LucidiTEE: A TEE-Blockchain System for Policy-Compliant Multiparty Computation with Fairness

Rohit Sinha
Visa Research
rohit.sinha@visa.com

Sivanarayana Gaddam
Visa
sgaddam@visa.com

Ranjit Kumaresan
Visa Research
rakumare@visa.com

ABSTRACT

Motivated by recent advances in exploring the power of hybridized TEE-blockchain systems, we present LucidiTEE, a unified framework for confidential, policy-compliant computing that guarantees fair output delivery. For context:

- Kaptchuk et al. (NDSS'19) showed that enclave-ledger interactions can enable applications such as one-time programs and rate limited logging. We generalize their ideas to enforcing arbitrary *history-based* policies within and across several multi-step computations. Towards this, we define a new ideal functionality for policy-compliant computing $\mathcal{F}_{\text{PCC}}$, and then show how LucidiTEE implements $\mathcal{F}_{\text{PCC}}$.
- Chaudhuri et al. (CCS'17) showed that enclave-ledger interactions enable fair exchange, contract signing, and more generally, fair secure multiparty computation. In a setting with n processors each of which possesses a TEE, they show how to realize fair secure computation tolerating up to t corrupt parties for any $t < n$. We improve upon their result by showing a novel protocol which requires only t out of the n processors to possess a TEE. When $n = 2$ and $t = 1$, this provides practical fair computation in client-server settings, where clients may not possess a TEE.
- Ekiden (EuroS&P'19) and FastKitten (Sec'19) use enclave-ledger interactions to enable practical, privacy-preserving smart contracts. However, Ekiden and FastKitten store the contract's inputs on-chain (during malicious behavior in the latter's case). In contrast, LucidiTEE implements privacy-preserving stateful computation while storing inputs, outputs, and state off-chain, using the ledger only to enforce history-based policies.

Summarizing, LucidiTEE enables multiple parties to jointly compute on private data, while enforcing history-based policies even when input providers are offline, and fairness to all output recipients, in a malicious setting. LucidiTEE uses the ledger only to enforce policies; i.e., it does not store inputs, outputs, or state on the ledger, letting it scale to big data computation. We show novel applications including a personal finance app, collaborative machine learning, and policy-based surveys amongst an apriori-unknown set of participants.

1 INTRODUCTION

Alice wishes to analyze her monthly spending behavior using a personal finance application, such as Mint [1]. Moreover, she seeks to gauge, and even control, how her transaction records are stored, analyzed, and further shared with other parties — however, mainstream applications today only present a privacy policy written in legalese, without any technical means to enforce them. We discuss Alice's requirements from a new, transparent personal finance application that we build in this paper, called Acme.

Ideally, Acme does not access her raw transaction data, and she opts for only select functions from Acme (e.g. a monthly aggregate summary), and control who can attain the output report. Moreover, she cannot provision her own servers (to engage in a multiparty computation (MPC) protocol with Acme, for instance), and is expected to go offline at any time. For privacy and correctness, we must enforce a policy over what inputs are supplied to a computation: 1) only a single instance of the analysis, over an entire month's worth of transactions, is permitted, as opposed to multiple, finer-grained analytics that can leak significantly more information about her spending behavior (*one-time program* [2] policy); 2) all of her monthly transactions are fed as input to the computation.

As another example, Acme wishes to run a survey to collect feedback from users (such as Alice), without having to build a survey application but rather outsourcing it to a third-party service. Ideally, instead of trusting them to collect genuine responses, Acme wishes to enforce the following policy: 1) only accept inputs from enrolled users of Acme; 2) all valid inputs are tallied in the output.

These policies are expressed over inputs and outputs over the entire history of a computation, and across histories of multiple computations, and we call them **history-based policies**. It is an open research problem to efficiently enforce such policies in a multiparty setting (such as the apps above), where the participants are not known in advance, may not be online, and may act maliciously.

In addition to enforcing policies, we wish to ensure **fair output delivery** to all participants, even when the computation is carried out by malicious parties (e.g. when Acme computes on Alice's data, or collaboration between businesses (see § 9.1). It is an open problem how to provide fairness [3, 4] — if any party gets the output, then so must all honest parties — in a multi-party computation (malicious setting), where a subset of the participants have commodity devices without a trusted execution environment [4], such as end users.

In this work, we build on recent advances in exploring the power of a hybridized TEE-blockchain system (cf. [4–6] and references therein) to address the problem of *policy-compliance* on computations over user data. To that end, we provide a concrete **definition for policy-compliant computation**. Our definition takes the form of an ideal functionality $\mathcal{F}_{\text{PCC}}$ in the UC framework of [7]. $\mathcal{F}_{\text{PCC}}$ accepts user data and user-defined policies as inputs, and guarantees (1) confidentiality of inputs on which computations are performed, and (2) fair output delivery to output recipients, and (3) execution of only policy-compliant computations.

$\mathcal{F}_{\text{PCC}}$ internally maintains a log of all function evaluations across all concurrent computations, and a computation-specific policy check uses this log to determine whether to allow any further function evaluation. Additionally, the interfaces provided by our $\mathcal{F}_{\text{PCC}}$ abstraction are well-suited to the practical setting of repeated (big data) computations on user data which may grow over time, thereby expressive enough to enforce policies on this important class of applications. In more detail, parties provide their input data to $\mathcal{F}_{\text{PCC}}$ once, and then bind it to an unbounded number of different computations, and also use it for an unbounded number of steps within a computation without having to resupply it. This interface is valuable for Acme, whose input database contains information of a large number of merchants and spans several GBs. $\mathcal{F}_{\text{PCC}}$ allows any (malicious) party to carry out the computation on behalf of the computation's input providers (e.g. a cloud compute provider), but the properties of policy compliance and fairness are ensured.

Next, we present LucidiTEE, a hybrid TEE-blockchain system that exploits enclave-ledger interactions to provide a **practical implementation of the abstractions provided by $\mathcal{F}_{\text{PCC}}$**. We assume a method that computes on encrypted data — for instance, MPC protocols or TEE-based systems such as VC3 [8], Opaque [9], StealthDB [10], Ryoan [11], Felsen et al. [12], etc. — and instead describe methods to enforce history-based policies and fair output delivery. While a variety of advanced cryptographic methods exist to enable confidentiality of computations [13–15], enclaves provide perhaps the most practical method for achieving this. Pure cryptographic methods also fall short of guaranteeing fair output delivery [16], or enforcing policies across computations involving different sets of parties. Also, secure computation does not apply to settings where participants are not known in advance or are offline or where confidentiality is required when different sets of users contribute input to different stages of a single computation [17, 18].

Improvements upon Related Work. While enclaves address many of the problems above, they still suffer from other problems such as rollback protection in a multiparty computation. Prior work has employed blockchains for addressing state continuity, and also to support more robust designs involving a distributed network of TEE nodes [6, 19]. Our work continues in the same vein. However, in addition to rollback protection, we rely on enclave-ledger interactions to (1) enforce policy compliance, and (2) guarantee fair output delivery. In the following, we first discuss how we extend ideas from Kaptchuk et al. [5] (see also [20]) to use enclave-ledger interactions to enforce policies in computations. After that, we discuss how we improve upon the work of Choudhuri et al. [4] to derive more practical protocols for fair output delivery and fair exchange. The latter may be of independent interest.

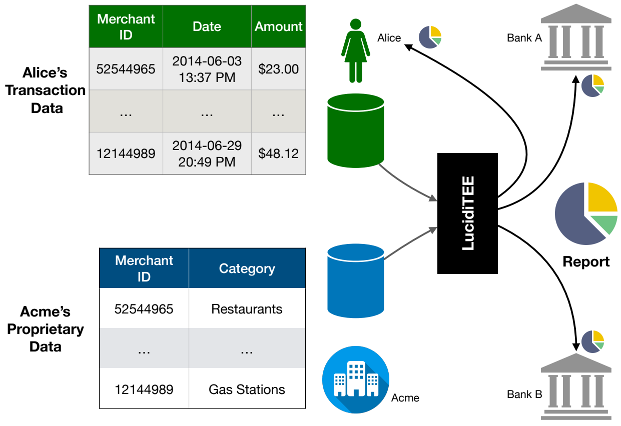
Kaptchuk et al. [5] showed that enclave-ledger interactions can enable applications such as one-time programs and "rate limited mandatory logging." To support applications such as one-time programs, [5]'s strategy is to record (the first) execution of the program on the blockchain. Then, the enclave running the program would first check the blockchain to see if the program was executed already (in which case it would abort), and if no record of program execution exists on the blockchain, then continue execution of the program. In the problem of rate limited mandatory logging, the goal is to log access to sensitive files before the necessary keys for an encrypted file can be accessed by the user. Here again, [5]'s strategy is to first check the blockchain to log the file access, and only then let the enclave release the key. See [5] for more details.

Extending [5], we provide general techniques to enforce arbitrary history-based policies within and across several multistage computations. At a high level, we implement such history-based policies by allowing the enclave executing a step of a computation to scan through the ledger to identify policies associated with the computation, and first check if the inputs to the computation step comply with the policies associated with the computation, and only then proceed with the execution of the computation step. In the following sections, we demonstrate several interesting practical applications which exploit the power of history-based policies. We note that such an extension is not straightforward from the "Enclave-Ledger Interaction" scheme suggested by [5]—among other things, concretely, their ExecuteEnclave function takes only a single ledger post, whereas our policies may involve multiple entries on the blockchain. Furthermore, unlike [5], our abstraction $\mathcal{F}_{\text{PCC}}$, and consequently LucidiTEE, can enforce policies across several computations. As an example, consider a survey application where one might wish to enforce a policy that only those users who have participated in a prior survey may participate in the current one.

Next, we discuss our contributions to the design of practical fair exchange and **fair computation** protocols. By fairness, we mean that either all output recipients obtain the output of a computation or none do. It is a well-known result [16] that fairness is impossible to achieve in a setting where a majority of the participants are corrupt. Faced with this result, several works have explored relaxed notions of fairness over the last few years [21–23]. However, very recently, Choudhuri et al. [4] showed that enclave-ledger interactions can enable fair secure computation. (Note that it is not known whether enclaves alone can enable the standard notion of fairness in secure computation [24].) In a setting with n processors each of which possesses a TEE, [4] show how to realize fair computation tolerating up to t corrupt parties for any $t < n$. We improve upon their result with a novel protocol which requires only t out of the n processors to possess a TEE. When $n = 2$ and $t = 1$, this provides practical fair computation in client-server settings, where clients may not possess TEEs. This contribution is of independent interest.

System Design. LucidiTEE provides a practical and scalable framework for privacy-preserving, policy-compliant computations. In our example applications, the inputs span large databases (such as Acme's proprietary database in § 9.1.1). The inputs also arrive from a large number of apriori-unknown users, such as in public surveys (§ 9.1.2) and applications that provide a service based on aggregate data of its growing consumer base (e.g., § 9.1.3). Finally, the set of computations grow over time as parties enroll in new computations that consume results from prior computations.

LucidiTEE supports history-based policies with the use of a shared ledger or blockchain, which plays the role of the log in $\mathcal{F}_{\text{PCC}}$ — since rollback or tampering attacks on the log can violate policy compliance and fairness, we use a shared ledger (accessible by all parties) in lieu of a centralized database. LucidiTEE achieves scalability by minimizing use of the ledger. To support the above mentioned applications, we record only commitments (i.e., hash digests, which support membership queries) of the inputs, outputs, and the updated state on the ledger after each function evaluation. We note that the recent works of Ekiden [6] and FastKitten [25] also use enclave-ledger interactions to enable privacy-preserving smart contracts. However, Ekiden and FastKitten store the contract's inputs on-chain (during malicious behavior in the latter's case). In contrast, LucidiTEE stores inputs, outputs, and state off-chain (recall only commitments to these go on-chain), using the blockchain only to enforce history-based policies. Furthermore, we use remote attestation [24, 26] to allow any party to act as a compute provider by providing a genuine TEE-enabled processor. For these reasons, we say that LucidiTEE embodies "bring-your-own-storage" and



Alice's Transaction Data

Merchant ID	Date	Amount
52544965	2014-06-03 13:37 PM	$23.00
...	...	...
12144989	2014-06-29 20:49 PM	$48.12

Acme's Proprietary Data

Merchant ID	Category
52544965	Restaurants
...	...
12144989	Gas Stations

Figure 1: Transparent Personal Finance Application

"bring-your-own-compute" paradigms , lending flexibility to the untrusted compute providers to manage their storage and compute.

In summary, we make the following contributions:

- definition of an ideal functionality $\mathcal{F}_{\text{PCC}}$ for multi-party, concurrent, stateful computations, with enforcement of history-based policies for offline parties and fairness for all output recipients
- LucidiTEE, a system that realizes this ideal functionality, using TEEs and a shared ledger
- protocol for fair n-party output delivery, requiring a shared ledger and t parties to allocate a TEE, for any corruption threshold $t < n$. We also provide a formal proof of security in the UC framework.
- evaluation of several applications, including a personal finance application (serving millions of users), federated machine learning over crowdsourced data, and a private survey. We also implement one-time programs, digital lockboxes, and fair exchange.

2 OVERVIEW OF LUCIDITEE

In this section, we introduce the components of our system using an example personal finance application. The design principles behind our system should be evident even when considering applications from different domains such as joint machine learning, etc.

2.1 Motivating Example: Personal Finance App

The open banking initiative [27] has fostered a variety of personal financial applications. Figure 1 illustrates a sample financial application by Acme, who provides a service for viewing aggregate spending behavior (i.e., the proportion of spending across categories for all transactions in a month), along with the feature to share this aggregate report with third parties (such as lending institutions who can provide mortgage offers).

To perform this joint computation, Acme maintains a proprietary database mapping merchant ids to category labels; Alice's data consists of a set of transaction records sorted by time, where each record contains several sensitive fields such as the merchant id, the amount spent, and the timestamp. The aggregation function (denoted by f) is evaluated over inputs from Alice and Acme, and the output is shared with Alice and two lending institutions, BankA and BankB. Alice's data is either imported manually by her, or more conveniently, provided by Alice's bank, via an OAuth-based OFX API [27] that pushes transaction data one-by-one as they are generated by her. Today, an application like Acme often hosts the users' raw data, and is trusted by them to adhere to a legalese policy.

2.2 Requirements of Acme

The application may be hosted by a malicious compute provider, who may collude with any party to violate either Alice's or Acme's privacy. Therefore, we discuss some concrete requirements of the system to ensure guarantees of policy compliance and fairness.

Privacy through Transparency

We find that transparency and control — i.e., enforcing which functions can be evaluated, and with whom the outputs are shared — are necessary for enforcing any measure of privacy. Without this basic enforcement, an attacker can proliferate arbitrary functions of sensitive user data. In our example, Alice wishes that the approved output recipients only learn the output of function f (on one months' worth of transaction data), and nothing else about her transaction data, such as her spending profile at daily granularity or the location patterns. For this reason, f cannot be evaluated by sharing Alice's plaintext data with Acme, or vice versa as Acme also wishes to maintain secrecy of its proprietary database.

> **Strawman Approach**
> Both parties first encrypt their data before uploading it to a *compute provider* (such as Acme or a cloud service). Next, to allow the agreed-upon evaluation of function f on their data, both parties establish a TLS channel (based on remote attestation) with an enclave program (loaded with function f) running on an untrusted host software at the compute provider, and share the decryption keys to the enclave. Then, the enclave evaluates f, encrypts the output under the output recipients' public keys, and asks the compute provider to transmit the encrypted output.

As a first step towards transparency and control, this design ensures that only f is computed on inputs from Alice and Acme, and that no other party beyond Alice, BankA, and BankB can observe the output. Note that the input providers can go offline after providing their inputs, and the output recipients come online only to retrieve the outputs. There are several TEE-based systems that fit this general design, such as VC3 [8], Opaque [9], StealthDB [10], Ryoan [11], Felsen et al. [12], etc., which we can use to implement the function f. To restrict scope, f is assumed to be safe (e.g., without additional leaks via side channels), so we execute f as given.

History-based Policies

While this strawman ensures that only f can be evaluated on Alice's input, we illustrate how this is insufficient for Alice's privacy.

