LucidiTEE: Policy-Compliant Fair Computing at Scale

Rohit Sinha
Visa Research
rohit.sinha@visa.com

Sivanarayana Gaddam
Visa
sgaddam@visa.com

Ranjit Kumaresan
Visa Research
rakumare@visa.com

ABSTRACT

We seek a system that provides transparency and control to users by 1) enforcing agreed-upon policies on what functions can be evaluated over private data (even when the users are offline), and 2) enforcing the set of parties with whom the results are shared. For this level of control, the system must ensure *policy compliance*, and we demonstrate, using modern applications, the need for *history-based policies*, where any decision to compute on users' data depends on prior use of that data. Moreover, the system must algorithmically ensure *fairness*: if any party gets the output, then so do all honest parties. It is an open research challenge to construct a system that ensures these properties in a malicious setting.

While trusted execution environments (TEEs), such as Intel SGX and Sanctum enclaves, offer partial solutions, they are at the mercy of an untrusted host software for storage and network I/O, and are therefore incapable of enforcing history-dependent policies or fairness. This paper presents LucidiTEE, the first system to enable multiple parties to jointly compute on large-scale private data, while guaranteeing policy-compliance even when the input providers are offline, and fairness to all output recipients. A key contribution is our protocol (with a formal proof of security) for fair n-party information exchange, which tolerates an arbitrary corruption threshold $t < n$, and requires only t parties to possess a TEE node (an improvement over prior result that requires TEEs from all n parties) — in our case studies, this result provides a practical benefit as end users on commodity devices can enjoy fairness when engaging with service providers. We define an ideal functionality for policy-compliant fair computing, $\mathcal{F}_{PCFC}$, which is the first to study history-based policies, and we develop novel protocols based on a network of TEEs and a shared ledger to enforce history-based policies.

LucidiTEE realizes $\mathcal{F}_{PCFC}$ with a heavy focus on efficiency. It uses the ledger only to enforce policies; i.e., it does not store inputs, outputs, or state on the ledger, which allows it to scale to large data and large number of parties. We demonstrate several policy-based applications including a personal finance app, federated machine learning, and policy-based surveys amongst unknown participants.

KEYWORDS

Privacy, Policy, Fairness, Trusted Execution Environment, Blockchain

1 INTRODUCTION

Modern web services pose a growing public concern regarding their lack of transparency. Aside from privacy policies specified in legalese, users have little insight, let alone control, on how their data is used or shared with third parties. It is not surprising that (unbeknownst to users) their sensitive data is proliferated and misused. While increased transparency and control would require significant changes across the board (including business models, regulations, etc.), there is still progress to be made on the technical front.

We observe that several services can be modeled as a stateful computation over data from multiple parties (comprising both the end users and the service provider). Moreover, a typical service performs computation over large datasets, on behalf of a large number of users, and allows users to go offline during the computation. For such a service to provide transparency and control to all parties, we need a system that (at the very least): 1) enforces agreed-upon policies on what functions can be evaluated over the joint datasets, along with an option for any party to revoke further use of their data, and 2) enforces the set of parties with whom the results are shared. Specifically, rather than relying solely on trust or legal recourse, protocols within the system must enforce *policy compliance* even when the input providers go offline during the computation, and ensure *fairness* towards the agreed-upon set of output recipients (i.e. if any party gets the output, then so do all honest parties). It is an open research challenge to construct a system that ensures both policy compliance and fairness to all parties, in a setting where any subset of the parties act maliciously, and where the computation is carried out by a malicious provider.

We first consider some potential solutions to this end. Since policies impose restrictions on how the input data is used, as a first step, the system must encrypt the data under keys controlled by the input providers — this prevents a compute provider from performing arbitrary computation and sharing of the raw data. Next, to perform computation over the encrypted data, we look towards functionalities such as multi-party computation (MPC) [1–3] and trusted execution environments (TEEs) [4, 5]. While protocols for MPC ensure that only the agreed-upon function over the inputs is revealed, they require parties to either be online to engage in an interactive protocol or trust one or more compute providers to execute the protocol on their behalf (using a secret sharing scheme, for instance). Alternatively, we can perform the computation within a TEE, based on one of several recent frameworks such as Ryoan [6], VC3 [7], Opaque [8], etc. This approach obviates any interaction after the user provides the input to a TEE, and also performs better on general-purpose computation over large data. However, neither policy compliance nor fairness can be fulfilled by TEEs alone, due to their inherent inability to protect I/O — a malicious compute provider (controlling the system software, such as the OS) can rollback the persistent storage and tamper with the network communication. For instance, the attacker can rollback the storage to violate a privacy budget policy (e.g. limited passcode attempts for a digital lockbox, or the privacy budget in differentially private database), or intercept network communication to deliver the output only to a subset of colluding parties. Addressing these threats, this paper presents LucidiTEE, a system for **policy-compliant fair computation** using protocols based on TEEs and a shared ledger.

A key contribution is an ideal functionality definition $\mathcal{F}_{PCFC}$ in the UC framework [9] for policy-compliant fair computation, which is realized by LucidiTEE. We believe that $\mathcal{F}_{PCFC}$'s interface lends

itself to building applications that process large data and service large number of users (e.g. personal finance application in § 9.1.1), where the entire set of participating users may not be known apriori (e.g. private survey in § 9.1.4) and the users cannot be expected to be online (beyond providing inputs or retrieving outputs in future).

A key concept in $\mathcal{F}_{PCFC}$ is **history-based policies**, and a protocol for enforcing their compliance in the presence of a malicious compute provider. $\mathcal{F}_{PCFC}$ maintains an audit 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, thus enabling policies based on the computation's history. As we show in § 9.1, example history-based policies include privacy budget [10], k-time programs [11], freshness of inputs, "democratic" accounting of all participating users' inputs (such as in voting), policies across multiple computations (e.g. survey only open to users of an app), etc. Finally, $\mathcal{F}_{PCFC}$ ensures fairness on output delivery, i.e., should any party get the output, then all honest parties must also get the output (although a malicious provider can deny sending the results to any party).

We develop LucidiTEE, a system that realizes $\mathcal{F}_{PCFC}$, using protocols based on ideal functionalities of TEE [12] and a shared ledger [13] to achieve both policy-compliance and fairness. By reducing fair computation to fair reconstruction (inspired by [14, 15]), LucidiTEE divides the computation into distinct phases: 1) providing input, 2) function evaluation producing encrypted outputs, 3) fair reconstruction for attaining the output. With this design, no interaction is needed from input providers and output recipients beyond providing inputs and retrieving outputs, respectively. Of independent interest is our protocol for **fair n-party exchange** (used in fair reconstruction) that withstands arbitrary corruption threshold $t < n$, requiring only t out of n output recipients to possess a TEE machine and access the shared ledger — this improves upon a prior result [13] for fair MPC using a shared ledger that requires all n parties to possess a TEE machine. This improvement provides a practical benefit, especially in bilateral service relationships between a user and a service provider (see § 9.1.1 and § 9.1.3), where the user enjoys fairness without owning a TEE machine.

LucidiTEE records the cryptographic hash digests of the (encrypted) inputs, outputs, and the updated state on the ledger after each function evaluation. We use the ledger only to enforce history-based policies, keeping both computation and storage off-chain, thus fostering applications with large number of users and large data. Our protocols withstand arbitrary corruption threshold amongst the participants of a computation, and allow for a malicious compute provider, as we defend against attacks such as providing a stale view of the ledger to TEE programs, aborts, etc.

Recent ledger-based systems use cryptographic primitives and/or TEEs to provide private computation. Hawk [16] and Zexe [17] support limited types of smart contracts, and their use of zero-knowledge proofs [18] adds significant computational overhead; moreover, Hawk only provides financial fairness based on penalties. Ekiden [19] supports generic smart contracts, but does not ensure fairness nor history-based policies (across computations), and is not efficient for large data computation, as inputs and state are stored on-chain. To our knowledge, LucidiTEE is the first system to provide history-based policies, and we demonstrate applications (§ 9.1) of such policies that cannot be implemented on Ekiden.

