LucidiTEE: Policy-based Fair Computing at Scale

Rohit Sinha
Visa Research
rohit.sinha@visa.com

Sivanarayana Gaddam
Visa
sgaddam@visa.com

Ranjit Kumaresan
Visa Research
rkumaresan@visa.com

***Abstract*—In light of widespread misuse of personal data, we enable users to control the sharing and use of their data, even when offline, by binding that data to policies. A policy specifies the allowed function, conditions guarding the execution (based on the history of all prior computations on that data), and identities of the input providers and output recipients. For this level of control, we aim for a compute system that ensures *policy compliance* to the input providers, and *fairness* (i.e., either all or no party gets the output) to the output recipients, without requiring these parties to trust each other or the compute host.**

Recently, trusted execution environments (TEEs), such as Intel SGX and Sanctum enclaves, are finding applications in outsourced computing on sensitive data. However, since TEEs are at the mercy of an untrusted host for storage and network communication, they are incapable of enforcing history-dependent policies or fairness. For instance, against a user's wish that only an aggregate function over her entire data is revealed, an adversarial host can repeatedly evaluate that aggregate function on different subsets of her dataset, and learn the individual records. The adversary may also collude and deliver the output only to a subset of the output recipients, thus violating fairness.

This paper presents LucidiTEE, the first system to enable multiple parties to jointly compute on large-scale private data, while guaranteeing that the aforementioned policies are enforced even when the input providers are offline, and guaranteeing fairness to all output recipients. To that end, LucidiTEE develops a set of novel protocols between a network of TEEs and a shared, append-only ledger. LucidiTEE uses the ledger only to enforce policies; it does not store inputs, outputs, or state on the ledger, nor does it duplicate execution amongst the participants, which allows it to scale to large data and large number of parties.

We demonstrate several policy-based applications including personal finance, federated machine learning, fair n-party information exchange, and private set intersection for medical records.

***Index Terms*—Privacy, Policy, Fairness, TEE, Blockchain**

I. Introduction

Modern web services pose a growing public concern regarding their lack of transparency in how they manage sensitive user data. 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 the users) their sensitive data is proliferated, misused, and at times lost to attackers in security breaches.

This work explores a way to govern the processing and sharing of private user data. We study this problem in a general setting where a stateful computation is performed on inputs provided by one or more mutually distrusting parties, who may be offline during the computation and also do not trust the compute provider. Since inputs may be used at various steps of a computation, the offline users need a system for enforcing a *policy*, which specifies: 1) what function is evaluated, 2) whether the evaluation is legal (via a logical condition that depends on the computation's history), and 3) the cryptographic identities of all parties that provide inputs and receive the outputs. The compute system must automatically enforce *policy compliance*, in a *publicly* verifiable way. In addition, for many applications [1], we stress that the compute system must ensure *fairness* (i.e., either all or no parties get the output).

While protocols for multi-party computation [2], [3] ensure that only the agreed-upon function over the inputs is revealed, they require users to be online (or trust one or more third parties to execute the protocol on the user's behalf). Alternatively, recent blockchain platforms for privacy-preserving smart contracts (e.g. Hawk [4], Ekiden [5]) do not naturally allow users to be offline (i.e., inputs cannot be reused across computations without manual interaction), nor scale due to on-chain replication of data and computation. Another alternative is to perform the computation within trusted execution environments (TEEs), which obviates any interaction after the parties provision their inputs to the TEE; in addition to confidentiality and integrity, TEEs implement remote attestation to generate a publicly verifiable proof that the agreed-upon function is (or will be later) performed. Prior works that have this general design include Ryoan [6], VC3 [7], etc.

Since offline users cannot manually approve each step of a computation, the system must automatically enforce policies on how inputs are used. This is beyond the capability of TEEs alone, due to their inherent inability to protect I/O — an adversarial OS (controlled by the compute provider) can rollback the persistent storage and tamper with the network communication. For instance, against a user's wish that only an aggregate function over her dataset is revealed, an adversary can repeatedly evaluate the aggregate function on different subsets of her dataset, and infer information about individual records. Based on this observation, we stress the need for *history-dependent policies*, where compliance is determined with respect to the computation's history, with the associated requirement that each computation is logged before revealing the output. Furthermore, since the OS may deliver the output only to a subset of colluding parties, we need a protocol to ensure fair output delivery, even when all parties are malicious.

To that end, LucidiTEE makes the following contributions:

- definition and construction of a multi-party system for concurrent, stateful computations, with enforcement of history-based policies for offline parties and fairness for all output recipients, in a threat model with arbitrary corruptions.

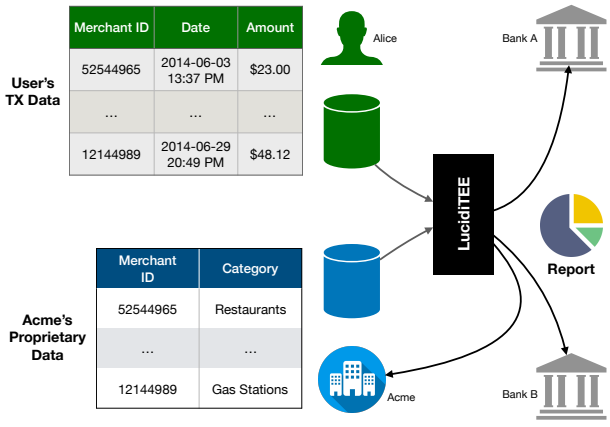


Fig. 1: Privacy-enabled Personal Finance Application

- protocol for fair n-party information exchange, requiring a shared, append-only ledger and $n-1$ parties to own TEEs.
- scalable system that uses the ledger to only enforce policies; by not replicating data or computation on-chain, we address scalability challenges of related blockchain-based systems.

II. Overview of LucidiTEE

A. Motivating Example: Private Personal Finance

The open banking initiative [8] has fostered a variety of third-party financial applications, such as Mint. However, by getting access to raw transaction data, these third party applications pose major concerns of potential breaches and data misuse, as there is lack of (publicly verifiable) transparency on the sharing, use, and retention of this data. While there exists an extreme option to not sign up for any third-party services, LucidiTEE seeks to provide a realistic tradeoff for end users.

Figure 1 illustrates our setting. The user'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. This data can be provided by the bank on behalf of the user (through OAuth-based OFX API [8], for instance), or imported manually by the user. Consider a fictitious financial application, Acme, that maintains a proprietary database mapping merchant ids to category labels, and uses it to provide a service for viewing aggregate spending behavior (i.e., the proportion of spending across categories for all transactions in that month). To perform this joint computation (denoted by the function f), Acme hosts the users' transaction data, and is inevitably trusted by its users to adhere to a privacy policy specified in legalese.

From here on, we discuss a privacy-enabled version of Acme, and a compute system needed by such an application. We iterate on some strawman designs, thus successively motivating the various building blocks used within LucidiTEE.

Privacy

A privacy-seeking user, Alice, wishes for Acme to only learn the output of function f, and nothing else about her input (such as her spending profile at daily granularity or the location patterns). This forms Alice's privacy requirement. For this reason, f cannot be evaluated by sharing Alice's data with Acme, or vice versa as Acme also wishes to maintain secrecy of its proprietary database. In this setting, privacy subsumes correctness in that f must be correctly evaluated lest the computation reveal extra information about the private inputs — note that f is assumed to be safely written, so we only concern ourselves with executing f as given.

To achieve privacy in this 2-party setting (i.e., without a trusted 3rd party), we introduce our first building block: an ideal functionality for secure function evaluation, implemented using secure multi-party computation (MPC) [2], [3] or hardware enclaves [9], [10]. As we see later, To allow participants to go offline, we use hardware enclaves (relying on sealing and attestation primitives) instead of interactive MPC protocols. In this case, the trusted processor is hosted by a *compute provider*, such as Acme or a cloud provider. As a first step, one may start with this strawman design: upon verifying (using remote attestation) that the enclave implements the genuine function f, Alice provisions her input to that enclave over a TLS channel [11], which produces an output encrypted under Alice's and Acme's public keys, and requests the host software to deliver it to them. In the rest of this section, we refine this strawman to support offline participation (which mandates some form of policy enforcement) and fair output delivery.

Offline Participation

In practice, a computation may occur at a later point when Alice is offline, and we cannot expect her to manually provision the input or approve the evaluation of f. Instead, Alice wishes to let Acme compute the approved function f over her transactions — similarly, since Acme's input is a large database comprising millions of merchants, it wishes to not re-transmit the database for each evaluation of f, for each user that it services. To summarize, we must enable participants to go offline after providing their inputs, yet enforce policies on how they are used within stateful, multi-step computation.

To that end, we refine our strawman solution. Instead of requiring Alice to provision her input during the evaluation of f, she (or her bank) provisions transaction records a-priori (rather, streamed to Acme as they are processed by the bank) over an attested TLS channel [12] to an enclave implementing f; Acme also provisions its database to that enclave. The enclave encrypts these records using a sealing key [12] before storing them on untrusted storage, with the hardware-backed guarantee that only that same enclave program can later access the sealing key to decrypt the records (see § III). At a later time (e.g. end of the month), Acme launches this enclave program to evaluate f over all transactions from that month. This refined scheme provides the same privacy guarantee as the earlier scheme, which needed all participants to be online.

History-based Policies

While this strawman ensures that Alice's input is only used for f, we show that this policy alone does not ensure privacy.

The enclave's host software (e.g. the host application, OS, etc.) is controlled by an untrusted party, who may launch multiple instances of the enclave and rollback the persistent storage. Even though the strawman restricts the attacker to evaluate only f on Alice's data, the attacker may repeat the computation with progressively smaller subsets of Alice's transaction data from that month — 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 observing the delta between pairs of output reports, the attacker recovers the category and amount of each transaction, and learns Alice's spending at a finer granularity[1]. Furthermore, when computing with randomness, the attacker uses a similar technique to learn secret inputs or bias outputs. Thus, we strengthen our strawman solution with history-based policies, which check whether an evaluation is legal with respect to that computation's entire history.

