Philip D. MacKenzie and Ke Yang. On simulation-sound trapdoor commitments. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 382–400. Springer, Berlin, Heidelberg, May 2004.

[Nat02] National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards Publication (FIPS) 180-2, https://csrc.nist.gov/files/pubs/fips/180-2/final/docs/fips180-2.pdf, 2002.

[OB22] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-knowledge proofs for distributed secrets. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 4291–4308. USENIX Association, August 2022.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, August 1992.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th FOCS*, pages 543–553. IEEE Computer Society Press, October 1999.

[SBV22] Samuel Steffen, Benjamin Bichsel, and Martin T. Vechev. Zapper: Smart contracts with data and identity privacy. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2735–2749. ACM Press, November 2022.

[SWA23] Ravital Solomon, Rick Weber, and Ghada Almashaqbeh. smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption. In *2023 IEEE European Symposium on Security and Privacy*, pages 309–331. IEEE Computer Society Press, July 2023.

[WBB20] Barry WhiteHat, Marta Bellés, and Jordi Baylina. ERC-2494: Baby Jubjub elliptic curve. Ethereum Improvement Proposal, https://eips.ethereum.org/EIPS/eip-2494, 2020.

[YLRDG26] Avishay Yanai, Meital Levy, Mike Rosulek, and Hila Dahari-Garbian. gcVM: Publicly auditable MPC via garbled circuits with applications to private EVM-compatible computation. Cryptology ePrint Archive, Report 2026/170, 2026.

A Related Work (Full)

For a detailed survey we refer the reader to [BCMR24], which focuses on private payments, and [AS22], which considers more general forms of private computation.

We firstly survey other works that do not follow our setting directly, yet are relevant in the direction of privacy-preserving smart-contracts and/or payments.

Trusted Execution Environments. Trusted Execution Environments (TEEs) have also been used to implement confidential smart contracts. Platforms such as Oasis[34] and Phala[35] execute contract logic inside hardware-protected enclaves that maintain private state, including user balances. Compared to cryptographic approaches, TEEs offer significantly improved performance since computations are performed in plaintext within the enclave. However, TEEs provide weaker security guarantees, as they rely on hardware trust assumptions and are susceptible to side-channel attacks. Numerous practical attacks against enclave implementations have been demonstrated, raising concerns about their long-term security.[36]

Fully Homomorphic Encryption Some works use Fully Homomorphic Encryption (FHE), which make them mostly interesting from a theoretical perspective. PriFHEte [MS25] proposes a solution that can be run as a smart contract on top of any account-based blockchain such as Ethereum. Balances are encrypted using FHE by users and updates are performed homomorphically on-chain. Their construction aims at providing full anonymity, hence hiding sender/receiver addresses. PrivFHEte is purely theoretical and has no implementation. Their focus is exclusively on private payments. SmartFHE [SWA23] is similar, only it uses multi-key FHE to enable multi-user contracts giving it greater generality. They implement and benchmark the costs of their FHE and ZK proof operations. However, they do not give any estimates of the gas costs of running these operations on-chain, as would be required by their construction. Computing these operations on-chain is most likely infeasible in practice. Recently, a confidential ERC-20 standard extension (dubbed cERC20) based on FHE was proposed in [GWKH25] in which they delegate heavy FHE computations to an off-chain server. However, current FHE-based solutions suffer from substantial performance limitations. Homomorphic operations and threshold decryption remain computationally expensive and often require powerful hardware. For example, TFHE [CGGI20] implementations achieve only hundreds of integer operations per second even on GPU-equipped servers[37], while CPU deployments provide significantly lower throughput[38]. Client-side proof generation can also be costly, requiring seconds of computation on multi-core machines[39]. These performance challenges currently limit the scalability of FHE-based payment systems.

Garbled Circuits. A recent MPC-based proposal gcVM [YLRDG26], which uses garbled circuits, claims to provide public auditability. In gcVm the input gets encrypted by

[34] https://oasis.net/
[35] https://phala.com/
[36] See https://sgx.fail/ for a summary of attacks.
[37] https://docs.zama.ai/tfhe-rs/get-started/benchmarks/gpu/gpu-integer-operations
[38] https://docs.zama.ai/tfhe-rs/get-started/benchmarks/cpu/cpu-integer-operations
[39] https://docs.zama.ai/tfhe-rs/get-started/benchmarks/zk-proof-benchmarks

the user using a symmetric AES key, which is then processed using Garbled Circuits accompanied with additively homomorphic encryption. These homomorphic commitments can then be checked by an external, public verifier which provides identifiable abort. This makes the integrity assumption optimistic, since the underlying blockchain assumes the encryption is correct until an external auditor challenges it. While the public auditor can be implemented using a smart contract, in practice, this is constrained by high homomorphic encryption costs. However, the presented proof-of-concept implementation again only guarantees malicious security in a dishonest-majority setting and does not include this public auditability property. Furthermore, even when comparing against our approach without accounting for the additional cost of proof generation, our system achieves significantly higher throughput.