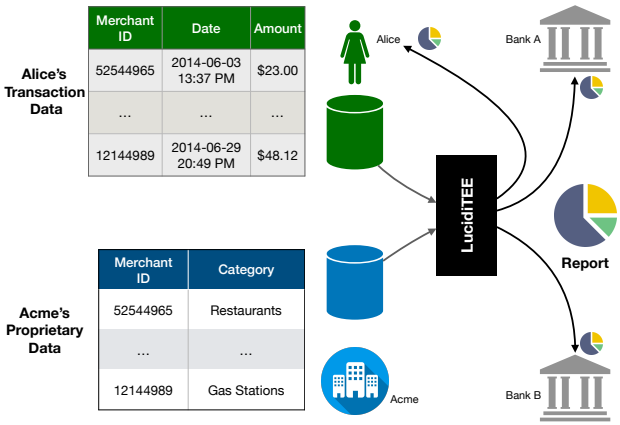


Figure 1: Privacy-enabled Personal Finance Application

In summary, this paper makes the following novel contributions:

- definition of an ideal functionality $\mathcal{F}_{PCFC}$ 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, and scales to large data applications
- protocol for fair n-party exchange, requiring a shared ledger and t parties to control a TEE (for any corruption threshold $t < n$)
- evaluation of several applications on LucidiTEE, including a personal finance application (serving millions of users), federated machine learning over crowdsourced data, and a private survey. We also implement micro-benchmarks including one-time programs, digital lockboxes, and fair n-party information exchange.

2 OVERVIEW OF LUCIDITEE

2.1 Motivating Example: Personal Finance App

The open banking initiative [20] has fostered a variety of personal financial applications. However, by getting access to raw transaction data, some of these applications pose major public concerns of data breaches and misuse [21], as there is lack of (verifiable) transparency on the sharing, use, and retention of this data.

Figure 1 illustrates a sample financial application, Acme, that 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). 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 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 [20] that pushes transaction data one-by-one as they are generated by her. Today,

an application like Acme often hosts the users' raw data to provide the service, and is inevitably trusted to adhere to a legalese policy.

The rest of this section proceeds as follows. We first discuss the requirements of a transparent Acme, while iterating on strawman system designs and motivating the building blocks used by LucidiTEE. Next, we discuss how Acme is implemented on LucidiTEE.

2.2 Requirements of Acme

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. For this discussion, f is assumed to be safely written with functional correctness, so we are only concerned with executing f as given.

> **Strawman 1**
> To restrict how their input data is used, both parties must first encrypt their data before uploading it to a *compute provider* (who may be 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. At some point in future, the enclave program evaluates f, and produces an output encrypted under the public keys of all output recipients, which is then delivered to them.

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 [7], Opaque [8], StealthDB [22], Ryoan [6], etc., which we can use to implement the function f.

History-based Policies

While this strawman ensures that only f can be evaluated on Alice's input, we show that this policy alone is insufficient for 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 [20]), 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 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, such *rewind-and-fork* attacks are detrimental for applications that maintain a privacy budget (e.g. digital lockboxes, differentially private databases), and applications that use secret randomness.

To counter such attacks, we enforce *history-based policies*, where the decision to execute the next step in a (stateful) computation depends on that computation's entire history — to our knowledge, this is the first work to study such policies. In Acme's example, Alice uses the following history-based policy ϕ[2]: all transaction records must 1) be fresh, in that they have never been consumed by a prior evaluation of f, and 2) belong to the same month.

In general, 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 other applications of 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) — applications on Ekiden [19] cannot access the ledger, and therefore cannot implement such behaviours.

> **Strawman 2**
> Enforcement of such policies require an audit log of all function evaluations. To that end, we modify the earlier Strawman 1 design to include an append-only ledger, shared between Alice and Acme. The ledger fulfills a dual purpose. First, a protocol forces the compute provider to record the enclave's evaluation of f on the ledger before extracting the encrypted output from the enclave — for each function evaluation, we record some metadata (e.g. hash digests or accumulators) of the encrypted inputs, outputs, and intermediate state. Second, enclave programs read this ledger to evaluate the history-based policy predicate ϕ.

The Strawman 2 design shares the ledger between Alice and Acme. In practice, a user may use the output of a function f as an input to another function f', where f and f' may receive additional inputs from other users — we call this *compute chaining*. This gives rise to a computation (directed, acyclic) graph containing evaluation of various functions, with inputs and outputs belonging to multiple users. For example, upon entering a domestic partnership with Bob, Alice requests Acme for a cumulative monthly report over both their transactions. To that end, we extend the shared ledger between Alice and Acme to include Bob as well. In general, to enable arbitrary groups of users to engage in unforeseen computation, we share the ledger amongst all parties and all concurrent computations. Details

[1] Although metadata, such as authenticated batches of inputs, can remedy this specific attack, banks may be unwilling to generate metadata specific to each application.
[2] While ϕ may be inlined within f, we find that it is worthwhile distinguishing the two functions, as ϕ depends on the ledger whereas f does not.

on policy enforcement are given in § 5, but we emphasize that this design enforces policies even when the input providers are offline.

Fairness

Policies provide a mechanism for input providers to restrict the use of their data; on the other hand, our desiderata for a transparent Acme also includes enforcing 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 host software, an enclave cannot ensure that a message is sent over the network, and therefore, must assume lossy links, making reliable message delivery impossible [23]. For Acme to be transparent, we argue that it must ensure fairness: if any party gets the output, then all honest parties must get the output. Choudhuri et al [13] provide a fair MPC protocol that requires all parties to possess a TEE machine, but it is unreasonable to place such a burden on Alice.

Without having to trust Acme, Alice wishes to have her aggregate monthly reports sent automatically to a set of chosen banks, perhaps to receive lucrative mortgage plans. To that end, we develop a novel protocol to ensure fair delivery to all output recipients, in a malicious setting where any subset of parties are corrupt.

> **Strawman 3**
> We reduce fair computation to fair reconstruction, inspired by [15]. First, we use the Strawman 2 system to produce the output, encrypted under the public keys of all output recipients, while also checking the history-based policy. Next, to send the decryption key to all parties, we design a protocol for fair n-party broadcast that withstands an arbitrary corruption threshold $t < n$; the protocol requires t (out of n output recipients) to use a TEE machine that they control, and all n parties to access the shared ledger.

In general, many services have a bilateral relationship (2-party computation) between a user and the service provider, where fairness can be achieved with only 1 TEE node at the service provider, allowing the user to use commodity devices. With this successive refinement of the strawman solutions, we now arrive at LucidiTEE.

2.3 Acme on LucidiTEE

LucidiTEE implements a set of protocols (between TEE enclaves and a shared ledger) that guarantee fair, policy-compliant computation. We walk through how Acme is deployed on LucidiTEE.

Specifying and Creating Computations

A computation is specified by a string, which any party can post to the ledger. For instance, Acme creates the following computation:

```
computation { id: 42, /* unique id */
  input:  [ ("txs": pk_Alice), ("db": pk_Acme) ],
  output: [ ("rprt": [pk_Alice, pk_BankA, pk_BankB]) ],
  policy: 0xcoff..eeee, /* ∀r ∈ txs. fresh(r) */
  func: 0x1337...c0de /* aggregate function */ }
```

The `id` field is a 64-bit value that uniquely identifies this computation on LucidiTEE. 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 output will be encrypted under their public keys). The `func` field uniquely identifies the function f using the hash measurement of the enclave program implementing f. The history-based policy predicate ϕ is specified within the field `policy`. Similar to `func`, it contains the hash measurement of the enclave program implementing ϕ.

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 (with measurement `func`) and satisfy ϕ (with measurement `policy`). In our example, f is evaluated over the inputs `txs` and `db`. 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 implement ϕ by performing membership test for each transaction in the `txs` within the input consumed by prior evaluations of f in computation of `id` 42, or more efficiently, by maintaining state tracking the latest processed transaction.