To that end, we introduce a second building block: 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 to extract the output — for each function evaluation, the ledger contains digests of the encrypted inputs (along with any random bits), outputs, and intermediate state. Second, we use this ledger to enforce a history-based policy ϕ[2]: all transaction records within the input must be signed by Alice, have the same calendar month, and must be fresh (i.e. they have never been consumed by a prior evaluation of f). In general, we find use of history-based policies in applications that 1) use a privacy budget (e.g. differentially private databases, where the policy ensures freshness of the application's state), and 2) read prior inputs (e.g. authorization logics [13], where access control is determined by prior statements made by the users).

Stateful Computation amongst Multiple Parties

We wish to enable concurrent, (multi-step) stateful computations amongst arbitrary sets of parties. This gives rise to a computation (directed, acyclic) graph containing evaluation of various functions, with inputs and outputs belonging to multiple users. 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*. 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 allow our shared ledger to be universally accessible (based on the set of allowed users), and common to all concurrent computations.

[1]Although additional metadata, such as authenticated batches of inputs, can remedy this attack, Alice's data may be used for other Acme services, and the privacy budget may be expressed collectively over all the computations. Moreover, the application-specific metadata may not be provided by all banks.

[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.

Fairness

To receive lucrative mortgage plans, Alice and Bob wish to have their reports sent automatically to chosen banks; so, in addition to privacy, they seek a fairness guarantee: if any party gets the output, then all honest parties must get the output. In a malicious setting, we can rewrite this as follows: all or none of the parties must get the output. Note that since parties must be online to receive outputs, a computation may specify the output recipients to be disjoint from the set of input providers.

Alice and Bob specify the output recipients to be Acme, Bank A and Bank B, all of whom get the output — since Alice wishes to remain offline, then she must trust one of these parties to present her output in the future (this can be Acme, who is incentivized to retain Alice as a user). Despite malicious behavior, such as collusion between Acme and Bank A, we must guarantee that all output recipients get the output.

As mentioned earlier, we must not trust the enclave's host software, as it is under Acme's control. Since all network communication is proxied via the host software, an enclave cannot ensure that a message is sent over the network — in that sense, an enclave must assume lossy links, which makes reliable message delivery impossible [14]. As a result, fairness cannot be guaranteed using enclaves alone. Therefore, we develop a novel protocol (between a set of enclaves and the shared ledger) to ensure fair delivery to all output recipients.

B. *Personal Finance on* LucidiTEE

LucidiTEE implements a set of protocols (between enclaves and a shared ledger) that ensure fair, policy-compliant computation. It uses a shared ledger to record each computation's specification (which lists f and ϕ), and record each function evaluation within each (stateful) computation. Additionally, as we will see later, LucidiTEE uses various types of enclaves, each fulfilling a distinct role in a computation's life. This subsection illustrates the main ideas using the Acme application.

Managing Computations in LucidiTEE

Alice and Acme agree to the following computation:

```
computation {
  id: 525600, /* unique id */
  in:  [ ("txs":vk_Alice), ("db":vk_Acme) ],
  out: [ ("rprt":[pk_Acme,pk_BankA,pk_BankB])],
  policy: 0xcoff..eeee /* ∀r ∈ txs. fresh(r) */
  func: 0x1337...c0de, /* aggregate function */
}
```

Any party may add this computation's specification to the shared ledger. 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 input and output data structures can have one of several types: file, list, key-value store, etc. The `func` field uniquely identifies the function f using the hash measurement

of the enclave program implementing f. In our example, f is evaluated over the inputs `txs` and `db` (and does not take the ledger as input). Finally, we specify the guard ϕ within the field `policy`. Similar to `func`, `policy` is specified by the hash measurement of the enclave program implementing the predicate ϕ. In our example, ϕ encodes the freshness property that no transaction within `txs` has been consumed by a prior evaluation of f. Unlike f, ϕ takes the entire ledger as input.

Each computation is bound to a unique specification, and it progresses via a potentially unbounded sequence of stateful evaluations of f guarded by ϕ. A computation is said to be *compliant* if all constituent steps use the function f (with measurement `func`) and satisfy ϕ (with measurement `policy`).

Binding Inputs to Computations

Alice must first protect her input such that only policy-compliant computations are performed on it. That is, Alice must encrypt her input `txs` such that it is only decrypted within an enclave to perform a computation based on a specification with desired `id`. To that end, she chooses a key k to encrypt `txs`, and uploads the encrypted data to an untrusted storage (e.g. Acme's server or a cloud storage service). Next, she provisions k and the computation's `id` over a TLS channel terminating within a *key manager enclave* on LucidiTEE (see Figure 2), 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. The key manager enclave's logic only reveals k to an enclave operating on a computation of the same `id`.

Invoking Computation

Acme 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 function evaluations within a computation, and across several computations. 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 a *policy checker enclave* (see Figure 2), which implements 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 the specified hash measurement). On approval from ϕ, Acme launches a *compute enclave*, which implements f, and provides access to the encrypted inputs and state (where the decryption keys are provisioned by the policy checker enclave). f produces the encrypted output and the next state, but the evaluation must be recorded on the ledger before releasing the output.

Since LucidiTEE uses trusted enclaves and an append-only ledger to enforce the policy, any (malicious) compute provider can bring TEE nodes and launch the aforementioned enclaves to store keys, and evaluate ϕ and f. Hence, we emphasize that LucidiTEE embodies a "bring-your-own-compute" paradigm.

Recording Computation

History-based policies necessitate that all function evaluations within a computation are logged. To that end, LucidiTEE implements a protocol (see § V) between the ledger and the compute enclave that ensures atomicity of the following two events: recording the evaluation on the ledger and revealing the output to any party. The ledger record of an evaluation contains 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. By only storing digests on the ledger, we stress that LucidiTEE uses the shared ledger only to enforce policies, and embodies a "bring-your-own-storage" paradigm. This design allows LucidiTEE to scale to large inputs (e.g. § IX-A2). Since the computation is deterministic with respect to the inputs (which includes randomness) and the prior state, a computation can be replayed from the ledger, in case we need to recover from a crash.

Delivering Outputs

We develop a novel protocol for fair n-party message delivery, using enclaves and a shared ledger (see § VI), which is of independent interest beyond LucidiTEE. The protocol withstands a dishonest majority, allowing arbitrary corruption thresholds amongst the $n+1$ parties (containing n output recipients and 1 compute provider), but requires the $n-1$ output recipients to possess a TEE machine. Once the computation is recorded on the ledger, the compute enclave engages in a protocol to deliver the output to all n output recipients, who must be online to participate in the protocol. In our example, the protocol ensures that if any party gets the output from the compute enclave, then all n recipients can get the output, even when the compute provider acts maliciously.

LucidiTEE can be used for one-time programs [15] and fair n-party exchange [16] in a malicious setting (see § IX-A).

III. Preliminaries

A. 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 both SGX and Sanctum, the CPU monitors all memory accesses to ensure that non-enclave software (including OS, Hypervisor, and BIOS / SMM / UEFI firmware) cannot access the enclave's memory — SGX also prevents hardware attacks on DRAM by encrypting and integrity-protecting the enclave's cache lines before writing them to DRAM. LucidiTEE assumes that the TEE platform provides the *secure remote execution* guarantee defined by Subramanyan et al. [10].

In addition to isolated execution, we assume that the TEE platform provides primitives for remote attestation (for generating proofs attesting to the code identity (hash-based measurement) of the enclave), and sealed storage (where the sealing key is derived from the processor's secret keys and the enclave's code identity). At any time, the enclave software may request a signed message (called a *quote*) binding an enclave-supplied value to that enclave's hash-based measurement.

We model the TEE hardware as an ideal functionality HW, adapted from [17], which maintains the internal state of each loaded enclave in variable st, and has the following interface:

- HW.CreateMachine() returns mid, an identifier for a new instance of a TEE machine.
- HW.Load(mid, prog) loads the enclave program prog on machine mid, and returns a unique id eid for that loaded program. Additionally, the internal state is initialized st[eid] = $\overrightarrow{0}$. We omit the argument mid when it is not relevant.
- HW.Run(eid, in) executes enclave eid (from prior state st[eid]) under adversarial input in, and produces an adversary-visible output out while also updating st[eid].
- HW.Quote(data) must be invoked within an enclave program and it returns quote $\doteq (\mu(\mathsf{prog}), \mathsf{data}, \sigma)$, thus binding the enclave-provided data to its hash-based measurement $\mu(\mathsf{prog})$. Here, σ is a signature over $\mu(\mathsf{prog}) \parallel \mathsf{H}(\mathsf{data})$, proving that the data originated from an enclave with measurement $\mu(\mathsf{prog})$ running on a genuine TEE machine.
- HW.QuoteVerify(quote) verifies the signature σ within the quote, using the TEE manufacturer's attestation service.
- HW.GenRnd(n) returns n cryptography-grade random bits.
- HW.GenSealKey() must be invoked within an enclave program, and it returns a key derived from the machine's secret and the enclave's measurement. This ensures unsealing of data, in that future calls to HW.GenSealKey() from the same prog on the same machine mid produces the same key.

We assume unforgeability of the remote attestation signature scheme, as defined in [17]. Moreover, we assume ideal enclaves without any side channels — the compute enclave only reveals the output of f, and the policy checking enclave only reveals the decision bit from evaluating ϕ. This assumption is discharged in part by using safer enclave processors, such as RISC-V Sanctum, which has defenses for several hardware side channels. Furthermore, the developer can compile f and ϕ using static and dynamic software defenses (e.g., [18], [19], [20], [21], [22]), for eliminating software side channels such as timing leaks and access patterns.