Recall that Alice's bank (we can treat Alice and her bank as one logical party) uploads encrypted transaction records to the compute provider (using the OFX API [27]), one-by-one as they are generated by Alice. The enclave's host software is controlled by an untrusted compute provider, who may perform attacks such as rewinding the persistent storage and launching multiple instances of the enclave program. Hence, an adversarial compute provider may repeat the computation with progressively smaller subsets of Alice's (encrypted) transaction data from that month (and collude to send the output to a corrupt party) — note that each of these computations is independently legal since it evaluates f on an input containing Alice's transactions that are timestamped to the same calendar month. By comparing any pair of output reports, the attacker infers more information about the transactions than what is leaked by the monthly aggregate report; for instance, one may learn that Alice tends to spend frivolously towards the end of the month[1]. In general, this form of a *rewind-and-fork* attack is detrimental for applications that maintain a privacy budget [28].

To counter such attacks, we enforce *history-based policies*, where the decision to execute the next step in a computation depends on that computation's history (and the history of any other computations over the input data) which contains some metadata about the inputs and outputs of a computation. In Acme's example, Alice uses the following history-based policy ϕ : all transactions must 1) be fresh, in that they have never been used by a prior evaluation of f, and 2) belong to the same month.

History-based policies find use in applications that maintain state, have privacy budgets, or make decisions based on prior inputs. We urge the reader to look at history-based policies in § 9.1, such as private surveys amongst unknown participants with policies across computations (e.g. survey only open to users who participated in a previous survey, or only open to Acme users) — smart contracts on Ekiden [6] or FastKitten [25] cannot read the ledger entries (of other contracts), and therefore cannot implement such applications.

To our knowledge, this is the first work to study such policies in a multi-party setting (where participants may be offline or act maliciously), and enforcing them incurs the following challenges. For instance, multiple parties may compute concurrently, and attempt to append the computation's history on the shared ledger — there must be some conflict resolution to enable concurrency across computations, but also ensure policy compliance. Furthermore, we must develop efficient methods to check policies such as *k-time programs* and accounting of all inputs.

Fairness

A policy also enforces the set of output recipients: Alice, BankA, and BankB. Simply encrypting the output under their public keys ensures that other parties cannot observe the results of the computation (assuming confidentiality of enclave execution). However, a malicious compute provider can collude with a subset of output recipients, and deliver the encrypted outputs to only those parties — since all network communication is proxied via the compute provider, an enclave cannot ensure that an outbound message is sent, and therefore, must assume lossy links, making reliable message delivery impossible [29].

Without having to trust Acme, Alice wishes to have her monthly reports sent to a set of chosen banks, perhaps to get lucrative loan offers. For Acme to be transparent, we argue that it must also ensure fairness: if any party gets the output, then so must all honest parties. Moreover, a protocol for fair output delivery should ideally not require Alice to possess a device with a TEE — the enclave-ledger protocol in [4] requires all parties to possess a TEE.

2.3 Acme on LucidiTEE

Specifying and Creating Computations

A computation's semantics is specified by a string, which anyone can post to the ledger for inspection by all parties. In Acme's case:

```
computation { id: 42, /* unique id */
  inp: [("txs": pk_Alice), ("db": pk_Acme)],
  out: [("rprt": [pk_Alice, pk_BnkA, pk_BnkB])],
  policy: 0xcoff..eeee, /* ∀r ∈ txs. ¬ consumed(r) */
  func: 0x1337...c0de /* aggregate function */ }
```

Each computation on LucidiTEE has a unique `id`. The `in` field lists a set of named inputs, along with the public key of the input provider (who is expected to sign those inputs). Similarly, the `out` field lists a set of named outputs, where each output has one or more recipients. The evaluation function f and the policy function ϕ are implemented as enclave programs, for confidentiality and integrity, and we uniquely identify them using the hash of the enclave binary.

A computation progresses via a potentially unbounded sequence of stateful evaluations of f, guarded by ϕ, and it is said to be *compliant* if all constituent steps use the function f and satisfy ϕ. Unlike f, ϕ takes the entire ledger as input. In our example, ϕ encodes the freshness property that no transaction within `txs` has been consumed by a prior evaluation of f; we can implement ϕ by performing membership test for each transaction (in `txs`) within the inputs consumed by prior evaluations of f in computation of `id` 42, or more efficiently, by maintaining state containing an accumulator (e.g. a Merkle tree) of all transactions previously consumed by f.

Enforcing Policies and Performing Computation

History-based Policies via Shared Ledgers
We introduce an append-only ledger, which is shared at least between Alice and Acme, but more generally, a global set of users to enforce policies across multiple applications that process some data. The ledger fulfills a dual purpose. First, a protocol (see § 7) forces the compute provider to record the enclave's evaluation of f on the ledger before extracting the output — for each function evaluation, we record a hash-based commitment of the encrypted inputs, outputs, and intermediate state. Second, enclave programs read the ledger to evaluate the policy ϕ.

The compute provider allocates a TEE machine, and downloads Alice's and Acme's encrypted inputs onto the machine's local storage — this expense may be amortized across several function evaluations. Next, Acme must convince an enclave that the requested function on Alice's inputs is compliant. To that end, Acme launches an enclave implementing the policy predicate ϕ, and provides it with a view of the ledger. Note that a malicious compute provider can provide a stale view of the ledger (by simply ignoring recent ledger entries), and our protocol defends against such attacks by requiring a proof that no relevant computation occurred since the ledger height at which ϕ is evaluated. On approval from ϕ, Acme launches an enclave to evaluate f, which gets the encrypted inputs'

[1] While metadata, such as authenticated batches of inputs, can remedy this attack, banks may be unwilling to generate this data for each third-party app.

decryption keys from a key manager (also implemented as an enclave; see § 7), and produces an output encrypted under the public keys of all output recipients. In § 7, we discuss practical methods to enforce several classes of policies.

LucidiTEE is oblivious to how or where the encrypted data is stored, and the ledger size is independent of the size of the inputs. Therefore, we stress that LucidiTEE uses the ledger only to enforce policies, and embodies a "bring-your-own-storage" paradigm. Moreover, since LucidiTEE uses trusted enclaves and an append-only ledger to enforce the policy, any (malicious) compute provider can bring TEE nodes and evaluate ϕ and f. Hence, we emphasize that LucidiTEE also embodies a "bring-your-own-compute" paradigm.

Fair Reconstruction of Outputs

Since fairness is impossible to achieve in a setting where a majority of the participants are corrupt [16], several works have explored relaxed notions of fairness over the last few years [4, 21–23] — specifically, Choudhuri et al. [4] showed that enclave-ledger interactions can enable fair secure computation. Our work continues in the same vein, and improves upon their result.

Inspired by [30, 31], we reduce fair computation to fair reconstruction of an additive secret sharing scheme, as follows. The enclave's evaluation of f encrypts the output under a fresh, random key k. The enclave also emits shares of k for all output recipients: Enc(pk_Alice, k_1), Enc(pk_BankA, k_2), and Enc(pk_BankB, k_3), s.t. $\mathsf{k} \doteq \mathsf{k}_1 \oplus \mathsf{k}_2 \oplus \mathsf{k}_3$, for random k_1, k_2, and k_3. All output recipients must engage in the reconstruction protocol with their key shares. For best case security (static corruption model), we set the corruption threshold t in Acme's example to 2, thus withstanding byzantine behavior from any 2 of the 3 parties. The protocol requires $t = 2$ parties to provide a TEE node (e.g., BankA and BankB).

Protocol for Fair N-Party Reconstruction
The protocol withstands an arbitrary corruption threshold $t < n$, and it requires any t recipients to allocate a TEE machine, and all n parties to access the shared ledger — in contrast, [4] needs all n parties to allocate a TEE machine and access the ledger, which is cumbersome for end users.

3 IDEAL FUNCTIONALITY OF LUCIDITEE

We define an ideal functionality that performs concurrent, stateful computations amongst arbitrary sets of parties. The universe of parties is an unbounded set, denoted by P^*, of which any subset of parties may engage in a computation. Each computation $\mathbf{c}$ involves a set of input providers P^c_{in} and a set of output recipients P^c_{out}, which may overlap, such that $(P^c_{\text{in}} \cup P^c_{\text{out}}) \subseteq P^*$. We assume a polynomial-time static adversary $\mathcal{A}$ that corrupts any subset $P^{\mathcal{A}} \subseteq P^*$, who act maliciously and deviate arbitrarily from the protocol. The attacker selects the inputs and learns the output of each party in $P^{\mathcal{A}}$.

We introduce an ideal functionality for policy-compliant computing, $\mathcal{F}_{\text{PCC}}$. Each step of a computation evaluates a transition function f if allowed by the policy predicate ϕ (expressed over the inputs and a ledger of all prior function evaluations). Each computation is defined by a specification $\mathbf{c}$, which fixes f, ϕ, and the identities of the input providers P^c_{in} and the output recipients P^c_{out}. Due to compute chaining, $\mathbf{c}$ also specifies the computations from which it receives inputs (denoted $C^{\mathbf{c}}_{\text{in}}$), and computations which consume its outputs (denoted $C^{\mathbf{c}}_{\text{out}}$). Since input providers may go offline after binding their input to a computation $\mathbf{c}$, they are not required to authorize each step of $\mathbf{c}$ — an input may be used for an unbounded number of steps of $\mathbf{c}$ (for instance, Acme's input is used to service multiple users for multiple months), as long as $\mathbf{c}.\phi$ approves each step and $\mathbf{c}$ has not been revoked. Moreover, an input may be bound to several computations concurrently, avoiding unnecessary duplication of the data. $\mathcal{F}_{\text{PCC}}$ is defined as follows:

Policy-compliant Computing: $\mathcal{F}_{\text{PCC}}$
The functionality has a private storage db and a publicly readable log ldgr, which are empty at initialization. db is indexed by handles, and supports an add(x) interface (which returns a unique handle h from the set $\{0,1\}^{\text{poly}(\lambda)}$) and a update($h$,$x$) interface. On parsing $\mathbf{c}$, we get the set of input recipients $P^{\mathbf{c}}_{\text{in}}$, output recipients $P^{\mathbf{c}}_{\text{out}}$, and the set of chained computations $C^{\mathbf{c}}_{\text{in}}$ and $C^{\mathbf{c}}_{\text{out}}$. The **ret** statement returns a value to a party and terminates execution of the enclosing command, whereas **send** continues the execution.
Let active($\mathbf{c}$) $\doteq$ (create$\|\mathbf{c}$) $\in$ ldgr $\wedge$ (revoke$\|\mathbf{c}$.id) $\notin$ ldgr
Let data(h) $\doteq x$ if $(x, \_, \_) \in$ db[h] else $\bot$

```
• On command create_computation($\mathbf{c}$) from $p \in P^*$
  send (create $\|$ $\mathbf{c}$ $\|$ $p$) to $\mathcal{A}$
  if $\exists \mathbf{c}'$ (create $\|$ $\mathbf{c}'$) $\in$ ldgr $\wedge$ $\mathbf{c}'$.id = $\mathbf{c}$.id { ret $\bot$ to $p$ }
  ldgr.append(create $\|$ $\mathbf{c}$); ret $\top$ to [$p$, $\mathcal{A}$]
• On command revoke_computation($\mathbf{c}$.id) from $p \in P^c_{\text{in}}$
  send (revoke $\|$ $\mathbf{c}$.id $\|$ $p$) to $\mathcal{A}$
  if $\neg$active($\mathbf{c}$) { ret $\bot$ to [$p$, $\mathcal{A}$] }
  ldgr.append(revoke $\|$ $\mathbf{c}$.id); ret $\top$ to [$p$, $\mathcal{A}$]
• On command provide_input($x$) from $p \in P^*$
  $h \leftarrow$ db.add(($x$, $p$, $\emptyset$))
  send (provide_input $\|$ $|x|$ $\|$ $h$ $\|$ $p$) to $\mathcal{A}$;  ret $h$ to $p$
• On command bind_input($\mathbf{c}$.id, $h$) from $p \in P^c_{\text{in}}$
  send (bind_input $\|$ $p$ $\|$ $\mathbf{c}$.id $\|$ $h$) to $\mathcal{A}$
  if ($\neg$active($\mathbf{c}$) $\vee$ db[$h$] = $\bot$) { ret $\bot$ to [$\mathcal{A}$, $p$] }
  let ($x$, $p'$, $C$) $\leftarrow$ db[$h$]; if ($p' \neq p$) { ret $\bot$ to [$\mathcal{A}$, $p$] }
  db.update($h$, ($x$, $p$, $C \cup \{\mathbf{c}\}$));
  ldgr.append(bind $\|$ $\mathbf{c}$.id $\|$ $p$ $\|$ $h$);  ret $\top$ to [$\mathcal{A}$, $p$]
• On command compute($\mathbf{c}$.id, $H_{\text{in}}$) from $p \in P^*$
  send (compute $\|$ $\mathbf{c}$.id $\|$ $H_{\text{in}}$ $\|$ $p$) to $\mathcal{A}$
  if $\mathcal{A}$ denies or $\neg$active($\mathbf{c}$) then { ret $\bot$ to [$\mathcal{A}$, $p$] }
  let $h_s$ be the handle to the latest state of $c$, based on the ldgr
  let $s \leftarrow$ data($h_s$), and let $X \leftarrow \{$ data($h$) $\}_{h \in H_{\text{in}}}$
  let bound $\leftarrow \forall h \in H_{\text{in}}\ \exists(\_, \_, C)$ = db[$h$] $\wedge\ \mathbf{c} \in C$
  let owned $\leftarrow \forall p \in P^c_{\text{in}}\ \exists(\_, p', \_)$ = db[$H_{\text{in}}[p]$] $\wedge\ p' = p$
  let compliant $\leftarrow$ $\mathbf{c}.\phi$(ldgr, $H_{\text{in}}$)
  if $\neg$(bound $\wedge$ owned $\wedge$ compliant) then { ret $\bot$ to [$\mathcal{A}$, $p$] }
  let ($s'$, $Y$) $\leftarrow$ $\mathbf{c}.f(s, X; r)$, where $r \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$
  let $h_{s'} \leftarrow$ db.add(($s'$, $\bot$, $\{\mathbf{c}\}$))
  let $H_{\text{out}} \leftarrow \{$ db.add(($y$, $\bot$, $\emptyset$)) $\}_{y \in Y}$
  for $\mathbf{c} \in C^{\mathbf{c}}_{\text{out}}$ { db.update($H_{\text{out}}[\mathbf{c}]$, ($Y[\mathbf{c}]$, $\bot$, $\{\mathbf{c}\}$)) }
  ldgr.append(compute $\|$ $\mathbf{c}$.id $\|$ $h_{s'}$ $\|$ $H_{\text{in}}$ $\|$ $H_{\text{out}}$)
  send $\{|\,y\,|\}_{y \in Y}$ $\|$ $|\,s'\,|$ to $\mathcal{A}$; ret $\top$ to $p$
• On command get_output($\mathbf{c}$.id, $h$) from $p \in P^c_{\text{in}}$
  send (output $\|$ $h$ $\|$ $p$) to $\mathcal{A}$;  send $\top$ to $p$
  if $\exists p \in P^c_{\text{in}}$ that hasn't called get_output or $\mathcal{A}$ denies then { ret }
  send data($h$) to all $p \in P^{\mathbf{c}}_{\text{out}}$;  ldgr.append(output $\|$ $\mathbf{c}$.id $\|$ $h$)
```

$\mathcal{F}_{\text{PCC}}$ maintains a publicly readable log `ldgr` and a private storage db. db provides protected storage of inputs and outputs (including chained outputs) and computational state, and is indexed by unique handles — accesses to db produce $\perp$ if the mapping does not exist. `ldgr` is an append-only log of all function evaluations, creation and revocation of computations, and binding of input handles. Since the specification $\mathbf{c}$ does not contain secrets, it can be created (via create_computation) by any party. A computation can be revoked (via revoke_computation) by any party listed as an input provider in $\mathbf{c}$.`inp`, preventing future evaluations of $\mathbf{c}.f$.

A party p uploads an input x (using provide_input), which $\mathcal{F}_{\text{PCC}}$ persists internally and returns a unique handle h — at this point, x is not bound to any computation. Next, using bind_input, p binds h to a computation $\mathbf{c}$, allowing that input x to be consumed by $\mathbf{c}.f$. From here on, x may be consumed by multiple stateful evaluations of $\mathbf{c}.f$ without p having to resupply x at each step (though each step must comply with $\mathbf{c}.\phi$). Party p may bind x to multiple computations concurrently. We find that the abstraction provided by $\mathcal{F}_{\text{PCC}}$ is useful in settings where parties make dynamic decisions to participate in new computations and become offline after providing their inputs, or when parties compute over large inputs, or in applications that provide a common service to many parties (e.g. Acme).