Encrypting and Binding Inputs to Computations

Alice wishes to provide her input with the guarantee that only policy-compliant computation is performed on it. As a first step, she chooses a key k to encrypt `txs`, and uploads the encrypted data to an untrusted storage (e.g. Acme's server). Next, Alice needs to bind the use of key k to computation of `id` 42, which may execute when she is offline. To that end, we introduce a *key manager enclave* in LucidiTEE (see Figure 2), which is run by any party such as the compute provider, and whose role is to protect k and reveal it only to an enclave evaluating `func` from a computation of user-specified `id`. Alice provisions k and the computation's `id` 42 over a TLS channel terminating within a *key manager enclave*, who then seals the key and stores it locally. Note that Alice's bank, should it provide the functionality, can also perform these steps on her behalf.

Invoking and Recording Computation

The compute provider provisions 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, which requires checking: 1) the computation's specification exists on the ledger and has not been revoked, and 2) the policy ϕ is satisfied. To that end, Acme launches an enclave implementing the policy predicate ϕ, and provides it with a view of the ledger. To evaluate ϕ, the enclave must decrypt the inputs and state, for which it contacts the key manager enclave — the key manager enclave verifies using remote attestation that the request originates from a genuine enclave (with hash measurement specified in the `policy` field of the computation's specification). On approval from ϕ, Acme launches an enclave to evaluate f, and provides access to the encrypted inputs and state

(where the decryption keys are given by the prior enclave that evaluated ϕ). Evaluation of f produces encrypted values of the output and the next state, but the evaluation must be recorded on the ledger before releasing them, for the sake of history-based policies.

To that end, LucidiTEE implements a protocol (see § 5) between the ledger and the enclave that ensures atomicity of the following events: recording the evaluation on the ledger and revealing the output to any party. We record cryptographic digests (e.g. Merkle tree root) of the encrypted inputs, outputs, and state.

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

As described in the Strawman 3 solution, we develop a novel protocol for fair n-party exchange (see § 6), which the n output recipients use to reconstruct the decryption key that protects the output. The protocol may execute at any time after the computation (e.g., when Alice comes online), and requires the parties to interact only with the compute provider or the shared ledger. By default, we set a maximum corruption threshold $t = n - 1$, in which case only BankA and BankB control a TEE node, and Alice only interacts with the ledger. In general, we demonstrate services with bilateral relationships between a user and a service provider (see § 9.1.1 and § 9.1.3), where the user can enjoy fairness without owning a TEE machine.

3 LUCIDITEE SPECIFICATION

3.1 Participants and Threat Model

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 $P^{\mathbf{c}}_{\text{in}}$ and a set of output recipients $P^{\mathbf{c}}_{\text{out}}$, which may overlap, such that $(P^{\mathbf{c}}_{\text{in}} \cup P^{\mathbf{c}}_{\text{out}}) \subseteq P^*$. 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 $P^{\mathcal{A}}$.

3.2 Ideal Functionality

We introduce an ideal functionality, $\mathcal{F}_{PCFC}$, for policy-compliant fair computations. A computation is modeled as a state transition system, where each step evaluates a transition function f if allowed by the history-dependent policy ϕ (expressed over the inputs, state, and a log 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 $P^{\mathbf{c}}_{\text{in}}$ and the output recipients $P^{\mathbf{c}}_{\text{out}}$. Due to compute chaining, $\mathbf{c}$ also specifies the set of 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 $\mathbf{c}$ (such as in Acme), 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. $\mathcal{F}_{PCFC}$ is defined as follows:

Policy Compliant Fair Computing: $\mathcal{F}_{PCFC}$

The functionality has a private storage db and a publicly readable log ldgr. 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 chained computations $C^{\mathbf{c}}_{\text{in}}$ and $C^{\mathbf{c}}_{\text{out}}$, and the set of parties $P^{\mathbf{c}}_{\text{in}}$ and $P^{\mathbf{c}}_{\text{out}}$. The **ret** statement performs **send** and terminates the command.
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 $\perp$

- 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** $\perp$ to p }
  ldgr.append(create ‖ $\mathbf{c}$); **ret** $\top$ to $[p, \mathcal{A}]$
- On **command** revoke_computation($\mathbf{c}$.id) from $p \in P^{\mathbf{c}}_{\text{in}}$
  send (revoke ‖ $\mathbf{c}$.id ‖ p) to $\mathcal{A}$
  if ¬active($\mathbf{c}$) { **ret** $\perp$ 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 $p \in P^{\mathbf{c}}_{\text{in}}$
  send (bind_input ‖ p ‖ $\mathbf{c}$.id ‖ h) to $\mathcal{A}$
  if (¬active($\mathbf{c}$) $\vee$ db[h] = $\perp$) { **ret** $\perp$ to $[\mathcal{A}, p]$ }
  let $(x, p', C) \leftarrow$ db[h]; if ($p' \neq p$) { **ret** $\perp$ 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_{in}) from $p \in P^*$
  send (compute ‖ $\mathbf{c}$.id ‖ H_{in} ‖ p) to $\mathcal{A}$
  if $\mathcal{A}$ denies or ¬active($\mathbf{c}$) then { **ret** $\perp$ 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 $\leftarrow \forall p \in P^{\mathbf{c}}_{\text{in}}\ \exists(\_, p', \_) =$ db[$H_{\text{in}}[p]$] $\wedge$ $p' = p$
  let compliant $\leftarrow \mathbf{c}.\phi$(ldgr, s, X, H_{in})
  if ¬(bound $\wedge$ owned $\wedge$ compliant) then { **ret** $\perp$ to $[\mathcal{A}, p]$ }
  let $(s', Y) \leftarrow \mathbf{c}.f(s, X; r)$, where $r \xleftarrow{\$} \{0,1\}^{\lambda}$
  let $h_{s'} \leftarrow$ db.add($(s', \perp, \{\mathbf{c}\})$)
  let $H_{\text{out}} \leftarrow$ { db.add($(y, \perp, \emptyset)$) }$_{y \in Y}$
  for $\mathbf{c} \in C^{\mathbf{c}}_{\text{out}}$ { db.update($H_{\text{out}}[\mathbf{c}]$, $(Y[\mathbf{c}], \perp, \{\mathbf{c}\})$) }
  ldgr.append(compute ‖ $\mathbf{c}$.id ‖ $h_{s'}$ ‖ H_{in} ‖ H_{out})
  send {| y |}$_{y \in Y}$ ‖ | s' | to $\mathcal{A}$; **ret** $\top$ to p
- On **command** get_output($\mathbf{c}$.id, h) from $p \in P^{\mathbf{c}}_{\text{in}}$
  send (output ‖ h ‖ p) to $\mathcal{A}$; **send** $\top$ to p
  if $\exists p \in P^{\mathbf{c}}_{\text{in}}$ 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)

$\mathcal{F}_{PCFC}$ 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 $\mathbf{c}.\texttt{inp}$, preventing future evaluations of $\mathbf{c}.f$.

A party p uploads an input x (using provide_input), which $\mathcal{F}_{PCFC}$ 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 the 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 $\mathbf{c}.\phi$, and as such, a policy can enforce one-time programs [11]). Party p may bind x to multiple computations concurrently, as long as each of them list p as an input provider. We find that these characteristics make $\mathcal{F}_{PCFC}$ suitable for 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 $P^{\mathbf{c}}_{\mathsf{in}}$ and chained computations $C^{\mathbf{c}}_{\mathsf{in}}$). Since $\texttt{ldgr}$ records the handle for the state after each evaluation, $\mathcal{F}_{PCFC}$ uses $\texttt{ldgr}$ to retrieve the most recent state. Party p can provide any handles of her choice, as $\mathcal{F}_{PCFC}$ checks the guard $\mathbf{c}.\phi$ prior to evaluating $\mathbf{c}.f$, in addition to some sanity checks that the inputs are existent and bound to the invoked 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 $P^{\mathbf{c}}_{\mathsf{out}}$ and chained computations $C^{\mathbf{c}}_{\mathsf{out}}$) — 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, $\mathcal{F}_{PCFC}$ records the evaluation on $\texttt{ldgr}$, along with the relevant handles.