B. Shared, Append-only Ledger

We borrow the bulletin board abstraction of a shared ledger, defined in [16], 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 implements an append-only ledger. The shared, append-only ledger L is modeled as an ideal functionality, 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 $\mathsf{Verify}_\mathsf{L}(\sigma, \mathsf{t}\|\mathsf{e}) = \mathsf{true}$. Note that $\mathsf{Verify}_\mathsf{L}$ is a public verification algorithm that can be run locally at any party (i.e. without access to the ledger).
- 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.

The bulletin board abstraction is implemented by forkless blockchains, such as permissioned blockchains [23], and potentially even by blockchains based on proof-of-stake [24].

C. Cryptographic Primitives and Assumptions

a) Hash Function: We use a hash function H (e.g. SHA-256) that is collision-resistant and pre-image resistant.

b) Public Key Encryption: We assume a *IND-CCA2* [25] public key encryption scheme PKE (e.g. RSA-OAEP) consisting of the following polynomial-time algorithms:

- $\mathsf{PKE.Keygen}(1^\lambda)$ is a randomized algorithm for generating a key pair (pk,sk) based on security parameter λ.
- PKE.Enc(pk, m) is an encryption algorithm which takes a public key pk and a message m, and outputs a ciphertext ct.
- PKE.Dec(sk, ct) is a deterministic decryption algorithm which takes a secret key sk and a ciphertext ct, and outputs the decrypted message m (if $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{m}) \wedge (\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$) or $\perp$ (denoting error).

c) Digital Signature Scheme: We assume a *EUF-CMA* [26] digital signature scheme S (e.g. ECDSA scheme) consisting of the following polynomial-time algorithms:

- $\mathsf{S.Keygen}(1^\lambda)$ is a randomized algorithm for generating a key pair (sk,vk) based on security parameter λ.
- S.Sig(sk, m) is a signing algorithm which takes a signing key sk and message m, and outputs a signature σ.
- S.Verify(vk, σ, m) is a verification algorithm which outputs true when $\sigma \leftarrow \mathsf{S.Sig}(\mathsf{sk}, \mathsf{m}) \wedge (\mathsf{sk},\mathsf{vk}) \leftarrow \mathsf{S.Keygen}(1^\lambda)$.

d) Symmetric Key Encryption: We assume a scheme for authenticated encryption with associated data AEAD (e.g. AES-GCM) that provides both *IND-CPA* and *INT-CTXT* [27]. The scheme consists of following polynomial-time algorithms:

- $\mathsf{AEAD.Keygen}(1^\lambda)$ returns a uniformly random $\mathsf{k} \in \{0,1\}^\lambda$.
- AEAD.Enc(k, m, ad) is an encryption algorithm which takes a symmetric key k, a message m, and (optional) associated data ad, and outputs a ciphertext ct. The associated data ad is authenticated but not included within the ciphertext ct.
- AEAD.Dec(k, ct, ad) is a decryption algorithm which takes a symmetric key k and ciphertext ct, and outputs the message m (if $\mathsf{ct} \leftarrow \mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{m}, \mathsf{ad})$) or $\perp$ (error).

IV. LucidiTEE Specification

A. 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 $\mathcal{P}^*$, of which any subset of parties may engage in a computation. Each computation involves a set of input providers $\{\mathcal{P}_i^1, \ldots, \mathcal{P}_i^m\}$ and a set of output recipients $\{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n\}$, which may overlap, such that $(\{\mathcal{P}_i^1, \ldots, \mathcal{P}_i^m\} \cup \{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n\}) \subseteq \mathcal{P}^*$.

We assume a polynomial-time adversary that corrupts any subset $\mathcal{P}^\mathcal{A} \subseteq \mathcal{P}^*$, whose members behave maliciously and deviate arbitrarily from the protocol. The attacker selects the inputs and learns the output of each party in $\mathcal{P}^\mathcal{A}$. The honest

parties output the computation result given to them by the ideal functionality, whereas the malicious parties output an arbitrary polynomial-time function of the adversarial view.

B. Ideal Functionality

We introduce an ideal functionality, $\mathcal{F}_{PCFC}$, for policy-compliant fair computations amongst multiple parties. A computation is modeled as a state transition system, where each step evaluates a transition function f if the history-dependent policy ϕ (expressed over the inputs, state, and the computation's history) is satisfied. Each computation is defined by a unique specification c, which fixes f, ϕ, and the identities of the input providers $\{\mathcal{P}_i^1, \ldots, \mathcal{P}_i^m\}$ and the output recipients $\{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n\}$. Since input providers may go offline after binding their input to a computation c, they are not required to authorize each step of c — an input may be used for an unbounded number of steps of c, as long as $\mathsf{c}.\phi$ approves each step and c has not been revoked by an input provider of c. Moreover, inputs may also be bound to several computations, avoiding duplication of the data. $\mathcal{F}_{PCFC}$ is defined below[3]:

POLICY COMPLIANT FAIR COMPUTING: $\mathcal{F}_{PCFC}$

$\mathcal{F}_{PCFC}$ services parties in $\mathcal{P}^*$ by performing one of the following commands. Moreover, it provides an entropy source, and maintains private `store` and public `ldgr` ($\mathcal{P}^*$ and $\mathcal{A}$ can read, not write).
Let $\mathsf{active}(\mathsf{c}) \doteq (\texttt{create}\|\mathsf{c}) \in \texttt{ldgr} \wedge (\texttt{revoke}\|\mathsf{c}.\texttt{id}) \notin \texttt{ldgr}$

- On **command** create_computation(c) from $\mathcal{P} \in \mathcal{P}^*$
  send $(\texttt{create} \parallel \mathsf{c} \parallel \mathcal{P})$ to $\mathcal{A}$
  if $\exists \hat{\mathsf{c}}$ $(\texttt{create}\|\hat{\mathsf{c}}) \in \texttt{ldgr} \wedge \hat{\mathsf{c}}.\texttt{id} = \mathsf{c}.\texttt{id}$ { **ret** $\perp$ to $\mathcal{P}$ }
  $\texttt{ldgr}$.append$(\texttt{create}\|\mathsf{c})$; **ret** $\top$ to $[\mathcal{P}, \mathcal{A}]$
- On **command** revoke_computation(c) from $\mathcal{P} \in \mathsf{c}.\texttt{in}$
  send $(\texttt{revoke} \parallel \mathsf{c} \parallel \mathcal{P})$ to $\mathcal{A}$
  if $\neg$ active(c) { **ret** $\perp$ to $[\mathcal{P}, \mathcal{A}]$ }
  $\texttt{ldgr}$.append$(\texttt{revoke}\|\mathsf{c}.\texttt{id})$; **ret** $\top$ to $[\mathcal{P}, \mathcal{A}]$
- On **command** provide_input(i) from $\mathcal{P} \in \mathcal{P}^*$
  $\texttt{store}[h] := (i, \mathcal{P}, \emptyset)$, where $h \xleftarrow{\$} \mathbb{N}$ is a fresh id
  send $(\texttt{provide\_input} \parallel | \, i \, | \parallel h \parallel \mathcal{P})$ to $\mathcal{A}$; **ret** h to $\mathcal{P}^*$
- On **command** bind_input(c, h) from $\mathcal{P} \in \mathsf{c}.\texttt{in}$
  send $(\texttt{bind\_input} \parallel \mathcal{P})$ to $\mathcal{A}$
  if $(\neg\mathsf{active}(\mathsf{c}) \vee \texttt{store}[h] = \perp)$ { **ret** $\perp$ to $[\mathcal{A}, \mathcal{P}]$ }
  $(x, \mathcal{P}', \mathsf{C}) \leftarrow \texttt{store}[h]$
  if $(\mathcal{P}' \neq \mathcal{P})$ { **ret** $\perp$ to $[\mathcal{A}, \mathcal{P}]$ }
  $\texttt{store}[h] := (x, \mathcal{P}, \mathsf{C} \cup \mathsf{c})$; **ret** $\top$ to $[\mathcal{A}, \mathcal{P}]$
- On **command** compute(c, h_s, $h_i^1, \ldots, h_i^m$) from $\mathcal{P} \in \mathcal{P}^*$
  send $(\texttt{compute} \parallel \mathsf{c} \parallel h_s \parallel h_i^1, \ldots, h_i^m \parallel \mathcal{P})$ to $\mathcal{A}$
  if $\neg$active(c) or $\mathcal{A}$ denies then { **ret** $\perp$ to $[\mathcal{A}, \mathcal{P}]$ }
  let $(i_x, \mathcal{P}_x, \mathsf{C}_x) \leftarrow \texttt{store}[h_i^x]$, for $x \in [1 \ldots m]$
  let $(s, \_, \_) \leftarrow \texttt{store}[h_s]$; let $r \xleftarrow{\$} \{0,1\}^{128}$
  let $b_1 \leftarrow (\forall x \in [1 \ldots m].\ i_x \neq \perp \wedge \mathcal{P}_x = \mathcal{P}_i^x \wedge \mathsf{c} \in \mathsf{C}_x) \wedge s \neq \perp$
  let $b_2 \leftarrow \mathsf{c}.\phi(\texttt{ldgr}, h_s, s, h_i^1, i_1, \ldots, h_i^m, i_m)$
  if $(\neg b_1 \vee \neg b_2)$ then { **ret** $\perp$ to $[\mathcal{A}, \mathcal{P}]$ }
  let $(s', o_1, \ldots, o_n, o_{\mathsf{c}_1}, \ldots, o_{\mathsf{c}_t}) \leftarrow \mathsf{c}.f(s, i_1, \ldots, i_m, r)$
  for each output o_x, let $h_o^x \xleftarrow{\$} \mathbb{N}$ in $\texttt{store}[h_o^x] := (o_x, \perp, \perp)$
  for each computation c_x, let $h_{\mathsf{c}_x} \xleftarrow{\$} \mathbb{N}$ in $\texttt{store}[h_{\mathsf{c}_x}] := (o_{\mathsf{c}_x}, \perp, \mathsf{c}_x)$
  for next state s', let $h_{s'} \xleftarrow{\$} \mathbb{N}$ in $\texttt{store}[h_{s'}] := (s', \perp, \mathsf{c})$
  $\texttt{ldgr}$.append$(\texttt{compute}\|\mathsf{c}.\texttt{id}\|h_{s'}, h_i^1 \ldots h_i^m, h_o^1 \ldots h_o^n, h_{\mathsf{c}_1} \ldots h_{\mathsf{c}_t})$
  if $\mathcal{A}$ ok { for $x \in [1 \ldots n]$ **send** o_x to $\mathcal{P}_o^x$ }
  send $\; | \, o_1 \, | \ldots | \, o_n \, |, | \, o_{\mathsf{c}_1} \, | \ldots | \, o_{\mathsf{c}_t} \, |, | \, s' \, |$ to $\mathcal{A}$; **ret** $\top$ to $\mathcal{P}$

$\mathcal{F}_{PCFC}$ maintains a publicly readable log `ldgr` and private storage `store`. `store` provides protected storage of inputs, outputs, and computational state, and is indexed by unique handles (producing $\perp$ if the mapping does not exist). `ldgr` is an append-only ledger recording all function evaluations, and the creation and revocation of all computations. Since the specification c does not contain secrets, it can be created (using the command create_computation) by any party in the universe $\mathcal{P}^*$. A computation can be revoked explicitly (using the command revoke_computation) by any party listed as an input provider in $\mathsf{c}.\texttt{in}$, preventing future evaluations of $\mathsf{c}.f$.