As only policy-compliant evaluations succeed, we allow any party $p \in P^*$ to invoke a function evaluation (using compute), by providing handles H_{in} referring to the inputs (from both input providers $P^{\mathbf{c}}_{\text{in}}$ and chained computations $C^{\mathbf{c}}_{\text{in}}$). Since `ldgr` records the handle for the state after each evaluation, $\mathcal{F}_{\text{PCC}}$ uses `ldgr` to retrieve the most recent state. Party p can provide any handles of her choice, as $\mathcal{F}_{\text{PCC}}$ checks the guard $\mathbf{c}.\phi$ prior to evaluating $\mathbf{c}.f$, in addition to sanity checks that the inputs are existent and bound to the computation $\mathbf{c}$. Observe that $\mathbf{c}.f$ operates over the inputs, prior state, and a random string r, and produces outputs (bound to output recipients $P^{\mathbf{c}}_{\text{out}}$ and chained computations $C^{\mathbf{c}}_{\text{out}}$) — we create new handles H_{out} for the outputs, and $h_{s'}$ for the next state. Outputs to chained computations are not revealed to any party, and they cannot be bound to other computations (as they are not owned by any party). Before returning, $\mathcal{F}_{\text{PCC}}$ records the evaluation on `ldgr`, along with the relevant handles.

The output recipients invoke get_output for fair output delivery. $\mathcal{A}$ may prevent sending the output; however, should any party get the output, then all parties in $P^{\mathbf{c}}_{\text{out}}$ get the output. Fair reactive computation is out of scope, since $\mathcal{A}$ can deny executing the compute command. That said, LucidiTEE uses a novel protocol enabling a limited form of fair reactive computation: a party is always able to advance a computation that requires input from only that party — this is useful for collaborative machine learning (i.e. after training, any party should be able to use the model for inference), for instance, but does not apply to more general computation such as fair Poker (which requires inputs from multiple parties at each step) as the adversarial parties may go offline. In short, $\mathcal{F}_{\text{PCC}}$ guarantees:

- $\mathcal{A}$ does not learn an honest party's input, beyond its size and the function evaluations which have used that input.
- In any computation $\mathbf{c}$, f is evaluated only if ϕ is satisfied.
- $\mathcal{A}$ learns the outcome of evaluating ϕ, and learns the outcome of f only if it controls a party in $P^{\mathbf{c}}_{\text{out}}$.
- Parties in $P^{\mathbf{c}}_{\text{out}}$ get the correct output with fairness.

4 BUILDING BLOCKS

4.1 Trusted Execution Environment (TEE)

An enclave program is an isolated region of memory, containing both code and data, protected by the TEE platform (where trust is only placed in the processor manufacturer). On TEE platforms such as Intel SGX and Sanctum, the CPU monitors all memory accesses to ensure that non-enclave software (including OS, Hypervisor, and BIOS or SMM firmware) cannot access the enclave's memory — SGX also thwarts hardware attacks on DRAM by encrypting and integrity-protecting the enclave's cache lines. In addition to isolated execution, we assume that the TEE platform provides a primitive for remote attestation. At any time, the enclave software may request a signed message (called a *quote*) binding an enclave-supplied value to that enclave's code identity (i.e., its hash-based measurement). We model the TEE hardware as an ideal functionality HW, adapted from [32]. HW maintains the memory of each enclave program in an internal variable mem, and has the following interface:

- HW.Load(prog) loads the enclave prog code within the TEE-protected region. It returns a unique id eid for the loaded enclave program, and sets the enclave's private memory mem[eid] = $\overrightarrow{0}$.
- HW.Run(eid, in) executes enclave eid (from prior state mem[eid]) under input in, producing an output out while also updating mem[eid]. The command returns the pair (out, quote), where quote is a signature over μ(prog) || out, attesting that out originated from an enclave with hash measurement μ(prog) running on a genuine TEE. We also write the quote as quote_{HW}(prog, out). We assume no additional information leaks to the adversary.
- HW.QuoteVerify(quote) verifies the genuineness of quote and returns another signature σ (such that $\text{Verify}_{\text{HW}}(\sigma, \text{quote}) = \text{true}$) that is publicly verifiable. Any party can check $\text{Verify}_{\text{HW}}$ without invoking the HW functionality. For instance, SGX implements this command using an attestation service, which verifies the CPU-produced quote (in a group signature scheme) and returns a publicly verifiable signature σ over quote || b, where $b \in \{0, 1\}$ denotes the validity of quote; then, any party can verify σ (using Intel's public key) without contacting Intel's attestation service.

4.2 Shared, Append-only Ledger

We borrow the bulletin board abstraction of a shared ledger, defined in [4], which lets parties get its contents and post arbitrary strings on it. Furthermore, on successfully publishing the string on the bulletin board, any party can request a (publicly verifiable) proof that the string was indeed published, and the bulletin board guarantees that the string will never be modified or deleted. Hence, the bulletin board is an abstraction of an append-only ledger. We model the shared ledger as an ideal functionality L, with internal state containing a list of entries, implementing the following interface:

- L.getCurrentCounter returns the current height of the ledger
- L.post(e) appends e to the ledger and returns (σ, t), where t is the new height and σ is the proof that e has been successfully posted to the ledger. Specifically, σ is an authentication tag (also called *proof-of-publication* in prior works [6]) over the pair t||e such that $\text{Verify}_{\text{L}}(\sigma, \text{t}||\text{e}) = \text{true}$ — here, Verify_{L} is a public verification algorithm (e.g., verifying a set of signatures).
- L.getContent(t) returns the ledger entry (σ, e) at height t.

The bulletin board abstraction can be instantiated using fork-less blockchains, such as permissioned blockchains [33], and potentially by blockchains based on proof-of-stake [34].

4.3 Cryptographic Primitives and Assumptions

We assume a hash function H (e.g. SHA-2) that is collision-resistant and pre-image resistant; we also assume a hash-based commitment scheme com with hiding and binding properties.

We use a *IND-CCA2* [35] public key encryption scheme PKE (e.g. RSA-OAEP) consisting of algorithms $\mathsf{PKE.Keygen}(1^\lambda)$, $\mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{m})$, $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$. Moreover, for symmetric key encryption, we use authenticated encryption AE (e.g. AES-GCM) that provides *IND-CCA2* and *INT-CTXT* [36], and it consists of polynomial-time algorithms $\mathsf{AE.Keygen}(1^\lambda)$, $\mathsf{AE.Enc}(\mathsf{k}, \mathsf{m})$, $\mathsf{AE.Dec}(\mathsf{k}, \mathsf{ct})$.

Finally, we use a *EUF-CMA* [37] digital signature scheme S (e.g. ECDSA) consisting of polynomial-time algorithms $\mathsf{S.Keygen}(1^\lambda)$, $\mathsf{S.Sig}(\mathsf{sk}, \mathsf{m})$, $\mathsf{S.Verify}(\mathsf{vk}, \sigma, \mathsf{m})$.

5 ADVERSARY MODEL

The attacker may corrupt any subset of the parties, in a static model where a party is said to be corrupt if it deviates from the protocol at any time. A corrupt party exhibits *byzantine* behavior, which includes aborts, and dropping or tampering any communication with other parties or the ledger. We discuss specific threats below.

TEE Threats

TEE machines can be operated by malicious parties, who can abort the TEE's execution, and delay, tamper, or drop its inputs and outputs (including the communication with the ledger). We assume that the remote attestation scheme is existentially unforgeable under chosen message attacks [32]. Though side channels pose a realistic threat, we consider their defenses to be an orthogonal problem. This assumption is discharged in part by using safer TEEs such as RISC-V Sanctum, which implement defenses for several hardware side channels, and in part by compiling f and ϕ using software defenses (e.g., [38], [39], [40], [41], [42]).

Blockchain Threats

We draw attention to the subtlety of blockchain instantiations. While our fair delivery protocol tolerates an arbitrary corruption threshold of $t < n$, the ledger admits a weaker adversary (e.g. less than 1/3rd corruption in PBFT-based permissioned blockchains, or honest majority of collective compute power in permissionless blockchains). In permissioned settings, this means that the n parties cannot instantiate a shared ledger amongst themselves, and expect to achieve fairness — they need a larger set of participants on the ledger, and require more than 2/3rd of that set to be honest. With that said, [4] also has the same limitation. Fundamentally, forks on proof-of-work blockchains can violate policies, as computation records can be lost. Even the proof-of-publication scheme in Ekiden [6], which uses a trusted timeserver to enforce the rate of production of ledger entries, offers a probabilistic guarantee of rollback prevention, which worsens as the attacker's computational power increases. We restrict our scope to forkless ledgers (providing the bulletin-board abstraction L), such as HyperLedger [33] and Tendermint [43], and even blockchains based on proof-of-stake [34].

6 FAIR OUTPUT DELIVERY

For a computation to be fair, all honest parties must get the output if any party gets the output. We encounter several challenges in designing such a protocol. Foremost, fair delivery is non-trivial in the presence of adversaries, as the physical network cannot ensure simultaneous message delivery to two nodes, giving the attacker an opportunity to prevent one party from getting a message that is delivered to another party — this is often modeled as a *rushing adversary*. Moreover, even a TEE does not address the problem of fair delivery, as the enclave's I/O is controlled by a (malicious) host party. While an append-only ledger ensures persistence of its contents, it alone does not provide fairness, as a corrupt party can bypass the ledger and share secrets with other colluding parties.

We develop a protocol for fair delivery (or fair exchange), which defends against malicious behavior from any subset of the parties. Although fairness is impossible in the standard setting with dishonest majority [16], our protocol makes additional assumptions by using TEE and ledger functionalities, requiring t ($\leq n - 1$) out of n output recipients to possess a TEE machine — in contrast, prior work by Choudhuri et al. [4] requires all n parties to possess a TEE and interact with the ledger. By reducing the TEE requirement over prior work, LucidiTEE enables novel use cases, such as bilateral service relationship between a B2C provider and an end user (who may only have a mobile device), and collaboration between enterprises (where some enterprises may be technically or financially incapable of providing compute). Our protocol also requires constant ledger space overhead (i.e., the size of the ledger contents is independent of the output size), which makes it practical on current blockchain deployments; however, it incurs two writes to the ledger, as opposed to one write in [4].

Inspired by [30, 31], we reduce fair evaluation of any function to fair reconstruction of an additive secret sharing scheme. First, using a functionality for multiparty computation, we produce an output y that is encrypted under a fresh key $\mathsf{k} \doteq \mathsf{k}_{p_1} \oplus \ldots \oplus \mathsf{k}_{p_n}$, where each party $p_i \in P^c_{\text{out}}$ is given k_{p_i} (in addition to commitments to ensure integrity during the protocol). Specifically, each party is given:

$$\begin{aligned} &\mathsf{AE.Enc}\Big(\bigoplus_{p \in P^{\mathsf{c}}_{\text{out}}} \mathsf{k}_p, y\Big) \\ &\quad \| \; \{ \; \mathsf{com}(\mathsf{k}_p; \omega_p) \; \| \; \mathsf{PKE.Enc}(\mathsf{pk}_p, \mathsf{k}_p \; \| \; \omega_p) \; \}_{p \in P^{\mathsf{c}}_{\text{out}}} \end{aligned}$$

Since this step is unfair, any party (and therefore the protocol) can abort if they do not receive this ciphertext — note that fairness is preserved if no party gets the output. Next, we have the parties use their shares and commitments in a protocol for fair reconstruction, such that if any party gets the key k, then so do all honest parties.

The reconstruction protocol works as follows. First, we introduce an enclave program containing the reconstruction logic, called the *reconstruction enclave* $\mathcal{E}_r$ (specified in Appendix B), and we make the t parties with TEE nodes launch a local instance of $\mathcal{E}_r$. Next, each party p_i in P^c_{out} transmits her secret key k_{p_i} to each $\mathcal{E}_r$, thus placing k in each $\mathcal{E}_r$. The protocol aborts if any party does not perform this step. Finally, we need a way to transmit k out of $\mathcal{E}_r$ to all parties in P^c_{out}.

Our key insight is a construct that "simultaneously" allows all parties to attain the key, which is accomplished by posting on the ledger L a value that serves a dual purpose: 1) it contains the encryption of k under the public keys of the non-TEE parties $\{p_{t+1}, \ldots, p_n\}$, and 2) it triggers the t $\mathcal{E}_r$ enclaves to release k to their local parties $\{p_1, \ldots, p_t\}$. Similar to [4], our use of TEE is akin to the cryptographic primitive of witness encryption, where a value is decrypted only upon a proof of publication on the ledger — the proof is verified within the enclave for computational integrity. As we show below, our protocol is designed to avoid cases where we have two corrupt parties, one with and one without TEE, who can abort and collude to get k in a way that leaves honest parties defenseless. Figure 2 illustrates our protocol for Acme's example.

6.1 Two-party Fair Reconstruction

We start with a simpler protocol (i.e., with fewer posts to the ledger) for the 2-party setting, where corruption threshold $t \leq 1$. The simplicity is valuable as bilateral relationships are common in practice — examples include services provided by B2C companies to their end users, data-sharing collaboration between two enterprises, etc.

Consider parties p_i and p_j, where p_i hosts a TEE machine running the $\mathcal{E}_r$ enclave. First, party p_j sends a signature σ_j (using secret key sk_{p_j}) confirming her receipt of the encrypted output — either party aborts the protocol if they do not receive the encrypted output from the compute provider. Party p_i provides σ_j to its local instance of the $\mathcal{E}_r$ enclave, causing it to return the output-encrypting key k encrypted under p_j's public key (i.e., $\mathsf{PKE.Enc}(\mathsf{pk}_{p_j}, \mathsf{k})$). A message m containing this encrypted key is posted by p_i on the ledger (which returns a proof σ_L on a successful post), allowing p_j to attain the key k from the ledger. Moreover, p_i provides the proof σ_L to its enclave, who asserts $\mathsf{Verify_L}(\sigma_\mathsf{L}, \mathsf{m})$ and returns k to its host p_i.

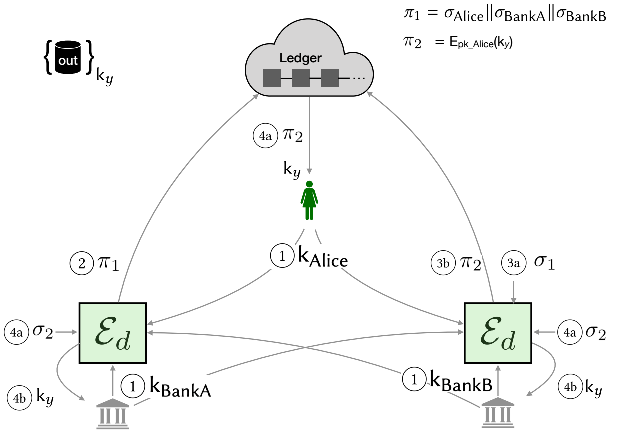
6.2 Multi-party Fair Reconstruction

A trivial generalization of the 2-party protocol to n parties does not work, as the TEE party, instead of posting to the ledger, sends the ledger-bound message to a colluding non-TEE party who can decrypt to attain the output's key. For instance, consider extending the protocol to 3 parties p_i, p_j, and p_k, where p_i hosts a TEE machine; instead of posting a message with $\mathsf{PKE.Enc}(\mathsf{pk}_{p_j}, \mathsf{k}) \parallel \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{p_k}},$ k) to the ledger, p_i forwards it to p_k, leaving p_j cheated. The following secure protocol defends against such collusions.