The output recipients initiate fair output delivery by invoking get_output, and the output is sent once all parties have invoked the command. $\mathcal{A}$ may prevent $\mathcal{F}_{PCFC}$ from sending the output; however, should any party get the output, then all recipients in $P^{\mathbf{c}}_{\mathsf{out}}$ get the output. In summary, $\mathcal{F}_{PCFC}$ guarantees that:

* $\mathcal{A}$ 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 $P^{\mathbf{c}}_{\mathsf{out}}$.
* Parties in $P^{\mathbf{c}}_{\mathsf{out}}$ get the correct output with fairness.

Fair reactive computation is out of scope, since $\mathcal{A}$ can deny executing the compute command and computations are revocable.

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 before writing them to DRAM. 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 [24]. HW maintains the memory contents of each enclave program in the internal state variable $\texttt{mem}$, and supports 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 $\texttt{mem}[\texttt{eid}] = \overrightarrow{0}$.
- HW.Run(eid, in) executes enclave eid (from prior state $\texttt{mem}[\texttt{eid}]$) under input in, producing an output out while also updating $\texttt{mem}[\texttt{eid}]$. The command returns the pair (out, quote), where quote is a signature over $\mu(\text{prog}) \,\|\, \text{out}$, attesting that out originated from an enclave with hash measurement $\mu(\text{prog})$ running on a genuine TEE. We also write the quote as $\text{quote}_{\mathsf{HW}}(\text{prog}, \text{out})$.
- HW.QuoteVerify(quote) verifies the genuineness of quote and returns another signature σ (such that $\text{Verify}_{\mathsf{HW}}(\sigma, \text{quote}) = \text{true}$) that is publicly verifiable. Any party can check $\text{Verify}_{\mathsf{HW}}$ 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. Any party can verify σ (using Intel's public key) without contacting Intel's attestation service.

We assume that the remote attestation scheme is existentially unforgeable under chosen message attacks [24]. LucidiTEE assumes ideal TEE platforms that provide *secure remote execution*, as defined in [25]. Though side channel attacks pose a realistic threat, we consider their defenses to be an orthogonal problem. This assumption is discharged in part by using safer enclave processors 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., [26], [27], [28], [29], [30]). From here on, we assume that HW only reveals the explicit output out; i.e., in a computation, the enclaves only reveal the output produced by ϕ and f.

4.2 Shared, Append-only Ledger

We borrow the bulletin board abstraction of a shared ledger, defined in [13], 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 indeed been added to the ledger. Specifically, σ is an authentication tag over the pair t‖e such that $\text{Verify}_{\mathsf{L}}(\sigma, \text{t}\|\text{e}) = \text{true}$, where $\text{Verify}_{\mathsf{L}}$ is a public algorithm that any party can run without access to L.
- L.getContent(t) returns the ledger entry (σ, e) at height t, or $\perp$ if t is greater than the current height of the ledger.

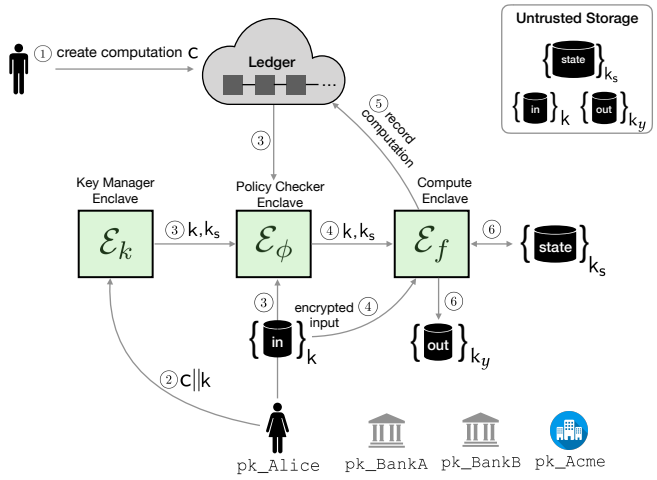


Figure 2: Policy enforcement using TEEs and a shared ledger

The bulletin board abstraction can be instantiated using fork-less blockchains, such as permissioned blockchains [31], and potentially even by blockchains based on proof-of-stake [32].

4.3 Cryptographic Primitives and Assumptions

Hash Function. We use a hash function H (e.g. SHA-256) that is collision-resistant and pre-image resistant. We also assume a commitment scheme com with hiding and binding properties.

Public Key Encryption. We assume a *IND-CCA2* [33] public key encryption scheme PKE (e.g. RSA-OAEP) consisting of polynomial-time algorithms PKE.Keygen(1^λ), PKE.Enc(pk, m), PKE.Dec(sk, ct).

Digital Signature Scheme. We assume a *EUF-CMA* [34] digital signature scheme S (e.g. ECDSA) consisting of polynomial-time algorithms S.Keygen(1^λ), S.Sig(sk, m), S.Verify(vk, σ, m).

Symmetric Key Encryption. We use a scheme for authenticated encryption AE (e.g. AES-GCM) that provides *IND-CCA2* and *INT-CTXT* [35]. It consists of polynomial-time algorithms AE.Keygen(1^λ), AE.Enc(k, m), AE.Dec(k, ct).

5 POLICY-COMPLIANT COMPUTATION

LucidiTEE realizes $\mathcal{F}_{PCFC}$, assuming ideal functionalities for TEE hardware HW and shared ledger L. By reducing fair computation to fair reconstruction, LucidiTEE decomposes a computation into distinct phases: 1) providing and binding the input to the computation, 2) evaluating ϕ and f, producing encrypted outputs, and 3) fair reconstruction for attaining the output. This section describes the first two phases, leaving the fairness protocol to § 6.

Figure 2 illustrates the primary components of LucidiTEE. Each entry on the shared, public ledger records either the creation or revocation of a computation (along with its specification), or a function evaluation (which records fixed sized digests of inputs, outputs, and state). We stress that the ledger does not store the inputs or state, and its contents 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, and recording the computation on the ledger). These enclaves are run on one or more physical TEE machines, managed by any untrusted party called the *compute provider*, 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.

5.1 Specifying and Creating a Computation

A computation's specification is a string with following syntax:

Name	n	::=	$[\mathtt{a}-\mathtt{zA}-\mathtt{Z0}-9]+$
Party	p	::=	$\{0,1\}^*$
Input	i	::=	$(\mathsf{n}:\mathsf{p}) \mid (\mathsf{n}:\rho^?) \mid (\mathsf{n}:(\mathsf{z},\mathsf{n}))$
Output	o	::=	$(\mathsf{n}:[\mathsf{p},\ldots,\mathsf{p}]) \mid (\mathsf{n}:(\mathsf{z},\mathsf{n}))$
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 $(\mathsf{n}:\rho^?)$. 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 — similarly, the destination computation must have a corresponding tuple identifying the output from the source computation. We use a hash h to encode the expected measurement of enclaves containing the code of f and ϕ. Finally, we specify a computation **c** by a string with fields: an id, hash of $\mathcal{E}_f$ (func) implementing function f, hash of $\mathcal{E}_\phi$ (policy) implementing ϕ, input description inp, and output description out.

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

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

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. 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 party $p \in P^{\mathbf{c}}_{\mathsf{in}}$ (i.e., p is listed as an input provider in **c**.inp) can revoke a computation by invoking revoke_computation(**c**.id), which LucidiTEE implements by having p execute the following:

$$p : \mathsf{L.post}(\mathtt{revoke} \parallel \mathbf{c}.\mathtt{id})$$

5.2 Providing Inputs

A party submits data to a computation by invoking provide_input(x), which returns a unique handle to the input data. $\mathcal{F}_{PCFC}$ maintains privacy of that data, and guarantees that the data is unmodified when later used by a computation (even when the input provider is offline). 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 inputs, but that is equivalent 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). We derive the handle h by computing a cryptographic digest, using hash functions or authenticated data structures such as a Merkle tree, over the ciphertext (e.g. H(AE.Enc(k, x))), and return h to the calling party. Generally, LucidiTEE uses cryptographic hash digests to assign handles to inputs, output, and state values, where the digest is computed over the encrypted value to protect values from low-entropy distributions.