A party $\mathcal{P}$ uploads an input i (using provide_input), which $\mathcal{F}_{PCFC}$ persists internally and returns a unique handle h — at this point, i is not bound to any computation. Next, using bind_input, $\mathcal{P}$ binds h to a computation c, allowing that input i to be consumed by $\mathsf{c}.f$, without $\mathcal{P}$ having to resupply i on each evaluation of $\mathsf{c}.f$ (though each such evaluation must comply with $\mathsf{c}.\phi$). $\mathcal{P}$ may bind i to multiple computations concurrently, as long as each of them list $\mathcal{P}$ as an input provider. We find that these characteristics make $\mathcal{F}_{PCFC}$ suitable for settings where users make dynamic decisions to participate in new computations and become offline after providing their inputs, or when computing over large inputs, or in applications that provide a common service to many users (e.g. Acme).

We allow any party $\mathcal{P} \in \mathcal{P}^*$ to invoke a function evaluation (using compute), as only policy-compliant evaluations succeed. compute is invoked with handles pointing to the inputs and the prior state. $\mathcal{P}$ can provide any handles of her choice, as $\mathcal{F}_{PCFC}$ checks the guard $\mathsf{c}.\phi$ prior to evaluating $\mathsf{c}.f$, in addition to some sanity checks that the inputs are existent and bound to the right computation c. Note that $\mathsf{c}.f$ produces $o_1, \ldots, o_n$ for the n output recipients, and $o_{\mathsf{c}_1}, \ldots, o_{\mathsf{c}_t}$ for the chained computations — outputs for chained computations are not owned by any party, and thus cannot be bound to other computations. Finally, before delivering the output, $\mathcal{F}_{PCFC}$ records the evaluation on the `ldgr`, including all the relevant handles. The adversary $\mathcal{A}$ may prevent $\mathcal{F}_{PCFC}$ from sending the output; however, should any party get the output, then all recipients $\mathcal{P}_o^1 \ldots \mathcal{P}_o^n$ listed in $\mathsf{c}.\texttt{out}$ get the output. $\mathcal{F}_{PCFC}$ guarantees the following properties for each computation:

* $\mathcal{A}$ does not learn an honest party's input, beyond its size and the function evaluations which have used that input.
* In any computation c, $\mathsf{c}.f$ is evaluated only if the policy $\mathsf{c}.\phi$ is satisfied (given the state, inputs, and history of c).
* $\mathcal{A}$ learns the outcome of evaluating ϕ, and learns the outcome of f only if it controls a party in $\{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n\}$.
* Parties in $\{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n\}$ get the same output with fairness.

Fair reactive computation is out of scope, since $\mathcal{A}$ can prevent executing the compute command and computations are revocable. For brevity, we omit the command for crash recovery: resuming a computation that is recorded on `ldgr`, but has not yet delivered the outputs. $\mathcal{F}_{PCFC}$ allows "replaying" from the `ldgr` to retry sending the output to all parties.

V. POLICY-COMPLIANT COMPUTATION

LucidiTEE realizes $\mathcal{F}_{PCFC}$, assuming ideal functionalities for secure hardware HW and shared, append-only ledger L. Figure 2 illustrates the primary components of LucidiTEE. Each entry on the shared, public ledger records either the

[3]The **ret** statement performs a **send** and also terminates the command.

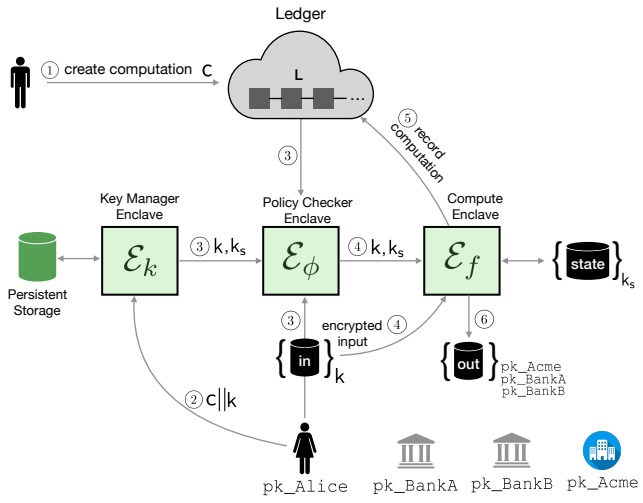


Fig. 2: Policy enforcement using enclaves and a shared ledger

creation or revocation of a computation (containing the specification c), 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, following the "bring-your-own-storage" paradigm, 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 $\mathsf{c}.f$ is compliant with $\mathsf{c}.\phi$); 3) a compute enclave $\mathcal{E}_f$ (responsible for evaluating $\mathsf{c}.f$, delivering the outputs, 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* (following the "bring-your-own-compute" paradigm), yet our protocols guarantee policy compliance and fairness to all parties. Note that the computation is only performed by one party. This section describes the phases of a computation, from protecting inputs to checking policies, and then on to executing and recording the computation on the ledger.

A. Specifying and Creating a Computation

A computation is defined by the function f, policy ϕ, and identities of the input providers and output recipients. A computation's specification c is a string with following syntax:

$$
\begin{array}{llll}
\mathit{Id} & \mathsf{z} & ::= & \{0,1\}^{64} \\
\mathit{Name} & \mathsf{n} & ::= & [\mathsf{a}-\mathsf{zA}-\mathsf{Z0}-\mathsf{9}]+ \\
\mathit{Key} & \mathsf{k} & ::= & \{0,1\}^* \\
\mathit{Input} & \mathsf{i} & ::= & (\mathsf{n}:\mathsf{k}) \mid (\mathsf{n}:(\mathsf{z},\mathsf{n})) \\
\mathit{Output} & \mathsf{o} & ::= & (\mathsf{n}:[(\mathsf{k},\mathsf{k}),\ldots,(\mathsf{k},\mathsf{k})]) \mid (\mathsf{n}:(\mathsf{z},\mathsf{n})) \\
\mathit{State} & \mathsf{s} & ::= & \mathsf{n} \\
\mathit{Hash} & \mathsf{h} & ::= & \{0,1\}^* \\
\mathit{Comp} & \mathsf{c} & ::= & \texttt{computation}\ \{ \\
 & & & \quad \texttt{id}:\mathsf{z},\ \texttt{func}:\mathsf{h},\ \texttt{policy}:\mathsf{h}, \\
 & & & \quad \texttt{in}:[\mathsf{i},\ldots,\mathsf{i}],\ \texttt{out}:[\mathsf{o},\ldots,\mathsf{o}], \\
 & & & \quad \texttt{state}:[\mathsf{s},\ldots,\mathsf{s}]\ \}
\end{array}
$$

A computation is identified by a unique 64-bit number z. A data structure such as an input, output, or state is named by a alphanumeric constant n. A key k is a finite length binary string, and it represents either an encryption key (e.g. RSA public key) or a signature verification key (e.g. ECDSA public key). An input is denoted by the tuple $(\mathsf{n}:\mathsf{k})$, containing its name n and the signature verification key k of the input provider. Similarly, an output is denoted by the tuple $(\mathsf{n}:[(\mathsf{k},\mathsf{k}),\ldots,(\mathsf{k},\mathsf{k})])$, containing its name n along with the list of key pairs (encryption and signature keys) 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 $(\mathsf{n}:(\mathsf{z},\mathsf{n}))$, where z is the destination computation's identifier and the second 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 as a string with a set of attributes: an `id`, hash of $\mathcal{E}_f$ (`func`) implementing function f, hash of $\mathcal{E}_\phi$ (`policy`) implementing ϕ, input description `in`, output description `out`, and state variables `state`.

A party $\mathcal{P}$ can create a new multi-party computation using the create_computation command in $\mathcal{F}_{PCFC}$, which LucidiTEE implements by having $\mathcal{P}$ execute the following:

$$\mathcal{P} \to \mathcal{E}_k:\ \mathsf{c}\|\sigma,\ \text{where}\ (\sigma,\mathsf{t}) = \mathsf{L.post}(\texttt{create} \parallel \mathsf{c})$$

Here, having posted the specification c on the ledger, $\mathcal{P}$ contacts the key manager enclave $\mathcal{E}_k$, who generates a key k_s used to protect the computation's state in all future function evaluations. Since c does not contain any secrets, any party can create a computation on the ledger. It is up to the input providers to examine c and choose to bind their inputs to it.