① *Signature for Receipt of Encrypted Output.* As a first step, we must ensure that all n output recipients have received the encrypted output $\mathsf{AE.Enc}(\mathsf{k}, y)$, as the key k is useless without the ciphertext. We must also ensure that the t parties with TEEs have received all key shares $\{\mathsf{k}_{p_1}, \ldots, \mathsf{k}_{p_n}\}$ within their local instance of $\mathcal{E}_r$. Parties also send their openings to the commitments to the $\mathcal{E}_r$ instances, to aid the correctness verification of the key shares — both the key shares and openings are placed in enclave memory. To that effect, each party sends a signature to other parties confirming their receipt of the expected values — a signature over the output's hashes H_out is sufficient. This step constitutes two rounds:

$$p_i \rightarrow p_j : \mathsf{quote_{HW}}(\mathcal{E}_r, \mathtt{c.id} \parallel \mathsf{pk}), \text{ where } \mathsf{pk} \leftarrow \mathsf{PKE.Keygen}$$
$$p_j \rightarrow p_i : \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{S.Sig}(\mathsf{sk}_{p_j}, H_\text{out}) \parallel \mathsf{k}_{p_j} \parallel \omega_{p_j})$$

② *Posting Signatures on the Ledger.* All n signatures are required to ensure that all parties can decrypt the output once the protocol terminates with fair delivery of the key k. However, a corrupt party can choose to not send his signature to another party. We use the shared ledger to broadcast the n signatures to all parties. Any party can collect signatures from step ① and post them on the ledger.



$\pi_1 = \sigma_\mathsf{Alice} \| \sigma_\mathsf{BankA} \| \sigma_\mathsf{BankB}$
$\pi_2 = \mathsf{E}_\mathsf{pk\_Alice}(\mathsf{k}_y)$
$\{\text{out}\}_{\mathsf{k}_y}$
Ledger
4a π_2
k_y
1 $\mathsf{k}_\mathsf{Alice}$
2 π_1
3b π_2
3a σ_1
4a σ_2
$\mathcal{E}_d$
$\mathcal{E}_d$
4a σ_2
4b k_y
1 $\mathsf{k}_\mathsf{BankA}$
1 $\mathsf{k}_\mathsf{BankB}$
4b k_y

Figure 2: Fair reconstruction using TEEs and a shared ledger

$$p \ : \ \mathsf{L.post}(\pi_1), \text{ which returns } (\sigma_1, \_)$$
$$\pi_1 \doteq \{\ \mathsf{S.Sig}(\mathsf{sk}_{p_i}, \mathtt{c.id} \parallel \mathsf{H_{out}})\ \}_{i \in [n]}$$

The n signatures are computed on the same message, and can be aggregated (using an enclave or a scheme such as [44]).

③ *Posting Encryption of Key* k *on Ledger.* Any of the t parties with a TEE, on seeing π_1 on the ledger, can advance the protocol to the next phase using the proof σ_1 (produced by L upon posting π_1). On providing σ_1 as input, $\mathcal{E}_r$ emits π_2, containing an encryption of k under the public keys of $n - t$ parties $\{p_{t+1}, \ldots, p_n\}$. That party then posts the ciphertext on L.

$$p \in \{p_1, \ldots, p_t\} \ : \ \mathsf{L.post}(\pi_2), \text{ which returns } (\sigma_2, \_)$$
$$\pi_2 \doteq \mathsf{quote_{HW}}(\mathcal{E}_r, \mathtt{c.id} \parallel H_\text{out} \parallel \{\mathsf{PKE.Enc}(\mathsf{pk}_{p_i}, \mathsf{k})\}_{i \in [t+1 \ldots n]})$$

④ *Decrypting the Output.* Each of the n parties can now attain k to decrypt the output. The t parties with $\mathcal{E}_r$ provide the proof σ_2 (produced by L upon posting π_2, or by invoking L.getContent) to their local $\mathcal{E}_r$ enclaves, allowing those enclaves to emit k. The $n - t$ parties without TEE simply retrieve π_2 from L, and decrypt using their private key sk_{p_i} to attain k.

Correctness Argument. The adversary may corrupt upto t parties, giving us 3 cases: 1) all corrupt parties have a TEE; 2) all corrupt parties do not have a TEE; 3) corrupt parties include both TEE and non-TEE parties. In the first two cases, the mechanism that the attacker uses to get the secret key also allows the honest parties to extract the key. In the third case, collusions do help the attacker learn secret key before the honest parties, but there exists at least one honest party with a TEE who can carry out the steps, and distribute the key to all honest parties. We formalize and prove the protocol in the $\mathcal{G}_\text{att}$-hybrid model [4, 24] within Appendix C.

7 POLICY-COMPLIANT COMPUTATION

We gave a protocol for fair output delivery in § 6, which implements the get_output interface of $\mathcal{F}_\text{PCC}$. In this section, we describe how

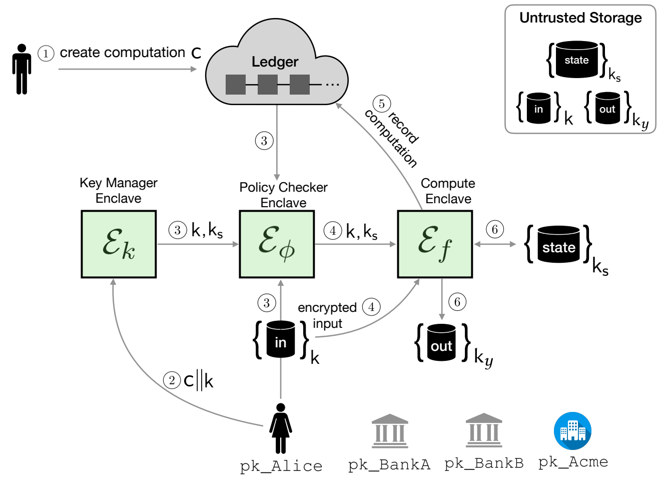


Figure 3: Policy enforcement using TEEs and a shared ledger

computations are setup, how inputs are bound to computations, and how policies are evaluated during the computation.

Figure 3 illustrates the primary components of LucidiTEE. Each entry on the shared ledger records either the creation or revocation of a computation (along with its specification), or a function evaluation containing hashes of inputs, outputs, and state. We stress that the ledger does not store the inputs or state, and its entries only help us to enforce policies. Computation involves three types of enclave programs: 1) a *key manager enclave* $\mathcal{E}_k$ (responsible for handling keys that protect the computation's state and the offline users' input); 2) a *policy checker enclave* $\mathcal{E}_\phi$ (responsible for checking whether a requested evaluation f is compliant with the policy ϕ); 3) a *compute enclave* $\mathcal{E}_f$ (responsible for evaluating f). These enclaves are run on one or more physical TEE machines, managed by any untrusted party — hereon called the *compute provider* p_c — yet our protocols guarantee policy compliance and fairness to all parties (who may also act maliciously). Computation happens off-chain, and is decoupled from the ledger's consensus mechanism.

7.1 Specifying and Creating a Computation

A computation's specification has the following grammar:

Name	n	::=	[a − zA − Z0 − 9]+
Party	p	::=	$\{0,1\}^*$
Input	i	::=	(n : p) | (n : $\rho^?$) | (n : (z, n))
Output	o	::=	(n : [p, . . . , p]) | (n : (z, n))
Hash	h	::=	$\{0,1\}^*$
Comp	c	::=	computation { id : $\{0,1\}^*$, policy : h, func : h, inp : [i, . . . , i], out : [o, . . . , o] }

A computation is identified by a unique 64-bit number z. Each input and output data structure is named by an alphanumeric constant n. Each party p is cryptographically identified by their public key material (e.g. RSA or ECDSA public key), which is a finite length binary string. An input is denoted by the tuple (n : p), containing its name n and the cryptographic identity of the input provider p — if the set of input providers is unknown at the time of specifying the computation (e.g. surveys), we write (n : $\rho^?$). Similarly, an output is denoted by the tuple (n : [p, . . . , p]), containing its name n along with the list of all output recipients. An output may also be fed as an input to a different future computation. We call this *compute chaining*, and write it as (n : (z, n)), where z is the destination computation's identifier and the latter n is the input's name in the destination computation. We use a hash h to encode the expected measurement of enclaves containing the code of f and ϕ. Finally, we combine these fields to specify a computation **c**: an id, hash of $\mathcal{E}_f$ (func) that implements f, hash of $\mathcal{E}_\phi$ (policy) that implements ϕ, input description inp, and output description out.

A party p can create a new multi-party computation, as specified by the string **c**, using the create_computation(**c**) command in $\mathcal{F}_{\text{PCC}}$, which LucidiTEE implements by having p execute:

$$p \rightarrow \mathcal{E}_k : \mathbf{c} \parallel \sigma, \text{ where } (\sigma, \mathsf{t}) = \mathsf{L.post}(\mathtt{create} \parallel \mathbf{c})$$

Having posted the specification **c**, p contacts the compute provider p_c, who forwards the request to its local instance of the key manager enclave $\mathcal{E}_k$. $\mathcal{E}_k$ generates a key k_s, used to protect the computation's state across all function evaluations, using the TEE's entropy source (such as rdrand on SGX). Since **c** does not contain any secrets, any party can post it on the ledger, and it is up to the input providers to examine **c** and choose to bind their inputs to it.

Any input provider $p \in P^{\mathbf{c}}_{\text{in}}$ can revoke a computation by invoking revoke_computation(**c**.id), which LucidiTEE implements by having p execute L.post(revoke ‖ **c**.id).

7.2 Providing Inputs

A party submits data to a computation by invoking provide_input(x), which returns a unique handle to the input data. $\mathcal{F}_{\text{PCC}}$ maintains privacy of that data, and guarantees that the data is unmodified when later used by a computation. To that end, LucidiTEE's implementation must store the data on an untrusted storage accessible by the compute provider, while also protecting the data — the attacker can always delete the data, but that is akin to denying compute.

The cryptographic protection must not only ensure confidentiality, integrity, and authenticity of the data, but also cryptographically bind it to its unique handle. To that end, the input provider chooses a random key k, and computes AE.Enc(k, x) — the encryption protects inputs that are derived from low-entropy distributions. We derive the handle h by computing a cryptographic digest over the ciphertext, using hash functions or accumulators such as a Merkle tree, and return h to the calling party. Accumulators enable membership queries, which we later show to be useful when enforcing policies such as one-time programs and accounting of all votes.

7.3 Binding Inputs to Computations

Recall the bind_input(**c**.id, h) command in $\mathcal{F}_{\text{PCC}}$, which makes the user-provided input x (referred by h) accessible by computation **c**. We have two requirements: 1) x must only be used for evaluating **c**.f, for each computation **c** that the user has bound x to; and, 2) the input provider must be able to commit to the value of x, in order to prevent any tampering by the compute provider.

By encrypting the input with user-chosen key k, LucidiTEE reduces this problem to ensuring that the key k is only provisioned to enclaves identified within the computation's specification. Binding the use of key k to a computation **c** is carried out via a protocol between the input provider p, ledger L, and the compute provider

p_c (who is running an instance of the key manager enclave $\mathcal{E}_k$):

$$p : \mathsf{L.post}(\ \mathtt{bind} \parallel \mathbf{c}.\mathtt{id} \parallel h \parallel \mathsf{S.Sig}(\mathsf{sk}_p, \mathbf{c}.\mathtt{id} \parallel h)\ )$$
$$p_c \to p : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_k, \mathsf{pk}), \text{where } \mathsf{pk} \leftarrow \mathsf{PKE.Keygen}$$
$$p \to p_c : \mathsf{PKE.Enc}(\mathsf{pk}, \mathbf{c}.\mathtt{id} \parallel \mathsf{k} \parallel \mathsf{n} \parallel \mathsf{S.Sig}(\mathsf{sk}_p, \mathbf{c}.\mathtt{id}\|\mathsf{k}\|\mathsf{n}))$$

First, p creates a ledger entry binding the data handle h (returned by bind_input) to computation **c**. Next, p contacts p_c, whose instance of $\mathcal{E}_k$ produces a fresh public key pk along with a quote attesting to the genuineness of $\mathcal{E}_k$. Upon verifying the attestation quote, p (with signing key $\mathsf{sk_p}$) signs and encrypts **c**.id and k, along with n which specifies the name of the input (from the list **c**.inp). $\mathcal{E}_k$ will later reveal the key k only to that enclave which is evaluating $\mathbf{c}.f$.

By binding handles to computations, we reuse inputs across function evaluations and computations, without having to clone the data or require users to store a local copy of the data.

7.4 Enforcing Policy-Compliance

Any party $p_c \in P^*$ can act as a compute provider, and invoke compute(**c**.id, H_{in}) on chosen inputs (referenced by handles H_{in}). Hence, we implement a protocol to ensure policy compliance even when other parties are offline and p_c acts maliciously, as follows.

Before evaluating f, p_c must first launch $\mathcal{E}_\phi$ to evaluate ϕ. Next, $\mathcal{E}_\phi$ must check whether the requested evaluation of f is compliant with ϕ, which requires checking three conditions (recall from $\mathcal{F}_{\mathsf{PCC}}$):

(1) active: **c** is created on the ledger, and not yet revoked
(2) bound: data for each $h \in H_{\mathsf{in}}$ is bound to computation **c**
(3) compliant: predicate ϕ(ledger, H_{in}) over ledger's contents

To perform these checks, p_c must provide $\mathcal{E}_\phi$ with a read-only view of L, by downloading the ledger's contents locally, in which case the enclave-ledger interaction is mediated by the host software controlled by p_c. Although using $\mathsf{Verify_L}$ allows $\mathcal{E}_\phi$ to detect arbitrary tampering of L's contents, an adversarial p_c may still present a stale view (i.e., a prefix) of L to $\mathcal{E}_\phi$. We mitigate this attack in § 7.6. For now, we task ourselves with deciding compliance with respect to a certain (potentially stale) view or height of L.

The policy ϕ is an arbitrary predicate. As an example, consider the policy ϕ from the Acme application: transactions in the input must not have been consumed by a prior evaluation of $\mathbf{c}.f$. Cryptographic accumulators, such as Merkle trees, play an important role in efficiently evaluating ϕ, as they support efficient membership queries — recall from § 7.2 that we use such accumulators to compute digests of encrypted inputs and outputs that are recorded on the ledger. In Acme's case, we scan the ledger to construct an accumulator over the input handles from all prior evaluations of $\mathbf{c}.f$, and check absence of each input transaction within the accumulator.

7.4.1 Performance Optimizations. It is not practical to process the entire ledger for each evaluation of $\mathbf{c}.\phi$. Specifically, if the ledger L is naively stored as a sequence of entries, it would force us to perform a linear scan for evaluating the three compliance checks.

Instead, our implementation stores L locally as an authenticated key-value database [45], whose index is the computation's id **c**.id. Each computation appends the ledger entry to the current value at **c**.id. Now, instead of scanning through the entire ledger, the first compliance check asserts the presence of key **c**.id, while the second check queries the list of records at key **c**.id. Finally, to evaluate ϕ, we maintain an accumulator as state, and update it on each entry of **c**.id — we let ϕ rely on an enclave that persists state, in the form of an authenticated key-value store [45], across several evaluations of ϕ. Note that this optimization does not impact security, as $\mathcal{E}_\phi$'s view of L is still controlled by p_c, and therefore potentially stale.

Consider other history-based policies used in apps from § 9.1. In the survey app (§ 9.1.2), we check that the input includes *all* votes for which we find commitments on the ledger (produced by the user's invocation of bind_input); accumulators again suffice for this policy. Both the machine learning app (§ 9.1.3) and PSI apps (§ 9.1.4) compare handles (i.e., equality check on hashes) for their policies.

7.5 Producing Encrypted Output

The compute provider launches the compute enclave $\mathcal{E}_f$, who then asks $\mathcal{E}_k$ for the keys to **c**'s state and all parties' input. $\mathcal{E}_k$ transmits these keys upon verifying that $\mathcal{E}_f$ has the correct hash **c**.func.

The computation can be performed using any enclave-based data processing system, such as Opaque [9], Ryoan [11], etc. A randomized f needs an entropy source. Recall that $\mathcal{E}_k$ generated a key k_s to protect the computation's state. Using this key k_s, f can internally seed a pseudo-random generator (e.g. PRF with key $\mathsf{H}(\mathsf{t} \parallel \mathsf{k_s})$) to get a fresh pseudo-random bitstream at each step of the computation. (Note that t uniquely identifies the step.) This ensures that the random bits are private to $\mathcal{E}_f$, yet allows replaying computation from the ledger during crash recovery, for instance.