5.3 Binding Inputs to Computations

Recall the bind_input($\mathbf{c}$.id, h) command in $\mathcal{F}_{PCFC}$, which makes the user-provided input x (referred by h) accessible by computation $\mathbf{c}$. 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 $\mathbf{c}$ 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 $\mathcal{E}_k$):

$$p : \mathsf{L.post}(\ \mathtt{bind} \parallel \mathbf{c}.\mathtt{id} \parallel h \parallel \mathsf{S.Sig}(\mathsf{sk}_p, \mathbf{c}.\mathtt{id} \parallel h)\ )$$

$$p_c \rightarrow p : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_k, \mathsf{pk}), \text{where } \mathsf{pk} \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$$

$$p \rightarrow p_c : \mathsf{PKE.Enc}(\mathsf{pk}, \mathbf{c}.\mathtt{id} \parallel \mathsf{k} \parallel \mathsf{n} \parallel \mathsf{S.Sig}(\mathsf{sk}_p, \mathbf{c}.\mathtt{id} \| \mathsf{k} \| \mathsf{n}))$$

First, p creates a ledger entry binding the data handle h (returned by bind_input) to computation $\mathbf{c}$. 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 sk_p) signs and encrypts $\mathbf{c}$.id and k, along with n which specifies the name of the input (from the list $\mathbf{c}$.inp). The logic of $\mathcal{E}_k$ is such that it will only later reveal the key k to an enclave that is computing on $\mathbf{c}$.id, either for evaluating ϕ or f.

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

5.4 Checking Policy-Compliance

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

To evaluate f, p_c must first launch $\mathcal{E}_\phi$ to evaluate ϕ. First, p_c downloads the encrypted state and inputs (from untrusted storage) for all the input providers listed in $\mathbf{c}$.inp. $\mathcal{E}_\phi$ contacts $\mathcal{E}_k$ in order to get the decryption keys for inputs $x_1, \ldots, x_m$ and state s, as follows:

$$\mathcal{E}_\phi \rightarrow \mathcal{E}_k : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_\phi, \mathbf{c}.\mathtt{id} \parallel \mathsf{pk}), \text{where } \mathsf{pk} \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$$

$$\mathcal{E}_k \rightarrow \mathcal{E}_\phi : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_k, \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{k}_s \parallel \mathsf{k}_{x_1} \parallel \ldots \parallel \mathsf{k}_{x_m}))$$

Next, $\mathcal{E}_\phi$ must check whether the requested evaluation of f is compliant with ϕ, which requires checking three conditions:

(1) active: $\mathbf{c}$ is created on the ledger, and not yet revoked.
(2) bound: data for each $h \in H_{\mathsf{in}}$ is bound to computation $\mathbf{c}$
(3) compliant: policy ϕ (evaluated over the ledger contents, prior state s, inputs $x_1, \ldots, x_m$, and input handles H_{in}) is valid.

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 on $\mathcal{E}_\phi$'s machine (which is fully controlled by p_c). Although using $\mathsf{Verify}_\mathsf{L}$ 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 § 5.6. For now, we task ourselves with deciding compliance with respect to a certain height of L.

The policy ϕ is an arbitrary function, implemented as an enclave program with the hash measurement listed in $\mathbf{c}$.policy. As an example, consider the policy ϕ from the Acme application: all transaction records in the input must be signed by Alice, belong to the same month, and be *fresh*. An efficient method of enforcing freshness is to propagate state containing the timestamp of the latest transaction record (from Alice's input in the previous evaluation), and then assert that all transaction records in the current input have a later timestamp, thus inductively implying freshness. To that end, we implement f such that it records the highest timestamp amongst the input txs within the computation's state, and use ϕ to prevent rollback of state. To evaluate ϕ, we ask p_c to provide $\mathcal{E}_\phi$ with $\mathbf{c}$'s encrypted state (which must have the handle h_s recorded in most recent entry of $\mathbf{c}$.id on the ledger), and transfer control to the entry-point of ϕ within $\mathcal{E}_\phi$, which takes each transaction record in the current input txs and compares its timestamp with the previous state. Alternatively, ϕ can load the inputs from all prior computations and check membership of each record in the current input txs; this choice of ϕ would incur significantly more computation and storage overhead. On that note, we stress that LucidiTEE simply provides the developers with the means to enforce history-dependent policies. The performance and the privacy guarantees ultimately depend on the developer's choice of ϕ.

5.5 Producing Encrypted Output

On approval from $\mathcal{E}_\phi$, p_c launches the compute enclave $\mathcal{E}_f$, which must have measurement $\mathbf{c}$.func. $\mathcal{E}_f$ requests $\mathcal{E}_\phi$ for the key k_s protecting $\mathbf{c}$'s state and the keys protecting each party's input:

$$\mathcal{E}_f \rightarrow \mathcal{E}_\phi : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_f, \mathbf{c}.\mathtt{id} \parallel \mathsf{pk}), \text{where } \mathsf{pk} \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$$

$$\mathcal{E}_\phi \rightarrow \mathcal{E}_f : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_\phi, \mathsf{t} \parallel \hat{\mathsf{k}} \parallel h_s \parallel H_{\mathsf{in}})$$
$$\text{where } \mathsf{t} \doteq \mathsf{L.getCurrentCounter used in } \mathcal{E}_\phi$$
$$\text{where } \hat{\mathsf{k}} \doteq \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{k}_s \parallel \mathsf{k}_{x_1} \parallel \ \ldots \ \parallel \mathsf{k}_{x_m})$$

Recall that $\mathcal{E}_\phi$ checked compliance of ϕ with respect to a certain height of the ledger t (albeit potentially stale), and with respect to input handles H_{in}. Therefore, $\mathcal{E}_\phi$ transmits these values to $\mathcal{E}_f$, in addition to the decryption keys protecting the state and inputs.

A randomized f needs an entropy source. Using secret key k_s (which was generated by $\mathcal{E}_k$ using the TEE's entropy source, such as rdrand on SGX), f can internally seed a pseudo-random generator (e.g. PRF with key H(t ‖ k_s)) to get a fresh pseudo-random

bitstream at each step of the computation (hence the use of t). This ensures that the random bits are private to $\mathcal{E}_f$, yet allows replaying computation from the ledger during crash recovery, for instance.

Once f produces the output y and next state s', $\mathcal{E}_f$ encrypts s' using k_s, and encrypts output y in a way that requires all output recipients to participate in a fair reconstruction protocol (§ 6) to decrypt the ciphertext. In the case of Acme, $\mathcal{E}_f$ emits $\mathsf{AE.Enc}(\mathsf{k}_1 \oplus \mathsf{k}_2 \oplus \mathsf{k}_3, \mathsf{rprt}) \parallel \mathsf{PKE.Enc}(\mathsf{pk\_Alice}, \mathsf{k}_1) \parallel \mathsf{PKE.Enc}(\mathsf{pk\_BankA}, \mathsf{k}_2) \parallel \mathsf{PKE.Enc}(\mathsf{pk\_BankB}, \mathsf{k}_3)$, where k_1, k_2, and k_3 are randomly chosen. Moreover, $\mathcal{E}_f$ emits commitments to help enforce correctness in our reconstruction protocol. More generally, $\mathcal{E}_f$ emits the following:

$$\mathsf{AE.Enc}(\bigoplus_{p \in P^{\mathbf{c}}_{\mathsf{out}}} \mathsf{k}_p, y) \parallel \{\, \mathsf{com}(\mathsf{k}_p; \omega_p) \| \mathsf{PKE.Enc}(\mathsf{pk}_p, \mathsf{k}_p \| \omega_p) \,\}_{p \in P^{\mathbf{c}}_{\mathsf{out}}}$$