UTXO model and privacy with new tokens. Privacy in the context of UTXO-based blockchains has been studied and deployed in practice. Zcash [BCG+14] and Monero [Mon26] are standalone cryptocurrencies set in the UTXO model. The main approach here is that whenever parties want to transact privately, they create a so-called note which can only be decrypted and spent by a dedicated receiver. This note is committed on-chain accompanied by a zero-knowledge proof, such that the on-chain data does not leak any information about the balances and transaction. When a party spends a note, it releases a nullifier, which prevents the note from being spent a second time. We refer the reader to [BCMR24] for more works/projects in this direction.

UTXO model and privacy on existing tokens. Zexe [BCG+20] builds on the Zcash [BCG+14] architecture and expands it to support general purpose computation beyond payments. Furthermore, Zexe also hides the function being computed.[40] It works as a smart contract atop any existing account-based blockchain, but follows the UTXO model instead which in turn creates the issue of blockchain storage growth and note discovery. In other words a wallet needs to scan all newly created notes to find which decrypt successfully with its secret key. Transactions are executed using the decentralized private execution (DPC) scheme, which consumes old records and creates new ones, accompanied by ZK proofs that the records are consistent and no function is malicious. While Zexe can in general be extended to support any application where statements about the users own data is proven, general-purpose computation involving data of multiple users requires a central coordinator that carries out the computation in the clear. Note that replacing this coordinator with an MPC network (similar to our work) can remove this privacy leakage. Zexe forms the foundation for systems such as Aztec [Azt26] and Aleo [Ale26].

Hawk [KMS+16] achieves general-purpose private computation in the UTXO model. It does so by having the blockchain store only commitments that hide the underlying data, and complement this with ZKPs to ensure correctness. However, to enable general-purpose computation the parties must trust a central coordinator that carries out the computation in the clear. Even though this party cannot break integrity, it certainly breaks privacy. On the other hand, zkHawk [BCT21] and Hawkness [BT22] replace this coordinator with an MPC protocol, which is somewhat similar to our setting, except they run this computation among the clients themselves—whereas we make use of external servers.

Similarly, Jigsaw [GGKS26] is a recent work that, building on top of the Zexe [BCG+20] and Zcash [BCG+14] architectures, presents a framework for privacy-preserving smart contracts where the actual computation, is not carried out by a central party and instead they are executed by a set of MPC servers. Integrity is ensured by the underlying blockchain itself due to the use of collaborative SNARKS, as in our setting. Unfortunately, Jigsaw shares the same drawback as Zexe: following the UTXO approach (even if set in the account-based model) leads to large blockchain storage and the issue of note discovery.

[40] To the best of our understanding, Zexe is the only work aiming at hiding the function itself.

Eagle [ByCDF23] uses MPC to enable general-purpose private computation, and also works as a smart contract on top of existing UTXO blockchains. Similar to us, they consider MPC servers that carry out the computation and maintain secret-shared state on behalf of the clients. In addition to the significant storage overhead imposed by the UTXO model, their approach ensures integrity through a non-collusion assumption among the servers, which they describe as 'insured MPC'. Even though they incorporate crypto-economic measures to discourage cheating (like public verifiability and identifiable abort), this results in a weaker property than our setting where integrity relies solely on the underlying blockchain itself.

Account-based model and privacy on existing tokens. Now, we focus on works that are closer to our topic of study, namely techniques for adding privacy in the account-based model. All of these works can be deployed as smart contracts on top of any account-based blockchain such as Ethereum, and enable "wrapping" existing currencies (e.g. ERC-20 tokens) with privacy.

Zether [BAZB20] achieves privacy in the account-based model by letting clients encrypt their balances using their own secret-key, and performing transactions by submitting the encrypted amount together with zero-knowledge proofs (ZKPs) of correctness. The balances are then updated homomorphically on-chain. The smart contract implements functionality that validates transactions before they are put on-chain. Further, the ownership of the balances can be transferred to the blockchain temporarily using the 'lock functionality'. Zether focuses on confidentiality, which corresponds to hiding the transfer amounts and the balances, but does not provide anonymity, which pertains to hiding the sender/receiver addresses. Zether provides an implementation, but acknowledges that the gas costs with > 7 million units of gas per confidential transfer are very high and barely feasible in practice. Zether describes several applications; however, they are limited to payment applications. Zapper [SBV22] extends this approach to hide both user identities and data access patterns using off-chain execution with oblivious Merkle trees and zero-knowledge proofs.