7.6 Recording Computation on Ledger

Recall that $\mathcal{E}_\phi$ checked compliance of ϕ with respect to a certain height of the ledger. Since a malicious compute provider p_c can present a stale view of the ledger to $\mathcal{E}_\phi$, we discuss how we defend against this attack at the time we record the computation on the ledger, which is a precursor to extracting the output. (Recall that the ledger record is also used to enforce history-based policies.)

The protocol works as follows. We first request the instance of $\mathcal{E}_\phi$ (from § 7.4) for the ledger height t and input handles H_{in} with which it evaluated $\mathbf{c}.\phi$; $\mathcal{E}_\phi$ sends these values as part of a quote.

$$\mathcal{E}_\phi \to \mathcal{E}_f : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_\phi, \mathsf{t} \parallel H_{\mathsf{in}})$$

Next, to extract the encrypted output, the compute provider p_c must record $\mathcal{E}_f$'s execution on L by posting the following message:

$$\mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_f, \mathtt{compute}\|\mathbf{c}.\mathtt{id} \parallel h_{s'} \parallel H_{\mathsf{out}}) \parallel \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_\phi, \mathsf{t} \parallel H_{\mathsf{in}})$$

The use of $\mathsf{quote}_{\mathsf{HW}}$ ensures that the computation evaluated the function $\mathbf{c}.f$ within a genuine TEE, consuming inputs with handles H_{in}, and producing outputs with handles H_{out} and next state with handle $h_{s'}$. Moreover, we append the quote returned by $\mathcal{E}_\phi$.

However, by the time L receives the post command, it may have advanced by several entries from t to t′. This can be caused by a combination of reasons including: 1) ledger entries from concurrent evaluations of c and other computations on LucidiTEE; and, 2) malicious p_c providing a stale view of L to $\mathcal{E}_\phi$. This potentially invalidates $\mathcal{E}_\phi$'s check, but instead of rejecting the computation, which would unnecessarily limit concurrency during honest behavior, we assert a *validity predicate* on the ledger's contents:

$$\forall \mathsf{t}, \mathsf{t}', \mathsf{t}^*.\ \mathsf{t}' = \mathsf{L.getCurrentCounter} \land \mathsf{t} < \mathsf{t}^* < \mathsf{t}' \Rightarrow$$
$$\neg((\sigma, \mathsf{e}) = \mathsf{L.getContent}(\mathsf{t}^*)\ \land\ \mathsf{Verify_L}(\sigma, \mathsf{t}^*\|\mathsf{e}) \land$$
$$(\exists a \in \{\mathtt{compute}, \mathtt{bind}, \mathtt{revoke}\}.\ \mathsf{e} = a \parallel \mathbf{c}.\mathtt{id} \parallel \ldots))$$

Here, we check that the computation **c** is still active and that no new function evaluation or bind for `c.id` is performed in between t and the current height t′. The computation is rejected if the check fails, and no entry is recorded on L. This validity predicate may be checked by the ledger's participants before appending any entry, but that would be outside the scope of the bulletin-board abstraction; instead, we rely on our trusted enclaves, $\mathcal{E}_\phi$ and $\mathcal{E}_f$ (and $\mathcal{E}_r$ from § 6) to assert the validity predicate, and abort any further computation or protocol execution atop an invalid ledger.

8 IMPLEMENTATION

We implement LucidiTEE with a heavy focus on modularity and minimality of the trusted computing base. If the ledger is naively stored as a sequence of entries, it would force us to perform a linear scan for evaluating policy compliance. Instead, our implementation stores the ledger locally as an authenticated key-value database [45], whose index is the computation's id. We instantiate the shared ledger with a permissioned blockchain, and evaluate using both Hyperledger [33] and Tendermint [43]. The ledger participant's logic is implemented as a smart contract (in 200 lines of Go code), which internally uses RocksDB [46].

To help developers write enclave-hosted applications (specifically, the compute enclave $\mathcal{E}_f$ and policy checker enclave $\mathcal{E}_\phi$ for each application), we developed an enclave programming library libmoat, providing a narrow POSIX-style interface for commonly used services such as file system, key-value databases, and channel establishment with other enclaves. libmoat is statically linked with application-specific enclave code, ϕ and f, which together form the enclaves, $\mathcal{E}_\phi$ and $\mathcal{E}_f$ respectively — note that the developer is free to choose any other library which respects LucidiTEE's protocol for interacting with the shared ledger L, and enclaves $\mathcal{E}_k$ and $\mathcal{E}_r$. libmoat transparently encrypts and authenticates all operations to the files and databases, using the scheme from § 7.2 — it uses the keys provisioned by the key manager enclave $\mathcal{E}_k$ for encryption, and implements authenticated data structures (e.g. Merkle tries) to authenticate all operations. LucidiTEE provides fixed implementations of $\mathcal{E}_r$ and $\mathcal{E}_k$, whose measurements are hard-coded within libmoat. Furthermore, libmoat implements the ledger interface L, which verifies signatures ($\mathsf{Verify}_\mathsf{L}$) and TEE attestation of ledger entries. libmoat contains 3K LOC, in addition to Intel's SGX SDK [47].

9 EVALUATION

9.1 Case Studies

We demonstrate applications which demonstrate novel history-based policies, and require fairness of output delivery. In addition, we build micro-benchmarks (not described here for space reasons) such as one-time programs [2], digital lockbox with limited passcode attempts, and 2-party fair information exchange.

9.1.1 ***Personal Finance Application.*** We implement Acme's personal finance application, which Alice uses to generate a monthly report. The application uses a history-based policy that transaction records are fresh, i.e., they are not used in a prior evaluation of Acme's function (which would otherwise violate Alice's privacy). Acme's input is encoded as a key-value database indexed by the merchant id — with over 50 million merchants worldwide, this database can grow to a size of several GBs (we use a synthetic database of 50 million records, totaling 1.6 GB in size). We also implemented a client that uses the OFX API [27] to download the user's transactions from their bank, and encrypt and upload the file to a public AWS S3 storage. This encrypted file is later fetched to perform the computation, by decrypting the contents within enclave memory.

9.1.2 ***Private Survey.*** Acme would like to conduct a privacy-preserving survey, such that Acme only learns the aggregate summary of the survey rather than individual responses. However, to maintain the integrity of the survey, we use a history-based policy consisting of two predicates. First, the survey is open only to users of their personal finance application (from § 9.1.1), as opposed to allowing arbitrary input providers to provide fake reviews — specifically, the user (identified by her public key) that participates in the survey must have a ledger entry of type bind_input, destined for Acme's personal finance application above. Second, the survey's result must aggregate all submitted votes, until Acme closes the survey with its own bind_input ledger entry. The policy function is evaluated on the contents of the ledger, using authenticated data structures that support membership queries, as described in **??**.

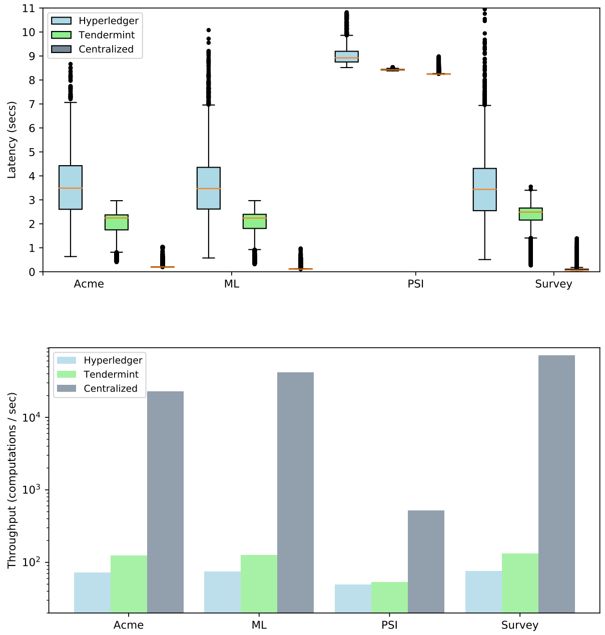
9.1.3 ***Federated Machine Learning.*** A hospital sets up a service for any user to avail the prediction of a model (specifically the ECG class of a patient), in exchange for submitting their data for use in subsequent retraining of the model — we require a fair exchange of user's ECG data and the model's output, which our protocol (§ 6) achieves while only requiring the hospital to provision a TEE node. The service is split into two chained computations: training and inference. Retraining happens on successive batches of new users' data, so when a user submits their ECG data, they wish to use the output model from the latest evaluation of the retraining function — this acts as our history-based policy. For the experiment, we use the UCI Machine Learning Repository [48], and the side-channel resistant k-means clustering algorithm from [49].

9.1.4 ***One-time Private Set Intersection.*** Two hospitals share prescription records about their common patients using private set intersection. Moreover, they require a guarantee of fair output delivery, and use a one-time program policy to prevent data misuse. We implement oblivious set intersection by adapting Signal's private contact discovery service [50]. Our experiment uses a synthetic dataset with 1 million records for each hospital (totalling 15GB).

9.2 Performance Measurement

We study the performance of our applications, and compare to a baseline version where the application runs without a ledger, and without our policy compliance and fairness protocols. The baseline versions of Acme, survey, ML, and PSI apps take 0.02, 0.41, 0.006, and 8.24 seconds, respectively, for each function evaluation of f (including ϕ), using the aforementioned inputs for each application.

9.2.1 End-to-end Latency and Throughput. Figure 4 reports the latency and throughput (results aggregated over 100 runs) on both HyperLedger [33] and Tendermint [43] ledgers (running with 4 peers), with 500 enclave clients concurrently querying and posting ledger entries — we use a 4 core CPU to run the ledger, and a cluster with 56 CPU cores to run the enclaves. We measure end-to-end latency, from launching $\mathcal{E}_\phi$ to terminating $\mathcal{E}_r$. Recall that each evaluation on LucidiTEE performs at least one read query (often



Application	Ledger	Input	Output	State
Acme Finance	2155 B	1.6 GB	1872 B	136 B
Federated ML	1835 B	132 KB	1088 B	-
Policy-based PSI	1835 B	30 MB	8 MB	-
Private Survey	100.1 MB	954.4 MB	2 KB	-

Figure 4: Latency, Throughput, and Storage Overheads

more in order to evaluate ϕ) and two writes (to record the `compute` and `deliver` entry) to the ledger. We found throughput to be bound by the performance of the ledger, which was highly dependent on parameters such as the batch size and batch timeout [33], with the exception of the PSI application which was compute bound (as each function evaluation took roughly 8.2 secs). The latency also suffered by several seconds, as the ledger faced a high volume of concurrent read and write requests. We also evaluate on a "centralized" ledger, implemented as trusted key-value store, thus demonstrating performance with an ideally-performant ledger.

9.2.2 Storage. Figure 4 shows the off-chain and on-chain ledger storage cost on each function evaluation; the ledger storage includes entries due to bind_input, get_output, and compute. Observe that the survey amongst 1 million participants incurred 1 million calls to bind_input, incurring a high on-chain storage cost. In other applications, inputs are orders of magnitude larger than the ledger storage. Since, Ekiden [6] stores inputs and state on the ledger, LucidiTEE has orders of magnitude improvement in ledger storage.

10 RELATED WORK

TEEs, such as Intel SGX, are finding use in systems for outsourced computing, such as M2R [51], VC3 [8], Ryoan [11], Opaque [9], EnclaveDB [52], etc. Felsen et al. [12] use TEEs for secure function evaluation in the multi-party setting. We find these systems to be complementary, in that they can be used to compute over encrypted data within enclaves, while LucidiTEE handles policies and fairness.

ROTE [19], Ariadne [53], Memoir [54], and Brandenburger et al. [55] address rollback attacks on TEEs. Similarly, Kaptchuk et al. [28] address rollback attacks by augmenting enclaves with shared ledgers. We extend their ideas to general history-based policies.

Ekiden [6], FastKitten [25], CCF [56], and Private Data Objects (PDO) [20] are closest to our work. FastKitten [25] provides fair distribution of coins in multi-round contracts such as poker and lotteries. Ekiden and PDO execute smart contracts within SGX enclaves, connected to a blockchain for persisting the contract's state. CCF provides a framework (based on executing smart contracts within SGX) for enterprises to deploy decentralized applications for their consumers. To our knowledge, none of these systems ([6], [25], [57], [20]) provide complete fairness [4] or the expressivity of history-based policies — FastKitten only provides financial fairness. On the practical front, LucidiTEE improves efficiency by not placing inputs or state on the ledger, which is used only to enforce policies, and therefore scales with the size of their inputs. In addition to the performance improvements, the ideal functionalities of Ekiden and LucidiTEE differ in: 1) history-based policies are expressed over the entire ledger, spanning multiple computations, whereas Ekiden only supports contract-specific state; 2) Ekiden's attacker can prevent sending the output to any party, while we ensure fairness.

Hawk [58] and Zexe [59] enable parties to perform off-chain computation with privacy, while proving correctness by posting zero knowledge proofs on the ledger. As mentioned in [6], they support limited types of computation, and Hawk only provides financial fairness. On that note, several works prior to Hawk, specifically [60, 61], [62], [63], and [64], use Bitcoin [65] to ensure financial fairness in MPC applications. Goyal et al. [66] show how blockchains can implement one time programs using cryptographic obfuscation.

MPC [67] [13] [68] protocols implement a secure computation functionality, but require parties to be online or trust one or more third parties. Choudhuri et al. [4] proposed a fair MPC protocol based on witness encryption (instantiated using a TEE at each of the n participants) and a shared ledger. We improve their protocol by requiring t (corruption threshold) out of n output recipients to possess TEE machines. Moreover, [4] requires all parties to be online, and only considers one-shot MPC as opposed to stateful computation with policies. Pass et al. [24] develop a protocol for 2-party fair exchange in the Δ-fairness model.

11 CONCLUSION

We defined an ideal functionality $\mathcal{F}_{\text{PCC}}$ for policy-based, fair multiparty computation, and built LucidiTEE, a TEE-blockchain system that implements $\mathcal{F}_{\text{PCC}}$. We show that LucidiTEE scales to big data applications and large number of users who need not be online.

REFERENCES

[1] "Mint," https://www.mint.com.
[2] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "One-time programs," in *Advances in Cryptology – CRYPTO 2008*, D. Wagner, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39–56.
[3] S. D. Gordon, Y. Ishai, T. Moran, R. Ostrovsky, and A. Sahai, "On complete primitives for fairness," in *TCC*, 2010, pp. 91–108.
[4] A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers, "Fairness in an unfair world: Fair multiparty computation from public bulletin boards," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications*

Security, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 719–728.

[5] G. Kaptchuk, I. Miers, and M. Green, "Giving state to the stateless: Augmenting trustworthy computation with ledgers," 2019.

[6] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution," *CoRR*, vol. abs/1804.05141, 2018.

[7] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS*, 2001, pp. 136–145.

[8] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: trustworthy data analytics in the cloud using SGX," in *Proc. IEEE Symposium on Security and Privacy*, 2015.

[9] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. Berkeley, CA: USENIX Association, 2017, pp. 283–298.

[10] A. Gribov, D. Vinayagamurthy, and S. Gorbunov, "Stealthdb: a scalable encrypted database with full sql query support," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 370–388, 2019.

[11] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 533–549.

[12] S. Felsen, Á. Kiss, T. Schneider, and C. Weinert, "Secure and private function evaluation with intel sgx," 2019.

[13] A. C.-C. Yao, "How to generate and exchange secrets," in *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, ser. SFCS '86. Washington, DC, USA: IEEE Computer Society, 1986, pp. 162–167.

[14] S. Goldwasser, S. D. Gordon, V. Goyal, A. Jain, J. Katz, F.-H. Liu, A. Sahai, E. Shi, and H.-S. Zhou, "Multi-input functional encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014, pp. 578–602.

[15] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *Stoc*, vol. 9, no. 2009, 2009, pp. 169–178.

[16] R. Cleve, "Limits on the security of coin flips when half the processors are faulty," in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '86. New York, NY, USA: ACM, 1986, pp. 364–369.

[17] A. Beimel, A. Gabizon, Y. Ishai, E. Kushilevitz, S. Meldgaard, and A. Paskin-Cherniavsky, "Non-interactive secure multiparty computation," in *Annual Cryptology Conference*. Springer, 2014, pp. 387–404.

[18] A. Beimel, Y. Ishai, and E. Kushilevitz, "Ad hoc psm protocols: secure computation without coordination," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 580–608.