5.6 Recording Computation on Ledger

As history-based policies need a log of all function evaluations, the compute provider p_c must post $\mathcal{E}_f$'s evaluation to the ledger before the output recipients can engage in fair reconstruction, as follows:

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

The use of $\mathsf{quote}_{\mathsf{HW}}$ ensures that the computation was performed for $\mathbf{c}.\mathtt{id}$ in a genuine TEE, consuming inputs with handles H_{in}, and producing outputs with handles H_{out} and next state with handle $h_{s'}$. Moreover, the quote contains the ledger height t, with respect to which $\mathcal{E}_\phi$ checked compliance. 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, 2) execution time of ϕ and f, and 3) malicious p_c causing $\mathcal{E}_\phi$ to evaluate ϕ using a stale view of L. This may potentially invalidate $\mathcal{E}_\phi$'s compliance check, but instead of simply rejecting the computation (which would unnecessarily limit concurrency), we assert the following *validity predicate* on the ledger's contents:

$$\forall \mathsf{t}, \mathsf{t}', \mathsf{t}^*.\ \mathsf{t}' = \mathsf{L.getCurrentCounter} \wedge \mathsf{t} < \mathsf{t}^* < \mathsf{t}' \Rightarrow$$
$$\neg((\sigma, \mathsf{e}) = \mathsf{L.getContent}(\mathsf{t}^*)\ \wedge\ \mathsf{Verify}_{\mathsf{L}}(\sigma, \mathsf{t}^* \| \mathsf{e}) \wedge$$
$$(\mathsf{e} = \mathtt{compute} \| \mathbf{c}.\mathtt{id} \| \ldots \vee \mathsf{e} = \mathtt{bind} \| \mathbf{c}.\mathtt{id} \| \ldots \vee \mathsf{e} = \mathtt{revoke} \| \mathbf{c}.\mathtt{id}))$$

Here, L checks that the computation $\mathbf{c}$ is still active and that no new function evaluation or bind for $\mathbf{c}.\mathtt{id}$ 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 new 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 $\mathcal{E}_d$ from § 6) to assert the validity predicate, and abort any computation (i.e., avoid posting entries) on an invalid ledger.

5.7 Chaining Computation

LucidiTEE supports concurrent, stateful computations amongst arbitrary sets of parties. Moreover, via *compute chaining*, outputs of a computation can serve as inputs in another computation (of different id), thereby forming a (directed, acyclic) graph composition of computations. For instance, the output model of a training phase can be used as an input during the inference phase, where the history-based policy ensures that inference uses the most recently trained model. To support compute chaining, we introduce the following modifications to the protocols discussed so far.

First, in addition to asserting the validity check presented in § 5.6, we assert that no evaluation is performed between t and t' on a computation $\mathbf{c}^*$ whose output is consumed by $\mathbf{c}$, as follows:

$$\forall \mathsf{t}', \mathsf{t}^*, \mathbf{c}^* \in \mathbf{c}.\mathtt{inp}.\ (\mathsf{t}' = \mathsf{L.getCurrentCounter} \wedge \mathsf{t} < \mathsf{t}^* < \mathsf{t}')$$
$$\Rightarrow\ \neg(\ (\sigma, \mathsf{e}) = \mathsf{L.getContent}(\mathsf{t}^*)\ \wedge\ \mathsf{Verify}_{\mathsf{L}}(\sigma, \mathsf{t}^* \| \mathsf{e})$$
$$\wedge\ \mathsf{e} = \mathtt{compute} \parallel \mathbf{c}^*.\mathtt{id} \parallel \ldots\ )$$

Second, we modify $\mathcal{E}_k$ to maintain an additional key $\mathsf{k}_{\mathbf{c}}^{\mathbf{c}^*}$ for encrypting the chained output from $\mathbf{c}^*$ to $\mathbf{c}$ (which must not be visible to any party). $\mathcal{E}_k$ generates $\mathsf{k}_{\mathbf{c}}^{\mathbf{c}^*}$ when $\mathbf{c}^*$ is created, and transmits $\mathsf{k}_{\mathbf{c}}^{\mathbf{c}^*}$ (along with $\{\mathsf{k}_{x_1}, \ldots, \mathsf{k}_{x_m}\}$ and k_s) to $\mathcal{E}_\phi$ and $\mathcal{E}_f$ in an evaluation.

6 FAIR COMPUTATION

At any time after the evaluation of f terminates, the output recipients ($P^{\mathbf{c}}_{\mathsf{out}} \doteq \{p_1, \ldots, p_n\}$ listed in $\mathbf{c}.\mathtt{out}$) engage in a fair reconstruction protocol to decrypt the output, with fairness: if any party gets the output, then so do all honest parties. Fairness is non-trivial as the enclave's I/O is controlled by an untrusted compute provider p_c, who may collude with any party. LucidiTEE implements fair reconstruction using a novel protocol for fair n-party exchange, that withstands arbitrary corruption threshold $t < n$ amongst the n output recipients, and ensures both fairness and correctness (i.e., all parties receive the same output). Although fairness is impossible in the standard setting with dishonest majority [36], 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. [13] requires all n parties to possess a TEE machine to avail fairness.

Our protocol works as follows. Recall from § 5.5 and § 5.6 and that $\mathcal{E}_f$ emits an encrypted output along with the ledger-bound quote, which must first be posted on the ledger for the fair reconstruction protocol to commence — further recall that the output y is encrypted under key $\mathsf{k}_y \doteq \mathsf{k}_{p_1} \oplus \ldots \oplus \mathsf{k}_{p_n}$, where each party $p_i \in P^{\mathbf{c}}_{\mathsf{out}}$ is given k_{p_i}. First, we introduce a new enclave program containing the reconstruction protocol's logic, called the *delivery enclave* $\mathcal{E}_d$, and we make the t parties with TEE nodes launch a local instance of $\mathcal{E}_d$. Next, each party p_i in $P^{\mathbf{c}}_{\mathsf{out}}$ transmits her secret key k_{p_i} to each $\mathcal{E}_d$, thus reconstructing k_y in each $\mathcal{E}_d$. Finally, we must engage in a fair broadcast of k_y out of $\mathcal{E}_d$ to all parties in $P^{\mathbf{c}}_{\mathsf{out}}$. 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_y under the public keys of the non-TEE parties $\{p_{t+1}, \ldots, p_n\}$, and 2) it triggers the t $\mathcal{E}_d$ enclaves to release k_y to their local parties $\{p_1, \ldots, p_t\}$. Our key insight is that by requiring t TEE nodes for corruption threshold t, we avoid the case where we have two corrupt parties, one with and one without TEE, who can abort and collude to get k_y. Figure 3 illustrates our protocol for Acme's example (where we set $t = n-1$, thus obviating Alice's responsibility to bring a TEE node). The rest of this section details the steps of this protocol.