B Concrete Practicalities of Merces

Here we discuss in detail some of our considerations in the implementation of Merces, not captured by the formal protocol Π_{CToken} or the smart contract Φ_{merces} from Section 4.2. The workflow of a transaction in Merces is described below and visualized in Figure 1.

B.1 Client Posts Intent On-Chain

The client first posts an intent of the specific action on-chain:

- A *deposit* requires revealing the amount of tokens when posting the intent on-chain since an actual token transfer (from the user to the smart contract) is happening. The intent then gets added to the action queue for later processing by the MPC network.
- A *withdraw* also requires revealing the amount of tokens since a token transfer will take place (from the smart contract to the user) as well. Unlike a deposit, the token transfer will not take place when the client posts the intent into the action queue, but after the MPC network posts the proof of correctly updating the commitments, which includes a range check of the new balance of the user being non-negative.
- When registering a *transfer* intent, the user has to provide the following additional information: (i) The receiver address of the transfer. (ii) A commitment to the transfer amount, which will then later be used to bind the MPC update proof to the user intent.

(iii) Ciphertexts of the secret shares of the amount and randomness used to generate the commitment, since we implement the client-to-MPC communication via the smart contract. (iv) The client posts a ZK proof of correctness of the provided input (see below). If the ZK proof is valid (this is verified when the intent is posted in the smart contract), the transfer intent is put into the action queue and the ciphertexts and commitment are stored on-chain.

B.1.1 The Transfer Client ZK Proof

The role of the client ZK proof is ensuring that if an action is put into the action queue on-chain, then the only thing that can go wrong is that the sender (in a withdraw/transfer) does not have enough balance to fulfill the request. The latter is checked by the MPC network when processing the request. Consequently, if the range check of the updated sender balance being non-negative is encoded into the MPC ZK proofs in such a way that the proof itself is still valid (i.e., it verifies) but indicates an invalid transaction with a public input, then an action can verifiably be removed from the action queue without raising censorship concerns about the MPC network.

To this end, the client has to prove the following statements in a ZK proof when posting the transfer intent on-chain:

- The amount b alongside the randomness r produce the amount commitment C.
- The three provided secret shares b_1, b_2, b_3 are valid arithmetic secret-shares of b, i.e., $b = b_1 + b_2 + b_3$.
- The three provided secret shares r_1, r_2, r_3 are valid arithmetic secret-shares of r, i.e., $r = r_1 + r_2 + r_3$.
- The ciphertext intended for MPC party i is an encryption of (b_i, r_i) using the public key pk_i, for $i \in \{1, 2, 3\}$.
- The amount b must be non-negative. This is implemented as a range check limiting the transfer amount to 80 bits.

We implement encryption of the secret shares by first letting the client sample a fresh $sk \in \mathbb{F}_q$, and deriving a symmetric encryption key s as the x-coordinate of $sk \cdot pk_i$, where the MPC parties' public key pk_i is a point on the BabyJubJub[41] curve [WBB20] and $\mathbb{F}_q$ is the scalar field of the BabyJubJub curve. For actual encryption, we then use Poseidon2 following the specification proposed in the SAFE-API [AKMQ23]. To be able to decrypt the shares, the client needs to send $pk = sk \cdot G$ (where G is the generator of the BabyJubJub curve) to the MPC network (via the smart contract), which the MPC parties then can use to derive s using their secret key sk_i corresponding to pk_i.

To verify the ZK proof, the smart contract needs to use the following public inputs: (i) The three MPC public keys pk_1, pk_2, pk_3 which are already stored on-chain. (ii) The user-provided commitment C, ciphertexts, as well as pk used for key derivation.

To reduce gas cost for on-chain proof verification, we use the two-hash compression technique proposed in [KVW26], instantiated with SHA-256 [Nat02] as the Solidity-friendly hash function and Poseidon2 as the ZK-friendly one.

Regarding performance, on a regular consumer laptop (benchmarked on an AMD Ryzen AI 9 HX 370 CPU with 24 threads @3.3 GHz), proof generation including witness extension takes roughly $135\,ms$ regardless of whether two-hash compression is used or not.