[19] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1289–1306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[20] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private data objects: an overview," *arXiv preprint arXiv:1807.05686*, 2018.

[21] B. Pinkas, "Fair secure two-party computation," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2003, pp. 87–105.

[22] S. D. Gordon and J. Katz, "Partial fairness in secure two-party computation," in *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, 2010, pp. 157–176.

[23] A. Beimel, Y. Lindell, E. Omri, and I. Orlov, "1/*p*-secure multiparty computation without honest majority and the best of both worlds," in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, 2011, pp. 277–296.

[24] R. Pass, E. Shi, and F. Tramer, "Formal abstractions for attested execution secure processors," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 260–289.

[25] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: Practical smart contracts on bitcoin," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 801–818. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/das

[26] E. Brickell and J. Li, "Enhanced privacy id from bilinear pairing," Cryptology ePrint Archive, Report 2009/095, 2009.

[27] [Online]. Available: https://developer.ofx.com/

[28] G. Kaptchuk, M. Green, and I. Miers, "Giving state to the stateless: Augmenting trustworthy computation with ledgers," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[29] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[30] S. D. Gordon, Y. Ishai, T. Moran, R. Ostrovsky, and A. Sahai, "On complete primitives for fairness," in *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, 2010, pp. 91–108.

[31] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan, "Improvements to secure computation with penalties," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 406–417.

[32] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional encryption using intel sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 765–782.

[33] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15.

[34] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 51–68.

[35] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation," Cryptology ePrint Archive, Report 2017/604, 2017, https://eprint.iacr.org/2017/604.

[36] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," *J. Cryptol.*, vol. 21, no. 4, pp. 469–491, Sep. 2008.

[37] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM J. Comput.*, vol. 17, no. 2, pp. 281–308, Apr. 1988.

[38] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 431–446.

[39] R. Sinha, S. Rajamani, and S. A. Seshia, "A compiler and verifier for page access oblivious computation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 649–660.

[40] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 317–328.

[41] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 563–574.

[42] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," 2017.

[43] "Tendermint core in go," https://github.com/tendermint/tendermint.

[44] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[45] R. Sinha and M. Christodorescu, "Veritasdb: High throughput key-value store with integrity," Cryptology ePrint Archive, Report 2018/251, 2018, https://eprint.iacr.org/2018/251.

[46] "Rocksdb," https://github.com/facebook/rocksdb.

[47] "Intel sgx for linux," https://github.com/intel/linux-sgx.

[48] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[49] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Securing data analytics on sgx with randomization," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 352–369.

[50] M. Marlinspike, "Private contact discovery for signal." [Online]. Available: https://signal.org/blog/private-contact-discovery/

[51] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M2r: Enabling stronger privacy in mapreduce computation." in *USENIX Security Symposium*, 2015, pp. 447–462.

[52] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *EnclaveDB: A Secure Database using SGX*. IEEE, 2018.

[53] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 875–892. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx

[54] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 379–394. [Online]. Available: https://doi.org/10.1109/SP.2011.38

[55] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 2017, pp. 157–168.

[56] A. Shamis, A. Chamayou, C. Avanessians, C. M. Wintersteiger, E. Ashton, F. Schuster, C. Fournet, J. Maffre, K. Nayak, M. Russinovich, M. Kerner, M. Castro, T. Moscibroda, O. Vrousgou, R. Schwartz, S. Krishna, S. Clebsch, and O. Ohrimenko, "Ccf: A framework for building confidential verifiable replicated services," Microsoft, Tech. Rep. MSR-TR-2019-16, April 2019.

[57] ——, "Ccf: A framework for building confidential verifiable replicated services," Microsoft, Tech. Rep. MSR-TR-2019-16, April 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/ccf-a-framework-for-building-confidential-verifiable-replicated-services/

[58] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP).* IEEE, 2016, pp. 839–858.

[59] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," Cryptology ePrint Archive, Report 2018/962, 2018, https://eprint.iacr.org/2018/962.

[60] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," in *Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE, 2014, pp. 443–458.

[61] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Łukasz Mazurek, "Fair two-party computations via bitcoin deposits," Cryptology ePrint Archive, Report 2013/837, 2013, https://eprint.iacr.org/2013/837.

[62] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *Advances in Cryptology – CRYPTO 2014.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 421–439.

[63] R. Kumaresan and I. Bentov, "How to use bitcoin to incentivize correct computations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2014, pp. 30–41.

[64] A. Kiayias, H.-S. Zhou, and V. Zikas, "Fair and robust multi-party computation using a global transaction ledger," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2016, pp. 705–734.

[65] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[66] R. Goyal and V. Goyal, "Overcoming cryptographic impossibility results using blockchains," in *Theory of Cryptography Conference.* Springer, 2017, pp. 529–561.

[67] A. C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science.* Washington, DC, USA: IEEE Computer Society, 1982, pp. 160–164.

[68] X. Wang, S. Ranellucci, and J. Katz, "Global-scale secure multiparty computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 39–56. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133979

[69] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *TCC*, 2007, pp. 61–85.

A DISCUSSION ON MODELING TEE

Broadly speaking, there are two different approaches to model attested executions. The "theoretical/abstract" variant is via $\mathcal{G}_{\text{att}}$ [4, 24] and the "practical/concrete" variant is via HW [23] algorithms as outlined in the main body of the paper. In the appendix of the paper, we also consider the $\mathcal{G}_{\text{att}}$-hybrid model to formally present and prove our fairness result. The justification for using $\mathcal{G}_{\text{att}}$-hybrid model there is because it provides an "apples-to-apples" comparison with [4, 24]. That is, our theorem statement can be compared with [4, 24] on concrete metrics such as (1) number of enclaves required (t for us, n for [4], (2) type of fairness realized ("Δ-fairness" for [24], perfect fairness for us and [4]), (3) whether a blockchain is required (no dependence for [24]), and if so, number of "on-chain" rounds (1 for [4], and 2 for us).

From a systems perspective, the closest paper to ours would be Ekiden [6]. While they describe their protocol in the $(\mathcal{G}_{\text{att}}, \mathcal{F}_{\text{blockchain}})$-hybrid model, we believe that our objectives differ—we support, say explicit history-based policy checking which is not their focus, and sometimes unachievable when the history dependency is *a priori* unknown. Thus, following their style in the presentation would likely yield a so to speak "apples-to-oranges" comparison. That said, we do follow their style in the UC modeling of our main functionality $\mathcal{F}_{\text{PCC}}$, as we benefit from the resulting succinct description of our objectives. Similarly, we also use the abstraction of $\mathcal{F}_{\text{blockchain}}$ to which we augment the bulletin board abstractions of [4]. In the following, we address and justify another consequence of our modeling choices.

Compute Provider's Protocol $\mathsf{Prot}_{\mathsf{compute}}[\mathsf{c}]$

```
(1) Launch the $\mathcal{E}_k$, $\mathcal{E}_\phi$, and $\mathcal{E}_f$ enclaves.
    let $\mathsf{e_k} \leftarrow \mathsf{HW.Load}(\mathsf{prog_{keymgr}})$
    let $\mathsf{e}_\phi \leftarrow \mathsf{HW.Load}(\mathsf{prog_{policy}}[\mathsf{c}])$
    let $\mathsf{e_f} \leftarrow \mathsf{HW.Load}(\mathsf{prog_{compute}}[\mathsf{c}])$
(2) Send keys from $\mathcal{E}_k$ to $\mathcal{E}_\phi$, and evaluate $\mathsf{c}.\phi$ within $\mathcal{E}_\phi$.
    let $\mathsf{q}_\phi \leftarrow \mathsf{HW.Run}(\mathsf{e}_\phi, (\text{"keygen"}))$
    let $\mathsf{q_k} \leftarrow \mathsf{HW.Run}(\mathsf{e_k}, (\text{"sendkeys"}, \mathsf{q}_\phi))$
    let $\mathsf{ct}_s, \mathsf{ct}_{x_1}, \ldots, \mathsf{ct}_{x_m} \leftarrow$ download from untrusted storage
    let $\mathsf{t} \leftarrow \mathsf{L.getCurrentCounter}$
    let $\mathtt{ldgr} \leftarrow [(\sigma, \pi, \mathsf{t}^*) \mid \mathsf{t}^* < \mathsf{t}, (\sigma, \pi) \leftarrow \mathsf{L.getContent}(\mathsf{t}^*)]$
    assert $\mathsf{HW.Run}(\mathsf{e}_\phi, (\text{"eval"}, \mathsf{q}_k, \mathtt{ldgr}, \mathsf{ct}_s, \mathsf{ct}_{x_1}, \ldots, \mathsf{ct}_{x_m}))$
(3) Send keys from $\mathcal{E}_\phi$ to $\mathcal{E}_f$, and compute $\mathsf{c}.f$ within $\mathcal{E}_f$.
    let $\mathsf{q_f} \leftarrow \mathsf{HW.Run}(\mathsf{e_f}, (\text{"keygen"}))$
    let $\mathsf{q}_\phi \leftarrow \mathsf{HW.Run}(\mathsf{e}_\phi, (\text{"sendkeys"}, \mathsf{q_f}))$
    $(\mathsf{q_f}, \mathsf{ct}_{s'}, \mathsf{ct}_y) \leftarrow \mathsf{HW.Run}(\mathsf{e_f}, (\text{"eval"}, \mathsf{q}_\phi, \mathsf{ct}_s, \mathsf{ct}_{x_1}, \ldots, \mathsf{ct}_{x_m}))$
(4) Send output to parties
    foreach $p \in \mathsf{c.out}$: send $\mathsf{q_f} \parallel \mathsf{ct}_y$ to $p$
    $\mathsf{L.post}(\mathsf{q_f})$
```

Technically speaking, one of the technical foci of [24] is in ensuring *deniability* (of say participation in a certain protocol). This in turn motivates various aspects of [24] given their choice to model TEEs via $\mathcal{G}_{\text{att}}$—a globally available functionality which provides attestations that are publicly verifiable and consequently attestations provided in one protocol may yield a proof of participation of an honest party that may be presented to a third party. This issue surfaces since [24] do not assume that all parties have a secure processor (i.e., in $\mathcal{G}_{\text{att}}$'s registry). (Such an issue is not discussed in [4] as their fairness result assumes that all parties have a secure processor.) Recall that we provide an implementation of a fair protocol that works even when $n - t$ parties (t is the corruption threshold) do not have a secure processor. Thus, in our formal presentation of our fairness protocol we address all the subtleties involved *a la* [24]. On the other hand, one of the goals of LucidiTEE is to provide transparency, and is in some sense orthogonal to the notion of deniability—consider for example an history based policy where participation in a previous protocol is a necessary precondition for participation in a future protocol (e.g., in the survey application discussed earlier). While ideas from systems such as Zexe [59] (i.e., zero-knowledge proofs) may help towards this goal (i.e., addressing the twin notions of policy compliance while allowing deniability), it is not the focus of this work, and we leave its careful consideration to future work.

B THE LUCIDITEE PROTOCOL

We formalize the protocol between the compute provider p_c (who runs the key manager, policy checking, and the computation) and the output recipients $P_{\mathbf{c}}^{\text{out}} = \{p_1, \ldots, p_n\}$ (who participate in the fair exchange protocol). p_c executes $\mathsf{Prot}_{\mathsf{compute}}[\mathsf{c}]$ (where c is the computation's specification), which runs the enclave programs

Key Manager Enclave $\mathsf{prog}_{\mathsf{keymgr}}$ (also denoted by $\mathcal{E}_k$)

```
• On input ("keygen")
    let (epk, esk) ← S.Keygen($1^\lambda$)
    ret HW.Quote(c.id ‖ epk)
• On input ("savekeys", msg)
    c ‖ name ‖ k ‖ $\sigma$ ← PKE.Dec(esk, msg)
    assert S.Verify(c.in[name], $\sigma$, c ‖ name ‖ k)
    let $\mathsf{k}_{\mathsf{seal}}$ ← HW.GenSealKey()
    keydb[c.id, name] := AEAD.Enc($\mathsf{k}_{\mathsf{seal}}$, k, c)
• On input ("sendkeys", $\mathsf{q}_\phi$)
    let ( $\mu$, c ‖ pk, _ ) ← $\mathsf{q}_\phi$
    assert HW.QuoteVerify($\mathsf{q}_\phi$) ∧ $\mu$ = c.$\phi$
    let $\mathsf{k}_{\mathsf{seal}}$ ← HW.GenSealKey()
    $\mathsf{k}_s$ ← AEAD.Dec($\mathsf{k}_{\mathsf{seal}}$, keydb[c.id, "state"], c)
    foreach ( $\mathsf{n}_i$: _ ) ∈ c.in:
        $\mathsf{k}_{x_i}$ ← AEAD.Dec($\mathsf{k}_{\mathsf{seal}}$, keydb[c.id, $\mathsf{n}_i$], c)
    ret HW.Quote(PKE.Enc(pk, $\mathsf{k}_s$ ‖ $\mathsf{k}_{x_1}, \ldots, \mathsf{k}_{x_m}$))
```

Policy Enclave $\mathsf{prog}_{\mathsf{policy}}[\mathsf{c}]$ (also denoted by $\mathcal{E}_\phi$)

```
• On input ("keygen")
    let (epk, esk) ← S.Keygen($1^\lambda$)
    ret HW.Quote(c.id ‖ epk)
• On input ("eval", $\mathsf{q}_\mathsf{k}$, ldgr, $\mathsf{ct}_s$, $\mathsf{ct}_{x_1}, \ldots, \mathsf{ct}_{x_m}$)
    assert "keygen" has successfully executed
    let ($\mu$, data, $\sigma$) ← $\mathsf{q}_\mathsf{k}$
    assert HW.QuoteVerify($\mathsf{q}_\mathsf{k}$) ∧ $\mu = \mu(\mathcal{E}_k)$
    let $\mathsf{k}_s, \mathsf{k}_{x_1}, \ldots, \mathsf{k}_{x_m}$ ← PKE.Dec(esk, data)
    let $s$ ← AE.Dec($\mathsf{k}_s$, $\mathsf{ct}_s$)
    let $x_1, \ldots, x_m$ ← AE.Dec($\mathsf{k}_{x_1}$, $\mathsf{ct}_{x_1}$) . . . AE.Dec($\mathsf{k}_{x_m}$, $\mathsf{ct}_{x_m}$)
    ret c.$\phi$(ldgr, H($\mathsf{ct}_s$), $s$, H($\mathsf{ct}_{x_1}$), $x_1$, . . ., H($\mathsf{ct}_{x_m}$), $x_m$)
• On input ("sendkeys", $\mathsf{q}_\mathsf{f}$)
    assert "eval" has successfully executed
    let ( $\mu$, $\mathsf{c}_{\mathsf{id}}$ ‖ epk, _ ) ← $\mathsf{q}_\mathsf{f}$
    assert HW.QuoteVerify($\mathsf{q}_\mathsf{f}$) ∧ $\mu$ = c.$f$ ∧ $\mathsf{c}_{\mathsf{id}}$ = c.id
    let $\hat{\mathsf{k}}$ ← PKE.Enc(epk, $\mathsf{k}_s, \mathsf{k}_{x_1}, \ldots, \mathsf{k}_{x_m}$)
    let data ← len(ldgr) ‖ $\hat{\mathsf{k}}$ ‖ H($\mathsf{ct}_s$), H($\mathsf{ct}_{x_1}$), . . . , H($\mathsf{ct}_{x_m}$)
    ret HW.Quote(data)
```