(1) *Signed Delivery of Encrypted Outputs and Keys.* As a first step, we must ensure that all n output recipients have received

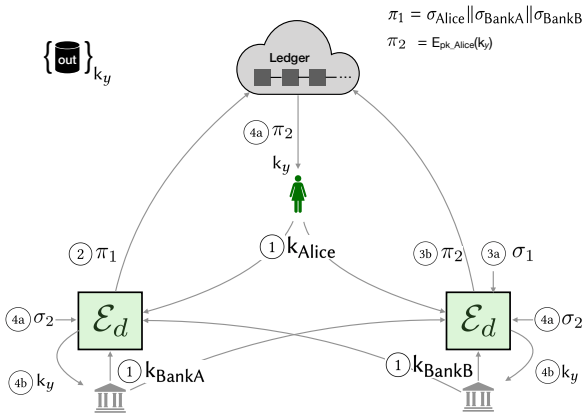


Figure 3: Fair reconstruction using TEEs and a shared ledger

the encrypted output $\mathsf{AE.Enc}(k_y, y)$, and that the t parties with TEEs have received all key shares $\{k_{p_1}, \ldots, k_{p_n}\}$ within their local instance of $\mathcal{E}_d$. We open commitments produced by $\mathcal{E}_f$ within $\mathcal{E}_d$ to ensure correctness of key shares. We do not prescribe a specific mechanism for transmitting these values; any party can act as a leader and broadcast them, or parties can form pairwise channels.

$$p_i \rightarrow p_j : \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_d, \mathtt{c.id} \parallel \mathsf{pk}), \text{ where } \mathsf{pk} \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$$
$$p_j \rightarrow p_i : \mathsf{S.Sig}(\mathsf{sk}_{p_j},\ h_y \parallel \mathsf{PKE.Enc}(\mathsf{pk}, k_{p_j} \parallel \omega_{p_j}))$$

(2) *Posting Signatures on the Ledger.* Each party computes a hash digest of their local copy of the encrypted output, and compares with the quote produced by $\mathcal{E}_f$. Furthermore, the t parties have their local $\mathcal{E}_d$ verify the commitment of the key shares. If both checks succeed, the party produces a signature, acknowledging their receipt of the encrypted output and key shares (held within $\mathcal{E}_d$). Any one of the n parties can collect signatures from all parties and post them on the ledger, as follows:

$$p \ : \ \mathsf{L.post}(\pi_1), \text{ which returns } (\sigma_1, \_)$$
$$\pi_1 \doteq \mathsf{S.Sig}(\mathsf{sk}_{p_1}, \mathtt{c.id} \parallel h_y) \parallel \ldots \parallel \mathsf{S.Sig}(\mathsf{sk}_{p_n}, \mathtt{c.id} \parallel h_y)$$

The n signatures are computed on the same message, and can be aggregated (e.g., by using [37]). As an optimization, we combine the post of π_1 with $\mathcal{E}_f$'s quote (from § 5.6) to reduce our usage of L.

(3) *Posting Encryption of Key k_y 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, or by invoking L.getContent). On providing σ_1 as input, $\mathcal{E}_d$ emits π_2, containing an encryption of k_y under the public keys of $n - t$ parties $\{p_{t+1}, \ldots, p_n\}$, which can be posted on L.

$$p_{\in\{p_1,\ldots,p_t\}} \ : \ \mathsf{L.post}(\pi_2), \text{ which returns } (\sigma_2, \_)$$
$$\pi_2 \doteq \mathsf{quote}_{\mathsf{HW}}(\mathcal{E}_d, \mathtt{c.id} \parallel h_y \parallel \{\mathsf{PKE.Enc}(\mathsf{pk}_p, k_y)\}_{p\in\{p_{t+1},\ldots,p_n\}})$$

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

?? presents a formal protocol in the $\mathcal{G}_{\mathsf{att}}$-hybrid model [12, 13].

7 DISCUSSION

We must draw attention to the subtlety of blockchain instantiations. While our fair delivery protocol tolerates a corruption threshold of t amongst n participants, 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 fair information exchange — 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, this limitation is not unique to us, as the fair exchange protocol in [13] also has the same limitation.

Fundamentally, forks on proof-of-work blockchains can violate policies, as computation records can be lost (akin to double spending in Bitcoin). Even the proof-of-publication scheme in Ekiden [19], 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. Hence, we deploy LucidiTEE on forkless ledgers (providing the bulletin-board abstraction L), such as HyperLedger [31] and Tendermint [38], though we can potentially deploy on public forkless blockchains based on proof-of-stake [32].

LucidiTEE does not support fair reactive computation, and is not suitable for applications such as Poker [15]. The primary issue here is that the compute provider p_c may collude with any party to abort the computation and destroy its intermediate state, thus preventing any honest party from making progress.

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 [39], whose index is the computation's id. We instantiate the shared ledger with a permissioned blockchain, and evaluate using both Hyperledger [31] and Tendermint [38]. The ledger participant's logic is implemented as a smart contract (in 200 lines of Go code), which internally uses RocksDB [40].

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 $\mathcal{E}_d$. libmoat transparently encrypts and authenticates all operations to the files and databases, using the scheme from § 5.2 — 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 $\mathcal{E}_d$ and $\mathcal{E}_k$, whose measurements are hard-coded within the smart contract and within libmoat, for use during remote attestation. Furthermore, libmoat implements the ledger interface L, which automatically verifies the signature (using Verify_L) and TEE attestation of all ledger entries (of type `compute` and `deliver`). libmoat contains 3K LOC, in addition to Intel's SGX SDK [41].

9 EVALUATION

9.1 Case Studies

We demonstrate applications which require history-based policies and fairness. In addition to the following applications, we build micro-benchmarks such as one-time programs [11], digital lockboxes (with limited guesses), and 2-party fair information exchange.

9.1.1 ***Personal Finance Application.*** We implement Acme's personal finance application using the following specification.

```
computation { id : 1,
  inp : [("txs":pk_Alice), ("db":pk_Acme)],
  out : [ ("rprt" : [pk_Alice, pk_BankA, pk_BankB])
        ,("rprt" : (3, "rprt_alice")) ],
  policy : 0xcoff..eee, /* ∀r ∈ txs. fresh(r)*/
  func : 0x1337...c0de /* aggregate function */ }
computation { id : 3,
  inp : [("rprt_alice":(1,"rprt")), ("rprt_bob":(2,"rprt"))],
  out : [ ("joint" : [pk_Alice, pk_Bob, pk_BnkA, pk_BnkB]) ],
  policy : 0xc000...10ff, /* same month? */
  func : 0x1ce..b00da /* joint report function */ }
```

Alice's computation (id 1) is chained with a computation (id 3) for producing a joint report along with Bob's expenses. The policy from computation 1 asserts that all transaction records belong to the same calendar month and are fresh (i.e., not used in a prior evaluation by Acme), and the policy from computation 3 asserts that the input reports belong to the same calendar month. 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 size 1.6GB). We also implemented a client that uses the OFX API [20] to download the user's transactions from their bank, and encrypt and upload the file (order of few MBs) to a public AWS S3 storage.

9.1.2 ***Private Survey.*** We conduct two surveys (with a simple tallying function), both amongst an unknown set of participants, with the history-based policy that only parties who voted on survey 1 can participate in survey 2. Moreover, for democratic reasons, both surveys require that all submitted votes be tallied — this is also a history-based policy expressed over the set of bind_input commands on the ledger. We use the following specification:

```
computation { id : 1,
  inp : [ ("vote": ρ?) ], out : [ ("result" : [pk_Acme]) ],
  policy : 0xcoff..eee, /* use all votes */
  func : 0x1337...c0de /* vote tally function */ }
computation { id : 2,
  inp : [ ("vote": ρ?) ], out : [ ("result" : [pk_Acme]) ],
  policy : 0xc000...10ff, /* use all votes that voted on 1 */
  func : 0x1ce..b00da /* vote tally function */ }
```

9.1.3 ***Federated Machine Learning.*** A hospital sets up a computation for any user to avail the prediction of a model (specifically the ECG class of a patient, used to detect Arrhythmia), 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 we achieve without requiring the user to possess a TEE node. Retraining happens for batches of new user data, so when a user submits their ECG data, they wish to use the latest model — this acts as our history-based policy. For the experiment, we use the UCI Machine Learning Repository [42], and adapt the k-means clustering algorithm and implementation from [43].

```
computation { id : 4,
  inp : [ ("training_data": pk_Hospital)],
  out : [ ("model" : (5: "model")) ],
  policy : 0xdaff..0d11, /* good accuracy on test set */
  func : 0xf1e...1d5 /* k-means clustering */ }
computation { id : 5,
  inp : [ ("model": (4, model)), ("input": ρ?) ],
  out : [ ("result" : [pk_Hospital, ρ?]) ],
  policy : 0xdaff..0d11, /* latest model produced by cid 4 */
  func : 0xf1e...1d5 /* k-means inference */ }
```