Any party $\mathcal{P} \in \mathsf{c}.\texttt{in}$ that is listed as an input provider can revoke a computation using the revoke_computation command, which LucidiTEE implements by having $\mathcal{P}$ execute:

$$\mathcal{P}:\ \mathsf{L.post}(\texttt{revoke} \parallel \mathsf{c}.\texttt{id})$$

Since we let parties revoke computations, we remark again that LucidiTEE (and $\mathcal{F}_{PCFC}$) do not provide fair reactive computation; however, fairness is guaranteed on output delivery.

B. Protecting Inputs, Outputs, and State

The input providers may go offline after providing their inputs, meaning they must be stored temporarily at an untrusted storage provider[4]. In addition to the inputs, we must protect the intermediate state of computations and their outputs, as they are handled by the untrusted compute provider. Hence, LucidiTEE must cryptographically protect data with confidentiality, integrity, and authenticity guarantees. We outline the cryptographic scheme used within LucidiTEE's implementation of the provide_input command — the scheme is executed locally by the input providers to protect inputs, and within the

[4]The attacker can always delete the inputs, and it is equivalent to denying the compute command; as before, this only violates fair reactive computation.

compute enclave $\mathcal{E}_f$ to protect state and outputs — producing encrypted, authenticated data structures (uniquely identified by hash digests) that are stored locally or on an untrusted storage.

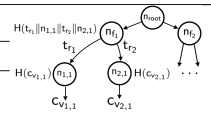
Encryption: Inputs, outputs, and state in LucidiTEE has one of several data types: list, table, or unstructured file; new types can be added as needed. Table I outlines the scheme for encrypting data, where the use of authenticated encryption (with associated data) ensures confidentiality and integrity.

TABLE I: Encryption of datatypes in LucidiTEE

Plaintext Value	**Encrypted Value**
List $[\mathsf{v}_1, \ldots, \mathsf{v}_z]$	**List** $[\mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{v}_1), \ldots, \mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{v}_z)]$
Tbl f_1 ... r_1 $\mathsf{v}_{1,1}$ ... r_2 $\mathsf{v}_{2,1}$ ...	**Tbl** $\mathsf{MAC}(\mathsf{k}, \mathsf{f}_1)$ ... $\mathsf{MAC}(\mathsf{k}, \mathsf{r}_1)$ $\mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{v}_{1,1}, \mathsf{r}_1 \Vert \mathsf{f}_1)$ ... $\mathsf{MAC}(\mathsf{k}, \mathsf{r}_2)$ $\mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{v}_{2,1}, \mathsf{r}_2 \Vert \mathsf{f}_1)$ ...
File $\mathsf{x} \in \{0,1\}^*$	**File** $\mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{x})$

Authentication: While the use of AEAD in Table I gives integrity and confidentiality at the record-level, it still lets the attacker delete and reorder records (within the list and table structures). We turn towards authenticated data structures, such as hash chains and Merkle tries, to defend against these attacks, but we must first discuss few practical requirements. First, the authenticated structures must support efficient appends, to allow participants to efficiently produce new input records. For instance, Alice's bank should be able to upload new transactions without significant computation (e.g. without having to process previous transaction records). Second, the authenticated structure must support efficient updates, while protecting against rollback attacks to older versions; this would allow compute enclaves, for instance, to efficiently perform in-place updates in large databases. Authenticated data structures is an orthogonal research problem [28], and we borrow existing constructions. Table II lists the transformation rules, applied to the encrypted values produced by Table I.

TABLE II: Authentication of datatypes in LucidiTEE

Encrypted Value	**Authenticated, Encrypted Value**	**Digest** δ
List $[\mathsf{c}_{\mathsf{v}_1}, \ldots, \mathsf{c}_{\mathsf{v}_z}]$	$[\mathsf{c}_{\mathsf{v}_1} \Vert \mathsf{H}(\vec{0}), \ldots, \star]$, $\star \doteq \mathsf{c}_{\mathsf{v}_z} \Vert \mathsf{H}(\mathsf{c}_{\mathsf{v}_{z-1}})$	$\mathsf{H}(\star)$
Tbl $\mathsf{t}_{\mathsf{f}_1}$ ... $\mathsf{t}_{\mathsf{r}_1}$ $\mathsf{c}_{\mathsf{v}_{1,1}}$ ... $\mathsf{t}_{\mathsf{r}_2}$ $\mathsf{c}_{\mathsf{v}_{2,1}}$ ...	**Tbl** $\mathsf{t}_{\mathsf{f}_1}$ ... $\mathsf{t}_{\mathsf{r}_1}$ $\mathsf{c}_{\mathsf{v}_{1,1}}$ ... $\mathsf{t}_{\mathsf{r}_2}$ $\mathsf{c}_{\mathsf{v}_{2,1}}$ ... 	$\mathsf{H}(\mathsf{n}_{\mathsf{root}})$
File c_x	$\mathsf{c}_\mathsf{x} \Vert \mathsf{H}(\mathsf{c}_\mathsf{x})$	$\mathsf{H}(\mathsf{c}_\mathsf{x})$

Lists are authenticated using a hash chain. Tables are authenticated using a Merkle trie, where the row identifier r_i acts as the keys in the trie structure. A file is authenticated by building a Merkle tree over its constituent blocks (or a hash over the entire encrypted file, in case of small files). LucidiTEE uses succinct, collision-resistant digests δ (see Table II) to uniquely identify and cryptographically authenticate the data. These digests play the role of handles in $\mathcal{F}_{PCFC}$.

We write δ_i, δ_o, and δ_s to denote digests of the input, output, and state values, respectively. As we see later, digests serve two purposes. First, LucidiTEE records each computation on the ledger (writing the digests of all inputs, outputs, and the next state), and uses those digests to enforce policies in all future computations. Second, an input provider can use compute chaining to force the digest of an input that a computation must use, by using a separate computation whose only purpose is to record the desired digest on the ledger.

C. Binding Inputs to Computations

Recall the bind_input command in $\mathcal{F}_{PCFC}$, used to protect a party's input such that it is only accessed within those computations that are approved by that party. By encrypting the input, LucidiTEE reduces this problem to ensuring that the encryption key is only provisioned to enclaves identified within the computation's specification ($\mathsf{c}.\phi$ and $\mathsf{c}.\mathsf{f}$). We refer to this as binding a key k to a computation c, and it is carried out via a protocol between the input provider $\mathcal{P}$ and the compute provider (running the key manager enclave $\mathcal{E}_k$):

$$\mathcal{E}_k \to \mathcal{P} : \mathsf{HW.Quote}(\mathsf{pk}), \text{where } (\mathsf{pk}, \_) \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$$
$$\mathcal{P} \to \mathcal{E}_k : \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{c.id} \Vert \mathsf{k} \Vert \mathsf{n} \parallel \mathsf{S.Sig}(\mathsf{sk}_\mathcal{P}, \mathsf{c.id} \Vert \mathsf{k} \Vert \mathsf{n}))$$

Here, $\mathcal{P}$ (with signing key $\mathsf{sk}_\mathcal{P}$) contacts the $\mathcal{E}_k$, which produces a fresh public key pk along with a quote attesting the genuineness of $\mathcal{E}_k$. Upon verifying the attestation quote, $\mathcal{P}$ signs and encrypts $\mathsf{c.id}$ and k, along with n which specifies the name of the input (from the list $\mathsf{c.in}$). Following the above interaction, $\mathcal{E}_k$ stores k locally, using the TEE's sealing feature, by writing $\mathsf{AEAD.Enc}(\mathsf{HW.GenSealKey}(), \mathsf{k} \Vert \mathsf{n}, \mathsf{c})$ to a persistent storage — later, $\mathcal{E}_k$ will only reveal k to an enclave that is computing on c, either for evaluating $\mathsf{c}.\phi$ or $\mathsf{c}.f$.

By binding keys (rather than raw inputs) to computations, LucidiTEE enables reuse of inputs across function evaluations and across computations, without having to duplicate the data.

D. Invoking Computation

LucidiTEE allows a computation to succeed only if it is compliant and executed within a genuine enclave. That is, any compute provider $\mathcal{P}_c \in \mathcal{P}^*$ (not necessarily an input provider or an output recipient listed in c) can launch enclaves $\mathcal{E}_\phi$ and $\mathcal{E}_f$ to perform the computation, thus following the "bring-your-own-compute" paradigm. First, $\mathcal{P}_c$ downloads the encrypted state and inputs[5] (from untrusted storage) for all the input providers listed in $\mathsf{c.in}$. Next, $\mathcal{P}_c$ launches $\mathcal{E}_\phi$, which contacts $\mathcal{E}_k$ in order to decrypt parts of the inputs and the prior state (for evaluating ϕ). The protocol executes as follows:

$$\mathcal{E}_\phi \to \mathcal{E}_k : \mathsf{HW.Quote}(\mathsf{c} \Vert \mathsf{pk}), \text{where } \mathsf{pk}, \_ \leftarrow \mathsf{PKE.Keygen}(1^\lambda)$$
$$\mathcal{E}_k \to \mathcal{E}_\phi : \mathsf{HW.Quote}(\mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{k}_s \parallel \mathsf{k}_i^1 \Vert \ldots \Vert \mathsf{k}_i^m))$$

$\mathcal{E}_\phi$ uses k_s to decrypt the state s, and keys $\mathsf{k}_i^1, \ldots, \mathsf{k}_i^m$ to decrypt all inputs $i_1, \ldots, i_m$. Since LucidiTEE must scale to large data and enclaves have limited physical memory, the encrypted inputs and state are placed in non-enclave memory, and the accessed bits are authenticated, integrity-checked, and decrypted on-demand within the enclave's protected memory.

[5]To amortize this overhead (e.g. inference using a trained model), $\mathcal{P}_c$ may locally store the encrypted inputs and state to avoid re-downloading them.

E. Policy Compliance Checking

$\mathcal{E}_\phi$ must check whether the requested evaluation of $\mathsf{c}.f$ is compliant with $\mathsf{c}.\phi$, which requires checking the following:

1) Computation c must have been added to the ledger L, and not revoked at any later time by any party $\mathcal{P} \in \mathsf{c}.\texttt{in}$.
2) Policy condition ϕ (evaluated over the ledger L, inputs $i_1, \ldots, i_m$, and the state s) must be satisfied.