[41] Note that BabyJubJub is chosen since its base field matches the scalar field of BN254, allowing proving curve operations much more efficiently.

B.2 Batched Processing by the MPC Network

The MPC network is able to read the action queue, as well as the corresponding ciphertexts of the secret shares from the blockchain. It then starts processing the request, which entails updating the secret-shared balance of the involved users and creating a ZK proof. Since our system is in general fast enough (see Section 6.3), and to save on-chain gas cost for proof verification, we let the MPC network create one proof attesting a batch of 50 transactions. We allow the MPC network to pad the batch with dummy transactions in case the action queue holds fewer than 50 transactions.

Note that proof generation requires six public inputs for each transfer: the old and new sender commitments, the old and new receiver commitments, the amount commitment, and the *valid* flag indicating whether the new sender balance is indeed ≥ 0. The MPC network only updates the balance shares of the involved parties if the *valid* flag is set to true.

B.2.1 Encoding Deposits and Withdraws as Transfers

To batch *deposits* and *withdraws* alongside *transfers* under a single ZK proof, we encode each as a degenerate transfer:

- **Deposit.** Use the old balance, old blinding, and new blinding of the deposit as the old receiver balance, old receiver blinding, and new receiver blinding of a transfer, respectively. Set the amount blinding to 0, the old sender balance to the deposit amount b, and both the old and new sender blindings to 0.

- **Withdraw.** Use the old balance, old blinding, and new blinding of the withdraw as the old sender balance, old sender blinding, and new sender blinding, respectively. Set the amount blinding to 0, the old receiver balance and old receiver blinding to 0, and the new receiver blinding to the withdraw amount b.

B.2.2 Implications of Batching

The MPC computations split into three phases: (i) updating the data structure, (ii) the *witness extension* that produces all auxiliary variables required by the ZK proof, and (iii) the actual distributed Groth16 proof generation. Within a batch, phase (i) must proceed in sequence so that a later transaction operates on the already-processed balances. This step, however, only consists of adding/subtracting balances and sampling a fresh random share as the new blinding value; both operations are non-interactive in replicated secret sharing and thus highly performant. Phase (ii) can then be fully computed in parallel for all transactions in the batch.

A subtlety here is that we update the data structure before knowing whether a transaction is valid: validity is only determined in phase (ii), where, among other values, the *valid* flag is computed. We therefore perform phase (i) on a *copy* of the shared balances, and apply the result to the actual shares only after phase (ii), and only if *valid* is set. Because later transactions in the batch may have already operated on balances modified by an invalid transaction, we track which addresses are involved in faulty transactions and allow each address to participate in at most one faulty transaction per batch; on a second occurrence we drop the subsequent transaction and pad the batch with a dummy instead.

Finally, with the witness extension complete for all applied transactions, we run CoGroth16 to produce the batched proof. The verifier requires 300 public inputs (five commitments plus the *valid* flag, for each of 50 transactions). On-chain verification incurs one elliptic-curve scalar multiplication per public input, which translates into substantial gas; this motivates our use of the two-hash compression technique of [KVW26], see Section 6.4.

B.3 On-chain Verification of the MPC Proof

After the MPC network has created the ZK proof for updating the data structure, it pushes the result into the smart contract. This requires posting two commitments (the new sender/receiver balance commitments) and the valid flag for each transaction, alongside the proof, on-chain. Furthermore, the MPC network also tells the smart contract how many transactions are actually processed in the batch, which implicitly tells the smart contract how many dummy transactions were appended to pad the batch.

For each of the processed transactions, the smart contract takes the next item from the queue and performs one of the following operations:

- If the item in the queue was a *deposit* and the *valid* flag is true, it updates the commitment of the receiving address with the corresponding one supplied by the MPC network.
- For a *withdraw* with the *valid* flag being true, it updates the commitment of the sender address with the corresponding one supplied by the MPC network and transfers the specified token amount to the address which registered the withdraw intent.
- For a *transfer* with the *valid* flag being true, both sender and receiver commitments are updated using the MPC-provided commitments.

In any case, the public inputs are prepared using the old sender/receiver commitments, the correct amount commitment, the MPC-provided new sender/receiver commitments and valid flags, so that the smart contract can then verify the ZK proof (with the inputs of the full batch), confirming the commitments were updated correctly according to the user intents. Furthermore, the processed items are then removed from the action queue.