Compute Enclave $\mathsf{prog}_{\mathsf{compute}}[\mathsf{c}]$ (also denoted by $\mathcal{E}_f$)

```
• On input ("keygen")
    let (epk, esk) ← S.Keygen($1^\lambda$)
    ret HW.Quote(c.id ‖ epk)
• On input ("eval", $\mathsf{q}_\phi$, $\mathsf{ct}_s$, $\mathsf{ct}_{x_1}, \ldots, \mathsf{ct}_{x_m}$)
    assert "keygen" has successfully executed
    let ( $\mu$, data, _ ) ← $\mathsf{q}_\phi$
    assert HW.QuoteVerify($\mathsf{q}_\phi$) ∧ $\mu$ = c.$\phi$
    let t ‖ $\hat{\mathsf{k}}$ ‖ $h_s, h_{x_1}, \ldots, h_{x_m}$ ← data
    let $\mathsf{k}_s$ ‖ $\mathsf{k}_{x_1}, \ldots, \mathsf{k}_{x_m}$ ← PKE.Dec(esk, $\hat{\mathsf{k}}$)
    assert H($\mathsf{ct}_s$) = $h_s$
    assert H($\mathsf{ct}_{x_1}$) = $h_{x_1}$ ∧ . . . ∧ H($\mathsf{ct}_{x_m}$) = $h_{x_m}$
    let $s$ ← Dec($\mathsf{k}_s$, $\mathsf{ct}_s$)
    let $x_1, \ldots, x_m$ ← AE.Dec($\mathsf{k}_{x_1}$, $\mathsf{ct}_{x_1}$), . . . , AE.Dec($\mathsf{k}_{x_m}$, $\mathsf{ct}_{x_m}$)
    let $\{ (\mathsf{k}_p, \omega_p) \}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$ ← $\{$ HW.GenRnd(256) $\}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$
    let ($s'$, $y$) := c.$f$($s$, $x_1, \ldots, x_m$; $r$)
    let ($\mathsf{ct}_{s'}$, $\mathsf{ct}_y$) ← (Enc($\mathsf{k}_s$, $s'$), Enc($\mathsf{k}_{p_1} \oplus \ldots \oplus \mathsf{k}_{p_n}$, $y$))
    let shares ← $\{$ com($\mathsf{k}_p$; $\omega_p$) ‖ PKE.Enc($\mathsf{pk}_p$, $\mathsf{k}_p$ ‖$\omega_p$) $\}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$
    let rprt ← comp‖c.id‖t ‖ $h_{x_1}, \ldots, h_{x_m}$, $h_y$, H($s'$) ‖ shares
    ret HW.Quote(rprt) ‖ $\mathsf{ct}_{s'}$ ‖ $\mathsf{ct}_y$
```

Reconstruction Enclave $\mathsf{prog}_{\mathsf{rec}}[\mathsf{c}, h, \mathsf{sk}]$ (also denoted $\mathcal{E}_r$)

```
• On input ("keygen")
    let (epk, esk) ← S.Keygen($1^\lambda$)
    ret HW.Quote(c.id ‖ epk ‖ S.Sig(sk, epk))
• On input ("receive", $\{\mathsf{ct}_\mathsf{p}\}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$)
    let $\{ (\mathsf{k}_p, \omega_p) \}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$ ← $\{$ PKE.Dec(esk, $\mathsf{ct}_\mathsf{p}$) $\}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$
    assert opening of each com($\mathsf{k}_p$; $\omega_p$) in $\{ (\mathsf{k}_p, \omega_p) \}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$
    let $\mathsf{k}_y$ ← $\bigoplus_{p \in P^{\mathsf{c}}_{\mathsf{out}}} \mathsf{k}_p$
    ret ⊤
• On input ("advance", $\pi^1_\mathsf{L}$, $\sigma^1_\mathsf{L}$, $\mathsf{t}^1_\mathsf{L}$)
    let "del1" ‖ $h_y$ ‖ $\mathsf{c}_{\mathsf{id}}$ ‖ ($\sigma^1_y, \ldots, \sigma^n_y$) ← $\pi^1_\mathsf{L}$
    assert $\mathsf{Verify}_\mathsf{L}$($\sigma^1_\mathsf{L}$, $\mathsf{t}^1_\mathsf{L}$ ‖ $\pi^1_\mathsf{L}$)
    assert $h_y$ = $h$ ∧ $\mathsf{c}_{\mathsf{id}}$ = c.id
    assert S.Verify($\mathsf{vk}_{p_1}$, $\sigma^1_y$, $h_y$) ∧ . . . ∧ S.Verify($\mathsf{vk}_{p_n}$, $\sigma^n_y$, $h_y$)
    let enc_keys ← $\{$ PKE.Enc($\mathsf{pk}_{p_i}$, $\mathsf{k}_y$) $\}_{i \in [t+1, \ldots, n]}$
    let $\pi^2_\mathsf{L}$ ← HW.Quote("del2" ‖ $h_y$ ‖ $\mathsf{c}_{\mathsf{id}}$ ‖ enc_keys)
    ret $\pi^2_\mathsf{L}$
• On input ("reveal", $\pi^2_\mathsf{L}$, $\sigma^2_\mathsf{L}$, $\mathsf{t}^2_\mathsf{L}$)
    assert $\mathsf{Verify}_\mathsf{L}$($\sigma^2_\mathsf{L}$, $\mathsf{t}^2_\mathsf{L}$ ‖ $\pi^2_\mathsf{L}$)
    let ( $\mu$, data, _ ) ← $\pi^2_\mathsf{L}$
    assert $\mu = \mu(\mathcal{E}_r)$
    let "del2" ‖ $h'$ ‖ $\mathsf{c}'_{\mathsf{id}}$ ‖ _ ← data,
    assert $\mathsf{c}'_{\mathsf{id}}$ = c.id ∧ h′ = h
    ret $\mathsf{k}_y$
```

$\mathsf{prog}_{\mathsf{keymgr}}$ (also referred to as $\mathcal{E}_k$), $\mathsf{prog}_{\mathsf{policy}}[\mathsf{c}]$ (also referred to as $\mathcal{E}_\phi$), and $\mathsf{prog}_{\mathsf{compute}}[\mathsf{c}]$ (also referred to as $\mathcal{E}_f$). Each output recipient in $\{p_1, \ldots, p_n\}$ executes $\mathsf{Prot}_{\mathsf{reconstruction}}[\mathsf{c}, \mathsf{sk}, \mathsf{i}]$ (where $\mathsf{i} \in \{1, \ldots, n\}$ is its identifier, and sk is its secret signing key). Each output recipient in $\{p_1, \ldots, p_t\}$ must launch the enclave program $\mathsf{prog}_{\mathsf{rec}}[\mathsf{c}, \mathsf{sk}]$ (also referred to as $\mathcal{E}_r$). Note that fairness and policy compliance are ensured even when any subset of parties are corrupt, though the enclave programs are trusted in LucidiTEE.

B.1 Ensuring Termination

Since the compute phase, as described in $\mathsf{Prot}_{\mathsf{compute}}[\mathsf{c}]$, is carried out entirely by the compute provider, its non-termination is simply an act of denial of service, and is therefore not relevant to our discussion on termination. Instead, we focus our attention on the multi-party protocol for fair output delivery, where any subset of the parties may act maliciously. The protocol sketch described in § 6 ensures correctness (i.e. all parties receive the same output) and fairness (i.e. if any party gets the output, then all honest parties must also get the output) — the guarantee holds even when the compute provider is malicious and $t < n$ out of n output recipients act maliciously. However, we have not yet discussed how termination (i.e. adversary cannot force the honest parties to wait forever) can be guaranteed.

Output Recipient's Protocol $\mathsf{Prot}_{\mathsf{reconstruct}}[\mathsf{c}, h, \mathsf{sk}, \mathsf{i}]$

```
(1) Receive encrypted output
    recv encrypted output $\mathsf{ct}_y$, and assert $\mathsf{H}(\mathsf{ct}_y) = h$
    if $\mathsf{i} \leq t$ { send $\mathsf{S.Sig}(\mathsf{sk}, \mathsf{H}(\mathsf{ct}_y))$ to $P^{\mathsf{c}}_{\mathsf{out}}$; goto step (5) }
(2) Launch the $\mathcal{E}_r$ enclave (only for TEE party $\mathsf{i} \leq t$)
    let $\mathsf{e} \leftarrow \mathsf{HW.Load}(\mathsf{prog}_{\mathsf{rec}}[\mathsf{c}, h, \mathsf{sk}])$
    let $\mathsf{q}_{\mathsf{d}} \leftarrow \mathsf{HW.Run}(\mathsf{e}, (\text{“keygen”}))$
    send $\mathsf{q}_{\mathsf{d}}$ to each $p \in P^{\mathsf{c}}_{\mathsf{out}}$; recv $\mathsf{ct}_{\mathsf{p}}$ from each $p \in P^{\mathsf{c}}_{\mathsf{out}}$
    assert $\mathsf{HW.Run}(\mathsf{e}, (\text{“receive”}, \{\mathsf{ct}_{\mathsf{p}}\}_{p \in P^{\mathsf{c}}_{\mathsf{out}}})) = \top$
    send $\mathsf{S.Sig}(\mathsf{sk}, \mathsf{H}(\mathsf{ct}_y))$ to $p \in P^{\mathsf{c}}_{\mathsf{out}}$
(3) Post signatures on ledger (only for TEE party $\mathsf{i} \leq t$)
    recv $\sigma_p$ from each $p \in P^{\mathsf{c}}_{\mathsf{out}}$
    let $\pi^1_{\mathsf{L}} \leftarrow$ “del1” $\| \mathsf{H}(\mathsf{ct}_y) \| \mathsf{c.id} \| \{\sigma_p\}_{p \in P^{\mathsf{c}}_{\mathsf{out}}}$
    $\mathsf{L.post}(\pi^1_{\mathsf{L}})$
(4) Post encrypted key on ledger (only for TEE party $\mathsf{i} \leq t$)
    let $\mathsf{t}' \leftarrow \mathsf{L.getCurrentCounter}$
    let $\mathsf{t}^1_{\mathsf{L}} \leftarrow \mathsf{head}\,[\mathsf{t}^* \mid \mathsf{t}^* < \mathsf{t}', \mathsf{L.getContent}(\mathsf{t}^*) = (\_, h \| \mathsf{c.id} \| \_)]$
    assert $\mathsf{t}^1_{\mathsf{L}} \neq \bot$;  let $(\sigma^1_{\mathsf{L}}, \pi^1_{\mathsf{L}}) \leftarrow \mathsf{L.getContent}(\mathsf{t}^1_{\mathsf{L}})$
    let $\pi^2_{\mathsf{L}} \leftarrow \mathsf{HW.Run}(\mathsf{e}, (\text{“advance”}, \pi^1_{\mathsf{L}}, \sigma^1_{\mathsf{L}}, \mathsf{t}^1_{\mathsf{L}}))$
    $\mathsf{L.post}(\pi^2_{\mathsf{L}})$
(5) Retrieve key $\mathsf{k}_y$ and decrypt $\mathsf{ct}_y$
    let $\mathsf{t}' \leftarrow \mathsf{L.getCurrentCounter}$
    let $f \leftarrow \lambda\pi.\ \pi = (\_, \text{“del2”} \| h \| \mathsf{c.id} \| \_)$
    let $\mathsf{es} \leftarrow [(\sigma, \pi, \hat{\mathsf{t}}) | \hat{\mathsf{t}} < \mathsf{t}', (\_, \pi) \leftarrow \mathsf{L.getContent}(\hat{\mathsf{t}}), f(\pi)]$
    assert $\mathsf{size}(\mathsf{es}) = 1$
    let $\pi^2_{\mathsf{L}}, \sigma^2_{\mathsf{L}}, \mathsf{t}^2_{\mathsf{L}} \leftarrow \mathsf{es}[0]$
    if $\mathsf{i} \leq t$ { let $\mathsf{k}_y \leftarrow \mathsf{HW.Run}(\mathsf{e}, (\text{“reveal”}, \pi^2_{\mathsf{L}}, \sigma^2_{\mathsf{L}}, \mathsf{t}^2_{\mathsf{L}}))$ }
    else { extract $\mathsf{k}_y$ from $\pi^2_{\mathsf{L}}$ }
    ret $\mathsf{Dec}(\mathsf{k}_y, \mathsf{ct}_y)$
```

To ensure progress, parties must place time bounds, as opposed to blocking waits or receives, when receiving messages from other parties and when polling the shared ledger for a desired ledger entry. First, the protocol participants use a synchronous network model when sending and receiving messages with each other, where each round of the protocol has a fixed time period. This ensures that parties do not perform blocking receive on network messages. Next, to ensure progress when interacting with the ledger, we extend our fair reconstruction protocol in two ways: 1) setting timeout when polling the ledger for a desired entry (where timeout is expressed in the number of ledger entries), and 2) ensuring the ledger is always making progress (increasing in the number of ledger entries). The former is implemented by having parties abort when a protocol-specified number of entries are posted on the ledger, and the latter is ensured by having parties post dummy clock messages on the ledger (where at least one honest party will ensure progress).

C FAIR RECONSTRUCTION

In this section, we formally present and prove security of an implementation of fair secure computation when only t parties have access to a secure processor. Here, t refers to the number of corrupt parties. Recall that fairness is impossible when $t \geq n/2$, and that the previous best result [4] required n secure processors. Since this contribution is of independent interest, in the following presentation we follow the standard notation of n parties $P_1, \ldots, P_n$ interested in *fair reconstruction* of a “secret shared” value. It is well known that this fair reconstruction primitive is complete for fair secure computation [3, 4]. Furthermore (and following [4]), we assume that parties begin with commitments to their secret shares as public input (this distribution of *correlated randomness* can be generated via an unfair MPC or via a secure compute enclave). Note that only party P_i is aware of the opening to the i-th commitment.

C.1 Technicalities and formalization

For formalization, we use the $\mathcal{G}_{\mathsf{att}}$ abstraction proposed in [24], and used in [4, 24] to construct fair protocols. The $\mathcal{G}_{\mathsf{att}}$ abstraction has the concept of a registry (variable Reg), which indicates which parties have access to $\mathcal{G}_{\mathsf{att}}$. This allows easy modeling of parties that possess TEE (i.e., those that are registered) vs. parties that do not have access to a TEE.

Since $\mathcal{G}_{\mathsf{att}}$ is a global functionality, we will need to allow $\mathcal{S}$ to “program” the output via trapdoors in the enclave program. Note that in [4], all parties had access to $\mathcal{G}_{\mathsf{att}}$, and where attested messages were only used locally or exchanged between enclaves. In our case, parties that are not registered with $\mathcal{G}_{\mathsf{att}}$ will need to rely on attested outputs produced by $\mathcal{G}_{\mathsf{att}}$ to obtain their outputs. Therefore, we rely on techniques introduced in [24], specifically the use of $\mathcal{G}_{\mathsf{acrs}}$ [24, 69], for output programming. Input extraction, on the other hand is typically trivial for corrupt parties that are registered to $\mathcal{G}_{\mathsf{att}}$ (as $\mathcal{S}$ can read these inputs directly). In cases where corrupt parties (say that are not registered to $\mathcal{G}_{\mathsf{att}}$) send encrypted inputs to $\mathcal{G}_{\mathsf{att}}$, input extraction will rely on $\mathcal{G}_{\mathsf{acrs}}$ (again following techniques in [24]). Also, we will need to rely on techniques introduced in [24] for ensuring that parties have *deniability* (again via $\mathcal{G}_{\mathsf{acrs}}$ [24]).

We also heavily borrow techniques from [4], starting from the use of $\mathcal{F}_{\mathsf{blockchain}}$ (which they model as a bulletin board since they achieve only standalone security). Specifically, following [4] we augment $\mathcal{F}_{\mathsf{blockchain}}$ to provide L.getContent which includes a (publicly) verifiable authentication tag (which can act as a release token for the enclave), L.post which returns an authentication tag along with the height. We also use their technique of using bulletin board's authentication tags as release tokens to activate the enclave.

In more detail, our formal protocol results from combining ideas from (1) the single secure processor (one server with enclave and n remote clients) outsourced computation protocol of [24], and (2) the fair MPC protocol from [4]. From (1) we borrow simulation techniques (specifically backdoors via $\mathcal{G}_{\mathsf{acrs}}$) to handle parties that do not possess a secure processor. From (2) we borrow techniques for correct handling of ledger posts and release tokens (specifically backdoors for one-way permutations). We now state our theorem:

Theorem C.1. *Assume the existence of one-way permutations, a signature scheme that is existentially unforgeable under chosen message attacks, an authentication scheme that satisfies the standard notion of unforgeability, an encryption scheme that is IND-CCA2-secure, an authenticated encryption scheme that is perfectly correct and satisfies standard notions of INT-CTXT and semantic security, a proof system that satisfies computational soundness and witness indistinguishability. Furthermore, let $\mathcal{G}_{\mathsf{att}}, \mathcal{G}_{\mathsf{acrs}}$ be as defined in [24], and let $\mathcal{F}_{\mathsf{blockchain}}$ be as defined in [6] supporting bulletin board abstractions as defined in [4]. Then, for every $n > 0$ and $t < n$, there exists an n-party protocol that UC-realizes $\mathcal{F}_{\mathsf{fairmpc}}$ in the $(\mathcal{G}_{\mathsf{acrs}}, \mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{blockchain}})$-hybrid model in the presence of a static*

Enclave Program $\mathsf{prog}_{\mathsf{fairmpc}}$

Initial state. Set $E \leftarrow \perp$.