To perform these checks, $\mathcal{P}_c$ must provide $\mathcal{E}_\phi$ with a view to L, potentially by downloading the ledger's contents locally. Observe that the enclave-ledger interaction happens via the untrusted host software controlled by $\mathcal{P}_c$; hence, an adversarial $\mathcal{P}_c$ may present a stale view (a prefix) of L to $\mathcal{E}_\phi$ — we deal with this attack in § V-G. For now, we task ourselves with deciding compliance with respect to a certain height of L.

Our scheme roughly works as follows. To implement the first check, we scan the ledger for `create` and `revoke` entries on $\mathsf{c}.\texttt{id}$. Here, $\mathcal{E}_\phi$ asserts validity of the predicate:

$$\begin{aligned}
&\exists \mathsf{t}_1.\ (\ (\sigma, \mathsf{e}) = \mathsf{L}.\mathsf{getContent}(\mathsf{t}_1) \ \wedge \\
&\qquad \mathsf{Verify}_{\mathsf{L}}(\sigma, \mathsf{t}_1 \| \mathsf{e}) \ \wedge\ \mathsf{e} = \texttt{create} \parallel \mathsf{c}\ ) \ \wedge \\
&\forall \mathsf{t}_2.\ \neg\ (\mathsf{t}_2 > \mathsf{t}_1 \wedge (\sigma, \mathsf{e}) = \mathsf{L}.\mathsf{getContent}(\mathsf{t}_2) \ \wedge \\
&\qquad \mathsf{Verify}_{\mathsf{L}}(\sigma, \mathsf{t}_2 \| \mathsf{e}) \ \wedge\ \mathsf{e} = \texttt{revoke} \parallel \mathsf{c}.\texttt{id}\ )
\end{aligned}$$

To implement the second check, we evaluate $\mathsf{c}.\phi$ within $\mathcal{E}_\phi$, over the requested computation's inputs and previous state, along with the following sequence of relevant entries in L:

$$\begin{aligned}
&[\ \mathsf{e} \mid \mathsf{t}^* \in \{0 \ldots \mathsf{t}\} \ \wedge\ \mathsf{t} = \mathsf{L}.\mathsf{getCurrentCounter} \ \wedge \\
&\qquad (\sigma, \mathsf{e}) = \mathsf{L}.\mathsf{getContent}(\mathsf{t}^*) \wedge\ \mathsf{Verify}_{\mathsf{L}}(\sigma, \mathsf{t}^* \| \mathsf{e}) \ \wedge \\
&\qquad \mathsf{e} = \texttt{compute} \parallel \mathsf{c}.\texttt{id} \parallel \ \ldots\ ]
\end{aligned}$$

As stated before, ϕ is an arbitrary function, ultimately implemented in machine code with the enclave hash measurement $\mathsf{c}.\texttt{policy}$. 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 (as the computation's state) the highest timestamp amongst the input `txs`, and implement ϕ to inspect the last entry (call this e) from the aforementioned list of ledger entries $[\mathsf{e} \mid \ldots]$. To evaluate ϕ, we ask $\mathcal{P}_c$ to load the c's state (which must produce the digest δ_s recorded in e), and transfer control to ϕ, which takes each transaction record in the current input and compares its timestamp with e's state. Alternatively, ϕ can also load the inputs from all prior computations (i.e., all of $[\mathsf{e} \mid \ldots]$) and check membership of each record in the current input; 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.

In the design so far, the ledger L is naively stored as a sequence of entries, which would force us to perform a linear scan for evaluating the two compliance checks. Instead, our implementation stores L locally as an authenticated key-value database, whose index is the computation's id $\mathsf{c}.\texttt{id}$. Creation of specification c on the ledger inserts a new entry at index $\mathsf{c}.\texttt{id}$, and revocation marks it invalid. Each computation appends the ledger entry to the current value at $\mathsf{c}.\texttt{id}$. Now, instead of scanning through the entire ledger, the first compliance check asserts the presence of key $\mathsf{c}.\texttt{id}$, while the second check reads the list of records at key $\mathsf{c}.\texttt{id}$. Note that this optimization does not impact security — $\mathcal{E}_\phi$'s view of L is still controlled by $\mathcal{P}_c$, and therefore potentially stale.

F. Performing the Computation

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

$$\begin{aligned}
&\mathcal{E}_f \to \mathcal{E}_\phi : \mathsf{HW.Quote}(\mathsf{c} \| \mathsf{pk}), \text{where } \mathsf{pk}, _ \leftarrow \mathsf{PKE.Keygen}(1^\lambda) \\
&\mathcal{E}_\phi \to \mathcal{E}_f : \mathsf{HW.Quote}(\mathsf{t} \parallel \hat{\mathsf{k}} \parallel \delta_s \parallel \delta_i^1, \ldots, \delta_i^m) \\
&\qquad \text{where } \mathsf{t} \doteq \mathsf{L}.\mathsf{getCurrentCounter} \text{ used in } \mathcal{E}_\phi \\
&\qquad \text{where } \hat{\mathsf{k}} \doteq \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{k}_s \parallel \mathsf{k}_i^1 \parallel \ \ldots\ \| \mathsf{k}_i^m)
\end{aligned}$$

In addition to the keys, $\mathcal{E}_\phi$ transmits the ledger height t, and the digests of the inputs and state, with which it checked policy compliance. § V-G shows how we use t to defend against a malicious $\mathcal{P}_c$, who provides a stale view of L to $\mathcal{E}_\phi$.

Similar to $\mathcal{E}_\phi$, $\mathcal{E}_f$ places the encrypted inputs and state in non-enclave memory, and performs decryption on demand. $\mathcal{E}_f$ also places the outputs in non-enclave memory, and performs state updates in-place by directly modifying the data structure holding the prior state. Throughout the evaluation of f, $\mathcal{E}_f$ maintains the digest of the state δ_s and the output digests $\delta_o^1, \ldots, \delta_o^n, \delta_o^{\mathsf{c}_1}, \ldots, \delta_o^{\mathsf{c}_\mathsf{t}}$ (which includes the digests of chained outputs), as they are being written. As $\mathcal{P}_c$ is responsible for storing encrypted inputs, state, and outputs, we emphasize the "bring-your-own-storage" paradigm embodied in LucidiTEE.

A randomized f needs an entropy source. Using the persistent secret key k_s (which was generated by $\mathcal{E}_k$ using the TEE's entropy source (e.g. `rdrand` on x64)), f can internally seed a pseudo-random generator (e.g. PRF with key $\mathsf{t} \oplus \mathsf{k}_s$) to get a fresh pseudo-random bitstream at each step (hence the xor with t). This ensures that the random bits are private, yet allows replaying computation from the ledger during crash recovery.

G. Recording Computation

Since history-based policies need all computation steps to be logged, $\mathcal{E}_f$ must record the evaluation of f on L before it can reveal the output to any party. To that end, $\mathcal{E}_f$ invokes:

$$\begin{aligned}
&\mathsf{L}.\mathsf{post}(\ \mathsf{HW.Quote}(\texttt{compute} \parallel \mathsf{c}.\texttt{id} \parallel \mathsf{t} \parallel \hat{\delta})\ ) \\
&\text{where } \hat{\delta} \doteq \delta_s \parallel \delta_i^1, \ldots, \delta_i^m \parallel \delta_o^1, \ldots, \delta_o^n, \delta_o^{\mathsf{c}_1}, \ldots, \delta_o^{\mathsf{c}_\mathsf{t}}
\end{aligned}$$

Recall that t was given by $\mathcal{E}_\phi$, and it represents the height of L 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 (let us call the new height t′). This can be caused by a combination of reasons including: 1) ledger entries from concurrent evaluations within c and other computations in LucidiTEE, 2) time delay from executing ϕ and f, and 3) malicious $\mathcal{P}_c$ causing $\mathcal{E}_\phi$ to evaluate ϕ using a stale view of L. Consequently, $\mathcal{E}_\phi$'s compliance check is rendered invalid, and serializability of the stateful computation — computations on LucidiTEE, even when executing concurrently, is equivalent to some serial execution — is violated. To correct this, LucidiTEE imposes a validity predicate on the ledger's content, which we call the *serializability check*:

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

Here, L checks that the computation c is still active and that no other function evaluation (with c.id) is performed in between t and the current height t′. The computation is rejected if the check fails, and no entry is recorded on L. $\mathcal{E}_\phi$ and $\mathcal{E}_f$ reject a ledger (i.e., they refuse to compute) that does not satisfy the serializability check. Therefore, the ledger (more accurately, the ledger's participants) are expected to deny entries that do not satisfy serializability — however, they are not trusted to do so, as each computation's $\mathcal{E}_\phi$ repeats the check internally.

H. 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 a different computation in the future, thereby forming a (directed, acyclic) graph composition of computations. For instance, Alice may wish to compute a joint monthly report including Bob's transactions, by entering into a new computation involving Alice, Bob, and Acme. To support compute chaining, we introduce the following modifications to the protocols discussed so far.

First, in addition to the serializability check in § V-G, we assert that no evaluation is performed between t and t′ on a computation c* whose output is consumed by c, as follows:

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

Second, we modify $\mathcal{E}_k$ to maintain an additional key $\mathsf{k}_\mathsf{c}^{\mathsf{c}^*}$ for encrypting the chained output from c* to c (which must not be visible to any party). $\mathcal{E}_k$ generates $\mathsf{k}_\mathsf{c}^{\mathsf{c}^*}$ when either c* or c is created, and transmits $\mathsf{k}_\mathsf{c}^{\mathsf{c}^*}$ (along with $\{\mathsf{k}_i^1, \ldots, \mathsf{k}_i^m\}$ and k_s) to $\mathcal{E}_\phi$ and $\mathcal{E}_f$ during policy checking and the evaluation.