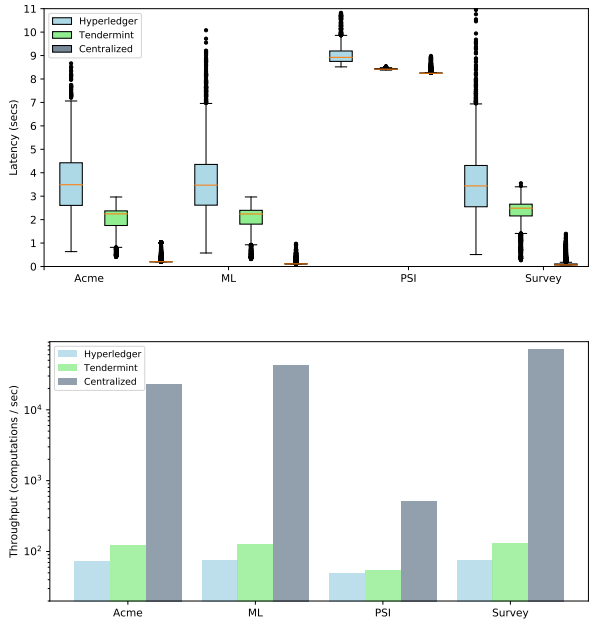
Observe the use of $\rho^?$ to denote an unknown user, whose public key is established only at the time of bind_input and compute.

9.1.4 ***Policy-based Private Set Intersection.*** Two hospitals A and B wish to share prescription records about their common patients, which we model as a 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 enclave-based private contact discovery service [44]. Our experiment uses a synthetic dataset with 1 million patient 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 (ignoring ϕ), using the aforementioned input 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 [31] and Tendermint [38] 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 $\mathcal{E}_d$. Recall that each evaluation on LucidiTEE performs at least one read query (often 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



Application	Ledger	Input	Output	State
Acme Finance	2372 B	1.6 GB	1872 B	136 B
Federated ML	2052 B	132 KB	1088 B	-
Policy-based PSI	2052 B	30 MB	8 MB	-
Private Survey	526.4 MB	954.4 MB	2 KB	-

Figure 4: Latency, Throughput, and Storage Overheads

dependent on parameters such as the batch size and batch timeout [31], with the exception of the PSI application (where each function call took roughly 8.2 seconds) which was compute bound. For that reason, we also evaluate on a "centralized" ledger, which is a local logging service, demonstrating throughput and latency with an ideally-performant ledger. Compared to the baseline, the latency also suffered by several seconds, as the ledger faced a high volume of concurrent read and write requests, causing wait time.

9.2.2 Storage. Figure 4 presents the off-chain and on-chain storage, which we compute for each function evaluation (including calls 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, the size of inputs are orders of magnitude greater than the ledger storage; in comparison, Ekiden [19] stores inputs and state on the ledger.

10 RELATED WORK

TEEs, such as Intel SGX, are finding use in systems for outsourced computing, such as M2R [45], VC3 [7], Opaque [8], EnclaveDB [46], etc. We find these systems to be complementary, in that users can use them to implement the compute function (i.e., f), while LucidiTEE handles history-based policy enforcement and fairness.

Ekiden [19], Coco [47], and Private Data Objects [48] are the most closely related works, in that they rely on shared ledger and trusted hardware functionalities. Ekiden executes smart contracts within SGX enclaves, connected to a blockchain for persisting the contract's state. 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 number of parties and the size of their inputs. In addition to performance improvements, the ideal functionalities differ in: 1) $\mathcal{F}_{PCFC}$ enforces history-based policies (both within and across computations, whereas Ekiden smart contracts do not read the ledger); 2) an attacker can prevent Ekiden's ideal functionality from sending the output to a party. To our knowledge, none of [19], [47], or [48] provide fairness in a malicious setting with arbitrary corruption threshold.

Hawk [16] enables parties to perform off-chain computation with transactional privacy, while proving correctness by posting zero knowledge proofs on the ledger. As mentioned in [19], Hawk supports limited types of computation, and only provides financial fairness (i.e., fairness with penalties, as opposed to perfect fairness provided by LucidiTEE and [13]). On that note, several works prior to Hawk, specifically Bentov et al. [15], Kumaresan et al. [49], Andrychowicz et al. [50, 51], and Kaiyias et al. [52], use Bitcoin [53] to ensure financial fairness in MPC protocols. Pass et al. [12] develop a protocol for 2-party fair exchange in the Δ-fairness model.

MPC [1] [2] [54] protocols implement a reactive secure computation functionality, but require parties to be online (or trust one or more third parties to execute the protocol on their behalf). furthermore, from Cleve's impossibility result [36], fairness is also impossible in the standard model with dishonest majority. Recently, Choudhuri et al. [13] proposed a fair MPC protocol based on witness encryption (instantiated using SGX 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; for instance, two distrusting parties can perform fair MPC on LucidiTEE requiring only one party to own a TEE machine. Moreover, [13] requires all parties to be online, and only considers one-shot MPC as opposed to stateful computation with policies. [55] augments stateless enclaves with shared ledgers, addressing the issue of rewind-and-fork attacks. However, they do not support multi-party computation or offline parties, nor ensure fairness.

While attribute-based encryption (ABE) supports ciphertext policies [56] and key policies [57], $\mathcal{F}_{PCFC}$ supports richer history-based policies, which are required in all of our applications, and our policies can be expressed over the plaintext. Moreover, both functional encryption [24] and ABE reveal the plaintext after the decryption or evaluation operations, whereas $\mathcal{F}_{PCFC}$ allows chaining of computations, and enforcing policies on further use of the output. Goyal et al. [58] show how blockchains can implement one time programs using cryptographic obfuscation.

11 CONCLUSION

LucidiTEE enables parties to jointly compute on private data, using protocols (between TEEs and a shared ledger) to ensure that all computations provide fairness and comply with history-based policies, even when any subset of parties act maliciously. The ledger is only used to enforce policies (i.e., it does not store inputs, outputs, or state), letting us scale to large number of parties and large data.

REFERENCES

[1] 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.

[2] 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.

[3] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *STOC*, 1987, pp. 218–229.

[4] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proc. 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[5] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Security Symposium (Security'16)*. Austin, TX: USENIX Association, 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[6] 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.

[7] 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.

[8] 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.

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

[10] C. Dwork, "Differential privacy," in *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006)*, ser. Lecture Notes in Computer Science, vol. 4052. Springer Verlag, July 2006, pp. 1–12. [Online]. Available: https://www.microsoft.com/en-us/research/publication/differential-privacy/

[11] 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.

[12] 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.

[13] 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.

[14] S. D. Gordon, Y. Ishai, T. Moran, R. Ostrovsky, and A. Sahai, "On complete primitives for fairness," in *TCC*, 2010, pp. 91–108.

[15] 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.

[16] 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.

[17] 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.

[18] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *Proceedings of the 23rd USENIX Conference on Security Symposium*. Berkeley, CA, USA: USENIX Association, 2014, pp. 781–796.

[19] 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.

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

[21] "Fintech apps and data privacy: New insights from consumer research," 2018.

[22] A. Gribov, D. Vinayagamurthy, and S. Gorbunov, "Stealthdb: a scalable encrypted database with full sql query support," *arXiv preprint arXiv:1711.02279*, 2017.

[23] 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.

[24] 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.

[25] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2435–2450.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

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

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

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

[43] 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.

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

[45] 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.

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

[47] "The coco framework: Technical overview," https://github.com/Azure/coco-framework/.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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

[55] G. Kaptchuk, I. Miers, and M. Green, "Giving state to the stateless: Augmenting trustworthy computation with ledgers," Cryptology ePrint Archive, Report 2017/201, 2017, https://eprint.iacr.org/2017/201.

[56] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *2007 IEEE symposium on security and privacy (SP'07)*. IEEE, 2007, pp. 321–334.

[57] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180418

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