- On input (init), run $\mathsf{PKE.KeyGen}$ to generate $\{(pk_i, sk_i)\}_{i\in[n]}$. Return $PK = (pk_1, \ldots, pk_n)$.
- On input (input, $\{\mathsf{ct}_j\}_{j\in[n]}$): Let $(\mathbf{pub}_j = (\mathbf{com}_j, \mathsf{Reg}_j, Y_j, T_j), k_j, x_j, \omega_j) \leftarrow \mathsf{PKE.Dec}(sk_j, \mathsf{ct}_j)$. Return $\perp$ if $\exists i, i'$ s.t. $\mathbf{pub}_i \neq \mathbf{pub}_{i'}$. Parse $\mathbf{com}_1 = (c_1, \ldots, c_n)$. Return $\perp$ if $\exists i$, the value (x_i, ω_i) is not a valid opening of c_i. Return “ok”.
- On input (proceed, $t, \sigma_{\mathsf{L},S}, S$): Assert getParams already called. If proceed already called, return E. Return $\perp$ if $\mathsf{Verify}_{\mathsf{L}}(\sigma_{\mathsf{L},S}, t\|S) = 0$ or if $t > T_1 + \Delta t$. Parse $S = \{(j, \rho_j)\}_{j\in\mathsf{Reg}_1}$ and $Y_1 = \{(j, y_j)\}_{j\in\mathsf{Reg}_1}$. Return $\perp$ if $\exists j : f(\rho_j) \neq y_j$. Compute $e_j \leftarrow \mathsf{AE.Enc}_{k_j}(\bigoplus_{i\in[n]} x_i)$ for $j \notin \mathsf{Reg}_1$. Return $E \leftarrow \{(j, e_j)\}_{j\notin\mathsf{Reg}_1}$.
- On input (output, $t, \sigma_{\mathsf{L},E'}, E'$): Assert proceed is already called. Return $\perp$ if $\mathsf{Verify}_{\mathsf{L}}(\sigma_{\mathsf{L},E'}, t\|E') = 0$ or $E \notin E'$ or if $t > T_1 + 2\Delta t$. Return $\mathsf{outp}_i = \bigoplus_{i\in[n]} x_i$.

Simulator programming.

- On input (extract, $\{idk_i\}_{i\in[n]}$): for $i \in [n]$: if $\mathsf{check}(\mathcal{G}_{\mathsf{acrs}}.mpk, P_i, idk) = 1$,$v_i = sk_i$,else $v_i = \perp$; return $\{v_i\}_{i\in[n]}$.
- On input (program, $\{idk_i, u_i\}_{i\in[n]}$): for $i \in [n]$: if $\mathsf{check}(\mathcal{G}_{\mathsf{acrs}}.mpk, P_i, idk) = 1$, set $\mathsf{outp}_i = u_i$.
- On input (getParams, v): assert init, input have been called. if $v = \perp$, send Y_1, else return v.

adversary that can corrupt at most t out of the n parties, and where at least t out of the n parties are contained in $\mathcal{G}_{\mathsf{att}}$'s registry.

Next, we present the protocol along with the enclave description. At a high level, where we differ from [4] is we use a “two-phase” release protocol as follows. In the 1st phase (marked by a ledger entry), all enclaves (run by registered parties) are assured to have the necessary inputs to reconstruct the output. Now the enclaves post (on the blockchain) the reconstructed output encrypted under the non-registered parties' keys. This proof of publication on the blockchain acts as a release token for the enclaves to release the reconstructed outputs to the registered parties. The intuition is that when all enclaves agree that they have the necessary inputs to reconstruct the output, the following holds: (1) all honest enclaves can obtain the output by simply following the protocol (i.e., irrespective of the actions of malicious parties); this actually ensures that non-registered honest parties also obtain the outputs, and (2) if no enclave is honest, then for the adversary to obtain the output, it must provide a release token which corresponds to posting encrypted outputs under non-registered parties' keys; since all honest parties are non-registered, they obtain the output by simply scanning the ledger for the encrypted outputs.

C.2 Description of the simulator

We describe the simulator $\mathcal{S}$ who will simulate $\mathcal{F}_{\mathsf{blockchain}}$ to $\mathcal{A}$ and interact with $\mathcal{G}_{\mathsf{att}}$ to $\mathcal{A}$, and receive messages from $\mathcal{Z}$ on behalf

Fair reconstruction in ($\mathcal{G}_{\mathsf{acrs}}$, $\mathcal{F}_{\mathsf{blockchain}}$,$\mathcal{G}_{\mathsf{att}}$)-hybrid model

Public input. Time interval Δt. Commitments $\mathbf{com} = (c_1, \ldots, c_n)$ where $c_i = \mathsf{com}(x_i; \omega_i)$.
Inputs. Each party P_i has private input (x_i, ω_i).
Outputs. Each party receives $x = \bigoplus_{i\in[n]} x_i$ or none receive x.
Preliminaries. Let Reg denote the set of parties registered with $\mathcal{G}_{\mathsf{att}}$. Let f be a one-way permutation, PKE be a CCA-2 secure encryption scheme, AE be an authenticated encryption scheme. Let L denote the interface to $\mathcal{F}_{\mathsf{blockchain}}$. In the following, we assume that $P_i \in \mathsf{Reg}$ contacts its local enclave only and prevents other parties from contacting its enclave (i.e., all messages to its local enclave go via P_i). In the following: (1) whenever P_i is required to output an attested message, it sends the message along with a ciphertext and a NIWI proof that the ciphertext either encrypts a valid attestation on the message, or encrypts the recipients “identity key” (managed by $\mathcal{G}_{\mathsf{acrs}}$), and (2) when a recipient checks for a valid attestation, it verifies the NIWI proof.

Protocol. Party P_i executes the following:

Enclave initialization. If $P_i \in \mathsf{Reg}$:

- Send `install`(sid, $\mathsf{prog}_{\mathsf{fairmpc}}$) to $\mathcal{G}_{\mathsf{att}}$ to receive eid_i. Send eid_i to each $P_j \in \mathsf{Reg}$.
- Receive eid_j from each $P_j \in \mathsf{Reg} \setminus \{P_i\}$. If eid_j was not received from some $P_j \in \mathsf{Reg}$, terminate the protocol, and output $\perp$.

Input submission.

(1) Samples $k_i, \rho_i \leftarrow \{0, 1\}^\lambda$ and broadcasts $y_i = f(\rho_i)$ to all parties. Let Y_i denote the set of all received y_j's.

(2) If $P_i \in \mathsf{Reg}$: Send $\mathcal{G}_{\mathsf{att}}$.`resume`($eid_i$, init) to receive $PK_i = (pk_{i,1}, \ldots, pk_{i,n})$. Broadcast PK_i to all parties.

(3) For every $P_j \in \mathsf{Reg}$: P_i waits to receive PK_j. Then compute $\mathsf{ct}_{i,j} \leftarrow \mathsf{PKE.Enc}(pk_{j,i}, (\mathsf{pub}_i = (\mathbf{com}, \mathsf{Reg}, Y_i, T_i), k_i, x_i, \omega_i))$, where T_i equals $\mathsf{L.getCurrentCounter}$ at the start of the protocol.

(4) If $P_i \in \mathsf{Reg}$:
   - Collect all $\mathsf{ct}_{j,i}$ received from each P_j, and send $\mathcal{G}_{\mathsf{att}}$.`resume`($eid_i$, $\{\mathsf{ct}_{j,i}\}_{j\in[n]}$). Wait to receive “ok”.
   - If “ok” received, then broadcast ρ_i to all parties. Collect $S_i = \{j, \rho_j\}_{j\in\mathsf{Reg}}$, where for all $j \in \mathsf{Reg}$ it holds that $f(\rho_j) = y_j$.
   - Once S_i is collected with $|S_i| = |\mathsf{Reg}|$, execute $\mathsf{L.post}(S_i)$ to $\mathcal{F}_{\mathsf{blockchain}}$ (if S_i is not already on the ledger) to obtain $\sigma_{\mathsf{L},S}, \mathsf{t}_{\mathsf{L},S}$ (possibly via $\mathsf{L.getContent}(\mathsf{t}_{\mathsf{L,S}})$).
   - Send $\mathcal{G}_{\mathsf{att}}$.`resume`(proceed, $\sigma_{\mathsf{L},S}, \mathsf{t}_{\mathsf{L},S}$) to obtain E_i.
   - Broadcast E_i to all parties. Collect E_j received from all parties in Reg to form E'. Proceed even if not all E_j's are collected.
   - Execute $\mathsf{L.post}(E')$ to $\mathcal{F}_{\mathsf{blockchain}}$ (if E' is not already on the ledger) to obtain $\sigma_{\mathsf{L},E'}, \mathsf{t}_{\mathsf{L},E'}$ (possibly via $\mathsf{L.getContent}(\mathsf{t}_{\mathsf{L,E'}})$).
   - Send $\mathcal{G}_{\mathsf{att}}$.`resume`(output, $\mathsf{t}_{\mathsf{L},E'}, \sigma_{\mathsf{L},E'},$) to obtain x.

(5) If $P_i \notin \mathsf{Reg}$: Scan the ledger to obtain $t < T_i + \Delta t$ such that $\mathsf{L.getContent(t)} = (\sigma_{\mathsf{L},E'}, E')$, where E' has valid attestation under mpk, and for some $E \in E'$, we have $(i, e_i) \in E$. Output $\mathsf{AE.Dec}_{k_i}(e_i)$ and terminate.

of $\mathcal{A}$. The extraction of corrupt inputs is trivial via simulation of $\mathcal{G}_{\mathsf{att}}$, where $\mathcal{A}$ needs to send input encrypted under a public key for which the secret key is known to $\mathcal{S}$ (which equivocates via $\mathcal{G}_{\mathsf{acrs}}$ the public keys s.t. it knows the corresponding secret keys to decrypt) (irrespective of whether the corrupt party is in Reg or not), and $\mathcal{S}$ will simply verify whether the decryption is a valid opening to the corresponding commitments specified in the public input. During this phase, on behalf of the honest parties, $\mathcal{S}$ will send encryptions of 0 (instead of x_i, ω_i), which will remain indistinguishable to $\mathcal{A}$ due to the semantic security of the (authenticated) encryption scheme. Also, $\mathcal{S}$ chooses ρ_i uniformly at random to generate y_i values corresponding to honest parties. If some corrupt party P_i in Reg never contacted $\mathcal{S}$ to provide inputs then $\mathcal{S}$ terminates the simulation and outputs $\bot_{\mathbf{com}}$.

Once $\mathcal{S}$ receives correct encryptions from all corrupt parties, it generates the NIWI proof for the message $(sid, eid_i, \mathsf{prog}_{\mathsf{fairmpc}}, \text{“ok”})$ using the corrupt party's identity key as the witness (i.e., as opposed to the attestation). For every simulated honest party for which $\mathcal{S}$ generates the NIWI proof, it broadcasts the corresponding simulated ρ value. (If there is no corrupt party in Reg, then $\mathcal{S}$ sends the corresponding encryption which can be monitored by $\mathcal{A}$.) If $\mathcal{S}$ did not broadcast simulated ρ values for *all* honest parties in Reg (before $T + \Delta t$), then $\mathcal{S}$ terminates the simulation and outputs $\bot_f$. $\mathcal{S}$ waits to receive the corresponding ρ values from corrupt parties in Reg. Next $\mathcal{S}$ proceeds to act as $\mathcal{F}_{\mathsf{blockchain}}$. We now consider two cases. If all ρ values for parties in Reg are obtained by any particular simulated honest party in Reg, then acting as that party, $\mathcal{S}$ simulates the posting of message $S = \{i, \rho_i\}_{i \in \mathsf{Reg}}$ on $\mathcal{F}_{\mathsf{blockchain}}$ by notifying $\mathcal{A}$. During this round, $\mathcal{S}$ accepts any message from $\mathcal{A}$, and simulates expected behavior of $\mathcal{F}_{\mathsf{blockchain}}$. The other case is when no simulated honest party in Reg received all ρ values. Now $\mathcal{S}$ (acting as $\mathcal{F}_{\mathsf{blockchain}}$) waits to receive a message S' from $\mathcal{A}$ containing all the preimages. If no such message was received (within some specified timeout), then $\mathcal{S}$ terminates the simulation and outputs $\bot_{\mathsf{L}}$. Else, $\mathcal{S}$ simulates the expected behavior of $\mathcal{F}_{\mathsf{blockchain}}$, and simulates posting of S', to return a valid tag $\sigma_{\mathsf{L},S'}$ to $\mathcal{A}$.

At this point, we are assured that $\mathcal{S}$ acting as $\mathcal{F}_{\mathsf{blockchain}}$ has received all valid preimages of the y values. Now, we split the simulation again in two cases. We deal with the first case: Suppose some honest $P_j \in$ Reg. In this case, the output needs to be delivered to all parties. Therefore, $\mathcal{S}$ contacts $\mathcal{F}_{\mathsf{fairmpc}}$ to obtain the final output x, and then upon receiving a valid programs $\mathcal{G}_{\mathsf{att}}$'s output (using $\mathcal{G}_{\mathsf{acrs}}$ identity key for the corrupt party) delivers E which consists of the encryptions of the output x under the shared symmetric keys of all parties not in Reg. After this, $\mathcal{S}$ simulates the expected behavior of $\mathcal{F}_{\mathsf{blockchain}}$ (say, to continue posting messages on behalf of the corrupt parties) till the protocol completes, and then simulates the expected behavior of $\mathcal{G}_{\mathsf{att}}$ by programming the output (via $\mathcal{G}_{\mathsf{acrs}}$) to deliver encrypted outputs that can be decrypted by the corresponding corrupt parties, and finally terminates the simulation. Thus, in this case, all (honest) parties receive the final output.

Finally, we deal with the second case where no honest $P_j \in$ Reg. (**Note:** Since we have $|\mathsf{Reg}| \geq t$, it follows that all corrupt parties are in Reg.)

In this case, $\mathcal{S}$ acting as $\mathcal{G}_{\mathsf{att}}$ waits to receive $\sigma_{\mathsf{L},S'}$. If a valid tag (valid wrt $\mathcal{F}_{\mathsf{blockchain}}$) is received from $\mathcal{A}$ corrupt P_i, then $\mathcal{S}$ prepares E_i consisting of encryptions of 0 (i.e., not the output which is still unknown to $\mathcal{S}$) for each party not in Reg (recall all of them are honest). Then, acting as $\mathcal{F}_{\mathsf{blockchain}}$ waits to receive posting of a message E_j (for some E_j that was programmed by $\mathcal{S}$ in the previous step). If no such E_j was received, then $\mathcal{S}$ outputs $\bot_{\mathsf{sig}}$ and terminates. If a correct message was posted by a corrupt $P_i \in$ Reg, then $\mathcal{S}$ contacts $\mathcal{F}_{\mathsf{fairmpc}}$ to receive the final output x, which it then sends the output x (again by programming the output via $\mathcal{G}_{\mathsf{acrs}}$) to each corrupt $P_i \in$ Reg, and terminates the simulation.

If no valid tag was received from $\mathcal{A}$ (within $T + 2\Delta t$), then $\mathcal{S}$ terminates the simulation and sends $\bot_{\mathsf{L}}$ to $\mathcal{F}_{\mathsf{fairmpc}}$.

This completes the description of the simulator.

Our strategy to show indistinguishability of real and ideal world follows closely to that of [4] (the main difference being we need to rely on $\mathcal{G}_{\mathsf{acrs}}$ for extraction and equivocation (like in [24]), and we also use CCA-2 secure public key encryption scheme since not all parties are registered, and the enclave can be queried with say incorrect decommitments). Like in [4], we need a backdoor getParams for the hybrid step involving the one-way permutation. In the description of the simulation above, we noted points of failure along with the associated primitive. Following this, it is a straightforward exercise to write out the hybrid steps in the order of the failure steps in which they appear. We defer the full proof to the full version of the paper.