VI. Guaranteeing Fairness

Once the computation is recorded on the ledger, the compute enclave $\mathcal{E}_f$'s final task is to fairly deliver the output to all n output recipients $\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n$ listed in c.out. Fairness is non-trivial as the enclave's communication with the external world is controlled by an untrusted compute provider $\mathcal{P}_c$, who may collude with any party. LucidiTEE implements a

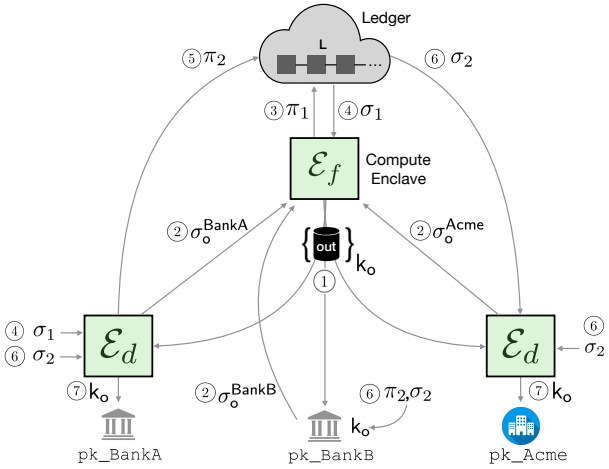


Fig. 3: Fair output delivery using enclaves and a shared ledger

novel protocol for fair n-party broadcast. Although fairness is impossible in the standard setting with dishonest majority [29], our protocol makes additional assumptions by using TEE and ledger functionalities, and ensures fairness and correctness (i.e., all parties receive the same output) even when any subset of the $n+1$ parties (i.e., $\mathcal{P}_c, \mathcal{P}_o^1, \ldots, \mathcal{P}_o^n$) act maliciously. The protocol, illustrated in Figure 3, requires all n output recipients to be online, and $n-1$ of them to possess a TEE machine, on which they launch a *delivery enclave* $\mathcal{E}_d$ — in contrast, prior work [16] required all parties to possess a TEE machine.

Though computations have multiple outputs, each of which are sent to some subset of the n output recipients, we discuss the simpler setting of delivering 1 output to all n parties, for exposition. By encrypting the output under a fresh random key k_o, the problem reduces to fair n-party broadcast of k_o. The core insight underlying our protocol is a construct or a trigger that "simultaneously" allows all parties to attain the key. We implement this construct by posting on the ledger L a value that serves two purposes: 1) it contains the encryption of k_o under the public key of the non-TEE party $\mathcal{P}_o^n$, and 2) it triggers the $n-1$ $\mathcal{E}_d$ enclaves to release k_o to their local parties. The rest of this section outlines the steps of the protocol.

Signed Delivery of Encrypted Outputs and Keys: $\mathcal{E}_f$ provisions k_o to all $n-1$ $\mathcal{E}_d$ enclaves, who store it internally.

$$\mathcal{E}_d \to \mathcal{E}_f : \mathsf{HW.Quote}(\mathsf{c.id} \parallel \mathsf{S.Sig}(\mathsf{sk}_{\mathcal{P}_o^*}, \mathsf{epk}) \parallel \mathsf{epk})$$
$$\mathcal{E}_f \to \mathcal{E}_d : \mathsf{HW.Quote}(\mathsf{c.id} \parallel \mathsf{t} \parallel \mathsf{PKE.Enc}(\mathsf{epk}, \mathsf{k}_o))$$

$\mathcal{P}_c$ delivers the encrypted output $\mathsf{Enc}(\mathsf{k}_o, \mathsf{out})$ (of digest δ_o) to $\mathcal{P}_o^* \in \{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n\}$, and collects their receipt signatures.

$$\mathcal{P}_c \to \mathcal{P}_o^* : \mathsf{HW.Quote}(\mathsf{c.id} \parallel \mathsf{t} \parallel \delta_o) \parallel \mathsf{Enc}(\mathsf{k}_o, \mathsf{out})$$
$$\mathcal{P}_o^* \to \mathcal{P}_c : \mathsf{S.Sig}(\mathsf{sk}_{\mathcal{P}_o^*}, \mathsf{c.id} \parallel \mathsf{t} \parallel \delta_o)$$

Posting Signatures on the Ledger: $\mathcal{P}_c$ (on behalf of $\mathcal{E}_f$) aggregates (e.g. using [30]) and posts all n signatures on L:

$$\mathcal{P}_c : \mathsf{L.post}(\pi_1), \text{ which returns } (\sigma_1, \_)$$
$$\pi_1 \doteq \mathsf{S.Sig}(\mathsf{sk}_{\mathcal{P}_o^1}, \mathsf{c.id} \| \mathsf{t} \| \delta_o) \parallel \ldots \parallel \mathsf{S.Sig}(\mathsf{sk}_{\mathcal{P}_o^n}, \mathsf{c.id} \| \mathsf{t} \| \delta_o)$$

As an optimization, we combine π_1 with the ledger record of the computation (from § V-G) to reduce our usage of L.

Posting Encryption of Key k_o on Ledger: Any of the $n-1$ parties controlling an $\mathcal{E}_d$ enclave, on seeing π_1 on the ledger, can advance the protocol to its next phase — when $n = 1$, $\mathcal{P}_c$ launches a $\mathcal{E}_d$ enclave to fulfill this role. On providing σ_1 (which was produced by L upon posting π_1), $\mathcal{E}_d$ emits π_2, containing an encryption of k_o under $\mathcal{P}_o^n$'s public key. Any $\mathcal{P}_o^* \in \{\mathcal{P}_o^1, \ldots, \mathcal{P}_o^{n-1}\}$ (or $\mathcal{P}_c$ when $n = 1$) can post π_2 on L.

$\mathcal{P}_o^*$: $\mathsf{L}.\mathsf{post}(\pi_2)$, which returns $(\sigma_2, \_)$

$\pi_2 \doteq \mathsf{HW}.\mathsf{Quote}(\texttt{deliver}\|\texttt{c.id}\|\texttt{t}\|\mathsf{PKE}.\mathsf{Enc}(\mathsf{pk}_{\mathcal{P}_o^n}, \mathsf{k}_o))$

Decrypting the Output: Finally, all parties $\mathcal{P}_o^1, \ldots, \mathcal{P}_o^n$ must decrypt the output. The $n-1$ parties with $\mathcal{E}_d$ provide the proof σ_2 (which was produced by L upon posting π_2) to their local $\mathcal{E}_d$ enclaves, allowing those enclaves to emit k_o. $\mathcal{P}_o^n$ simply retrieves π_2 from L and decrypts $\mathsf{PKE}.\mathsf{Enc}(\mathsf{pk}_{\mathcal{P}_o^n}, \mathsf{k}_o)$ to attain k_o. All parties now decrypt the output using k_o.

VII. Discussion

Assuming ideal functionalities for secure hardware HW and append-only ledger L, LucidiTEE $\mathcal{F}_{PCFC}$, thus providing policy-compliant fair computation to all parties. However, we draw attention to the subtlety of different blockchain instantiations. While our fair delivery protocol tolerates a corruption threshold of n amongst $n+1$ 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 fairness (of message delivery or 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 a prior work [16] 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 [5], which uses a trusted timeserver to enforce the rate of production of entries on the ledger, offers a probabilistic guarantee of rollback prevention, which worsens as the attacker's computational power increases — a silo'd powerful adversary can always mine ledger entries. Hence, we deploy LucidiTEE on forkless ledgers (providing the L abstraction), such as HyperLedger [23] and Tendermint [31].

LucidiTEE does not support fair reactive computation, and is not suitable for applications such as Poker [1]. The primary issue here is that the compute provider $\mathcal{P}_c$ may collude with any party to abort the computation and destroy its intermediate state, thus preventing any honest party from making progress.

VIII. Implementation

We implement LucidiTEE with a heavy focus on modularity and minimality of the trusted computing base.

We instantiate the shared ledger with a permissioned blockchain, and evaluate using both Hyperledger [23] and Tendermint [31]. Each participant's logic is implemented as a smart contract (in 200 lines of Go code), which internally uses a key-value store (RocksDB [32]) to store the ledger's entries (indexed by the computation's `id`, as described in § V-E).

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 § V-B — 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 $\mathsf{Verify}_\mathsf{L}$) and TEE attestation of all ledger entries (of type `compute` and `deliver`). libmoat contains 3K LOC, in addition to Intel's SGX SDK [33].

IX. Evaluation

A. Case Studies

We demonstrate several applications where multiple distrusting parties collaborate on large volumes of sensitive data, while attaching policies governing the use of their data and fairly attaining the output of the joint computation. In addition to the applications described below, we build micro-benchmarks such as one -time programs [15], digital lockboxes (with limited passcode attempts), and a secure logging service.