Notice that the action queue can only be processed in order. Since any item that is put into the action queue will produce a valid ZK proof (see Section B.1.1), the MPC network cannot censor transactions by claiming the ZK proof will not verify. Consequently, faulty user intents in the action queue can only be removed by providing a valid ZK proof with the *valid* flag set to false. Thus, the MPC network can only halt the full Merces system by refusing to post ZK proofs on-chain, but can never censor specific addresses, nor misuse any funds.

C Using Homomorphic Commitments

Since our design is based on verifiably updating on-chain commitments, a natural question that might arise is whether our choice of hash-based commitments is actually superior to choosing homomorphic commitments (e.g., Pedersen commitments [Ped92]). We discuss the difference between choosing these types of commitments in this section.[42]

Let us start by sketching a solution based on homomorphic commitments, instantiated with Pedersen commitments over the BabyJubJub curve. First, all involved commitments will be replaced by homomorphic commitments. This includes the balance commitments stored on chain for each user, as well as the amount commitment provided by the client when initiating a transfer. This already has the following implications:

- The client transfer proof is with 10 047 R1CS constraints (11 172 constraints with two-hash compression) slightly more expensive than the hash-based one with 8 896 constraints (9 465 with compression).

[42] Replacing the homomorphic commitment with additively homomorphic encryption (e.g., ElGamal encryptions) leads to a system similar to Zether [BAZB20], where no MPC network is required anymore. This, however, comes at the cost of less flexibility for integrating confidential tokens into a wider array of use cases (see Section 7).

- On-chain storage, as well as calldata, for a commitment is two uint256 (one for each coordinate of the Pedersen commitment) instead of one uint256 for Poseidon, increasing gas costs.

Using the homomorphic property of the Pedersen commitment we can directly perform the updates of the balance commitments on-chain (after the MPC network confirmed validity). Consequently, the role of the MPC network is reduced to proving that a sender during a *withdraw* and *transfer* has enough funds to fulfill the transaction, as well as letting users query their own secret balances and blinding values (as required to implement an account based model instead of UTXO). Notice that a *deposit* does not reduce the users balance. Consequently, *deposits* do not require the MPC network and a *deposit* can be finalized when the user posts it intent.[43]

The ZK proof for a withdraw and transfer thus changes to the following statements:

- The amount b and randomness r produce the commitment C, which is required as public input to verify the proof.
- The valid flag is true if b is at most 100 bit. This serves as the check whether the balance $b \geq 0$.
- r is in range to be a field element in the scalarfield of BabyJubJub.

This proof can be verified by providing the updated sender commitment which is computed on chain. Despite only having to prove one commitment instead of five in case of the hash-based commitment solution, this circuit requires more R1CS constraints to fully express. Indeed, our implementation in Circom leads to 1884 R1CS constraints per transfer compared to 1597 constraints for the hash-based one. On the other hand, the proof only requires 3 public inputs to verify (two for the coordinates, plus the valid flag) instead of 6 for the hash-based proof, leading to some small gas savings when verifying the proof on-chain (with and without two-hash compression).

Summarizing, the proofs are slightly more expensive using homomorphic commitments and storing the commitments costs more gas. On the other hand, deposits do not require proofs by the MPC network and the MPC proof verification for withdraw and transfer is slightly more gas efficient due to having less public inputs. Consequently, the question of whether homomorphic commitments are better boils down to how efficiently the MPC-network can compute the ZK proof. In other words, how MPC friendly is proving Pedersen commitments using CoGroth16.[44] While computing the actual commitment as $[\![b]\!] \cdot G + [\![r]\!] \cdot H$ (i.e., secret scalar times public point and adding two secret points) is very MPC friendly, proving this relation in ZK requires proving the multiply-and-add algorithm for the scalar multiplications where the witness consists of the (x, y)-coordinates of many intermediate points, alongside some other intermediate values. Computing these extra witnesses in MPC, however, is significantly more expensive compared to just using Poseidon2 commitments. Indeed, preliminary benchmarks show that computing the witness extension of a batch of 50 transactions using Pedersen commitments requires $\approx 1.5\,s$, compared to the $< 50\,ms$ for the hash-based ones.

[43] The MPC network needs to listen to on-chain events emitted by a deposit to update the secret shares of the balance accordingly.

[44] We do not consider alternative proof systems in this comparison. One concrete example would be using Bulletproofs [BBB+18] which is highly efficient in proving range checks of pre-images of Pedersen commitments. However, Bulletproofs has the disadvantage of inefficient verifiers, which scale logarithmically with the circuit size. Indeed, a proof attesting to one 100-bit range check would cost ≈ 2 million gas to verify on-chain already.