1) Private Personal Finance Application: Acme's finance application is implemented on LucidiTEE using the specification shown below. Observe that 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, and the policy from computation 3 asserts that the input reports belong to the same calendar month. Acme's input is encoded as an encrypted key-value database, indexed by the merchant id — with over 50 million merchants worldwide that accept credit cards, 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 [8] to download the author's transactions from the bank's OFX service, encrypt them locally (to size 1.3MB), and upload the encrypted file to a public AWS bucket.

```
computation {
  id : 1,
  inp : [("txs":vk_Alice), ("db":vk_Acme)],
  out : [ ("report" : [(pk_Acme,vk_Acme),
          (pk_BnkA,vk_BnkA),(pk_BnkB,vk_BnkB)])
        ,("report" : (3, "alice_report")) ],
  policy : 0xcoff..eee, /* ∀r ∈ txs. fresh(r)*/
  func : 0x1337...c0de /* aggregate function */
}
computation {
  id : 3,
  inp : [ ("alice_report" : (1, "report")),
          ("bob_report" : (2, "report")),
  out : [ ("joint_report" : [pk_Acme]) ],
  policy : 0xc000...10ff, /* same month? */
  func : 0x1ce..b00da /* compare function */
}
```

2) Policy-based Private Set Intersection: We implement a patient information sharing service described in [34]. Two hospitals A and B wish to share prescription records about their common patients who have been prescribed both drugs 1 and 2, under the policy-encoded restrictions: 1) both hospitals have the patient's consent (due to HIPAA); 2) the patient must have been treated by both drugs. Since both hospitals wish to benefit from this service, they require a guarantee of fair output delivery. We implement oblivious set intersection by adapting Signal's enclave-based private contact discovery service [35]. Our experiment uses a synthetic dataset with 1 million patient records in each hospital's database (size 15MB).

```
computation {
  id : 4,
  inp : [("A_db":vk_A), ("B_db":vk_B)],
  out : [ ("ab" : [(pk_A,vk_A),(pk_B,vk_B)]) ],
  policy : .., /*cond 1 & 2, and execute once*/
  func : 0x7331...ed0c /* oblivious psi */
}
```

3) Federated Machine Learning: Two organizations, A and B, contribute private data to train a joint model. Since neither party sees the other's data, we use a policy that a model locally trained on one party's data alone must have sufficient accuracy. Note that federated learning is an active area of research, with schemes providing strong privacy (e.g. differential privacy); our goal instead is to study the performance implications (rather than the privacy guarantees) of such applications.

In our experiment, we have two hospitals, A and B, that contribute patient records containing sensitive attributes and ECG measurements, with the goal of learning a better model for predicting the ECG class of a patient. We split the Arrhythmia dataset from the UCI Machine Learning Repository [36] into two halves, one for each hospital, and adapt the k-means clustering algorithm and implementation from [37].

```
computation {
  id : 5,
  inp : [ ("recs_A":vk_A), ("recs_B":vk_B) ],
  out : [ ("model":[(pk_A,vk_A),(pk_B,vk_B)])],
  policy : 0xdaff..0d11, /* check accuracy */
  func : 0xf1e...1d5 /* k-means clustering */
}
```

4) Fair Information Exchange: Two parties perform a fair exchange (denoted FX) of their private inputs (of size 1 KB).

```
computation {
  id : 6,
  inp : [("x" : vk_A), ("y" : vk_B)],
  out : [("o": [(pk_A,vk_A), (pk_B,vk_B)])],
  policy : 0xbad..beef, /* ψ_Alice(y) ∧ ψ_Bob(x) */
  func : 0xface...b00c /* f(x,y) = (y,x) */
}
```

Each party provides their private input, with along with a condition ψ that the other party's input must satisfy. The program only proceeds if both parties' conditions are met, and fairly delivers the output to both parties. Note that while the above policy is simplified to only include 2 parties, LucidiTEE allows any number of parties (≥ 2) to perform fair exchange.

B. Performance Measurement

We study the latency, throughput, and storage requirement of our applications, and compare to a baseline version where the application runs without a ledger (i.e. without our protocols for recording computation or output delivery) and without the protections of TEE (i.e., we run the application in SGX simulation mode). The baseline versions also inline the policy checking logic ϕ within the compute function f. In this baseline setting, the Acme, PSI, ML, and FX apps take 0.2, 8.24, 0.11, and 0.06 seconds, respectively, on each invocation.

1) End-to-end Latency and Throughput: Figure 4 reports the latency and throughput (results aggregated over 100 runs) on both HyperLedger [23] and Tendermint [31] 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. Recall that each evaluation on LucidiTEE performs at least one read query (to retrieve the specification and the `compute` records) and two writes (to record the `compute` and `deliver` entry) to the ledger. We found throughput to be bound by the performance of the ledger, which was highly dependent on parameters such as the batch size and batch timeout [23]. Compared to the baseline, the latency also suffered by several seconds due to the ledger, quite likely due to the high volume of concurrent read and write requests.

2) Storage: Table III presents the off-chain and ledger storage needs (where the measurement is for each function evaluation within that application). In comparison, Ethereum [38] and Ekiden [5] store the inputs and the state on the ledger.

X. Related Work

Secure Computation based on Enclave-Ledger Interaction

Ekiden [5], Coco [39], and Private Data Objects [40] are the most closely related works, in that they rely on shared ledger and trusted hardware functionalities. They execute smart contracts (e.g. Ekiden executes Ethereum bytecode) within SGX enclaves, connected to a blockchain for persisting

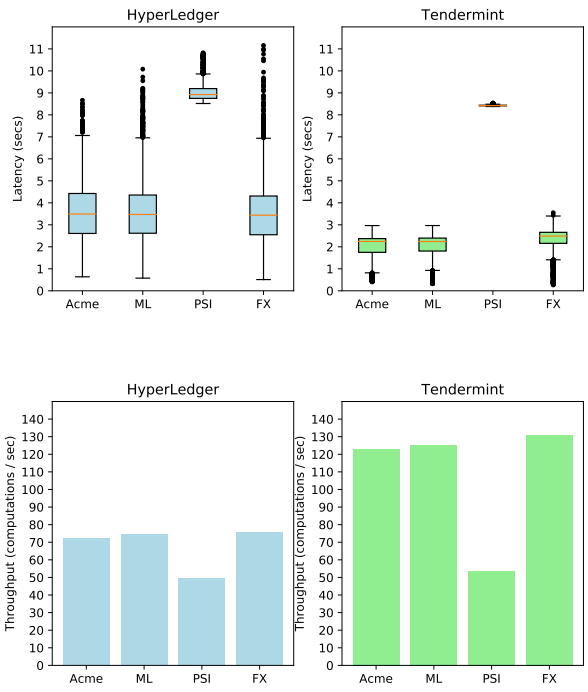


Fig. 4: End-to-end latency and throughput measurement

TABLE III: Storage Requirements

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	-
2-party Fair Exchange	2052 B	2 KB	2 KB	-

the contract's state. LucidiTEE pursues a different goal of enforcing policies on behalf of offline users (allowing their inputs to be used in several multi-step computations, without interaction) and fairness. The main improvements over Ekiden lie in fairness and scalability — LucidiTEE does not put 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, their ideal functionalities have the following differences: 1) all parties in Ekiden must interact with the functionality on each state transition to provide inputs and receive outputs (like Ethereum); 2) an attacker can prevent the ideal functionality from fairly sending the output to a party. To our knowledge, these differences carry over to Coco and Private Data Objects.

Hawk [4] enables parties to perform off-chain computation (specified as smart contracts) with transactional privacy (which protects the inputs), while proving correctness by posting zero knowledge proofs on the ledger. Compared to LucidiTEE, Hawk incurs significant computational overhead of generating zero-knowledge proofs (specifically zk-SNARKS [41]), is not suitable for computing on large databases (computation must be expressed as arithmetic circuits), and only provides financial fairness (i.e., fairness with penalties to honest parties, as opposed to perfect fairness in LucidiTEE). On that note, several works prior to Hawk, specifically Bentov et al. [1], Kumaresan et al. [42], Andrychowicz et al. [43], and Kaiyias et al. [44], use Bitcoin [45] to ensure financial fairness in MPC protocols, by constructing abstractions such as "multi-lock" and "claim-or-refund" transactions. Pass et al. [9] develop a protocol for 2-party fair exchange in the Δ-fairness model.

Recently, Choudhuri et al. [16] 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 $n-1$ 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, [16] requires all parties to be online, and only considers secure function evaluation [46] (i.e., one-shot MPC) as opposed to stateful computation, which incurs added complexity in ensuring privacy, integrity, and safe concurrency. A related work [47] augments stateless enclaves with shared ledgers to perform secure stateful computation, addressing well-known issues of state rollback (i.e., rewind-and-fork) attacks. However, they do not support computation amongst multiple parties or offline users, nor ensure fairness.

Data Processing Systems based on Trusted Hardware

Trusted execution environments (e.g. Intel SGX) have recently found use in systems for outsourced computing: M2R [48] and VC3 [7] for Map-Reduce jobs, Opaque [49] for analytics, EnclaveDB [50] for SQL databases, 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 the policy enforcement and fair output delivery.

Secure Computation from Cryptographic Assumptions

MPC [2] [3] 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 [29], fairness is also impossible for general functions in the standard model with dishonest majority (our fair exchange relies on trusted hardware and shared ledgers). Goyal et al. [51] show how blockchains can implement one time programs using cryptographic obfuscation techniques.

XI. Conclusion

This paper presented LucidiTEE, the first system to enable multiple parties to jointly compute on private data, while enforcing policies even when the participants are offline. Novel protocols between TEE nodes and a shared ledger ensure that all computations are fair, recorded on the ledger, and compliant with user-defined, history-based policies. The ledger is only used to record computations and enforce policies (i.e., it does not store inputs, outputs, or state, nor do the ledger's participants repeat the computation), allowing LucidiTEE to scale to big data applications and large number of participants.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[9] 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.

[10] 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.

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

[12] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing."

[13] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 265–310, Nov. 1992.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] R. Tamassia, "Authenticated data structures," in *Algorithms - ESA 2003*, G. Di Battista and U. Zwick, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–5.

[29] 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.

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

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

[32] "RocksDB: A persistent key-value store for Flash and RAM storage," https://github.com/facebook/rocksdb.

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

[34] E. Stefanov, E. Shi, and D. Song, "Policy-enhanced private set intersection: Sharing information while enforcing privacy policies," in *Public Key Cryptography*. Springer Berlin Heidelberg, 2012, pp. 413–430.

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

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

[37] 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.

[38] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger."

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] Z. Beerliová-Trubíniová, M. Fitzi, M. Hirt, U. Maurer, and V. Zikas, "Mpc vs. sfe: Perfect security in a unified corruption model," in

Proceedings of the 5th Conference on Theory of Cryptography, ser. TCC'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 231–250.

[47] 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.

[48] 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.

[49] 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, USA: USENIX Association, 2017, pp. 283–298.

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

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